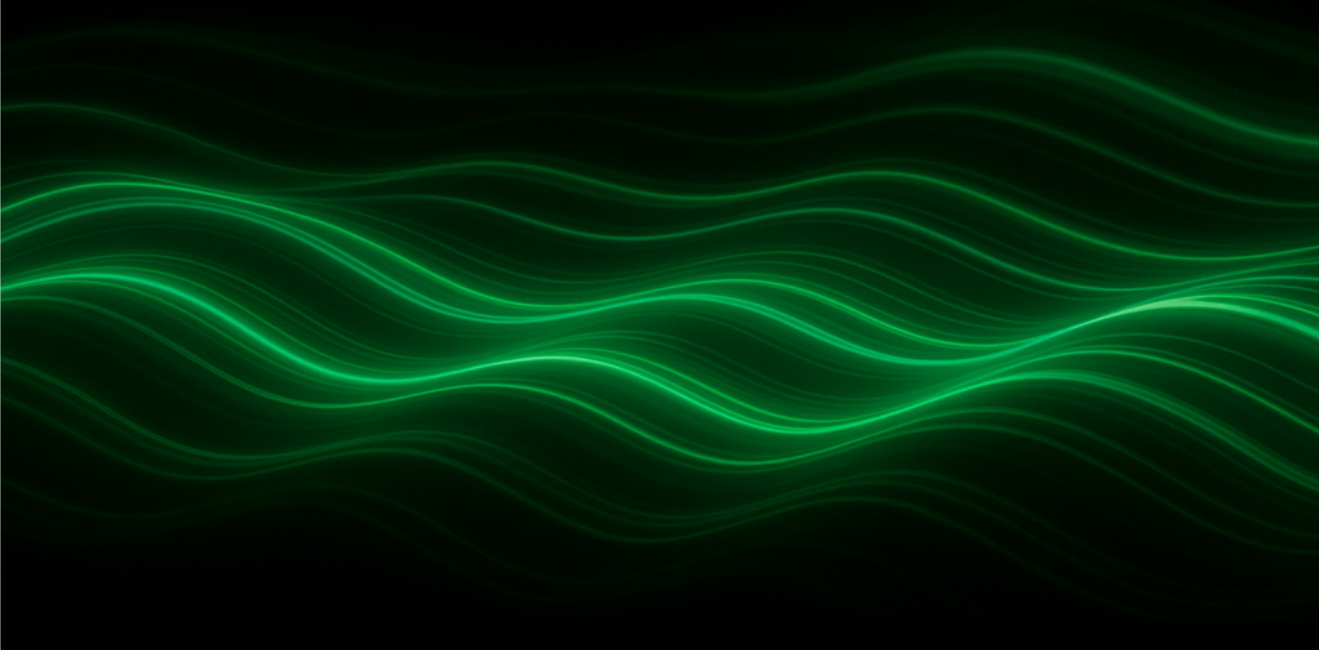
Algorithm

saturngod



မိတ်ဆက်

ဒီစာအုပ်က အဓိက self study သမားတွေ အတွက် ရည်ရွယ်ပြီး ရေးသားသည့် စာအုပ် တစ်အုပ်ပါပဲ။ ပုံမှန်အားဖြင့် algorithm က Computer တက္ကသိုလ်တွေမှာ ပထမ နှစ်ကတည်း က စတင်ခဲ့ပါတယ်။ သို့ပေမယ့် ပြည်တွင်း က သင်တန်းကျောင်း အများစုက လုပ်ငန်းခွင် ဝင်ရောက်နိုင်ရန် အဓိက ထားသင်သည့် အတွက် algorithm ဟာ နားလည်ရခက်သည့် ဘာသာရပ် တစ်ခု interview ဖြေဖို့ လေ့လာရသည့် ဘာသာရပ် တစ်ခုလို့ အမှတ်မှားနေကြတာ ရှိပါတယ်။ Algorithm စာအုပ် တော်တော် များများ ဟာ စာတွေ့ဘက်ကို အားပြုရေးထားကြပြီး ကျောင်းက သင်ရိုးပုံစံ တွေ ဖြစ်နေတာကြောင့်လည်း လေ့လာရတာ အနည်းငယ် ခက်ခဲမှု ရှိတာ အမှန်ပါ။

ဒီစာအုပ် မှာ စာတွေ့နှင့် လက်တွေ့ မျှတ မှု ရှိအောင် ကြိုးစားပြီး ဖော်ပြသွားပါမယ်။ ဒီစာအုပ် ကျွန်တော် Master တက်တုန်းက ကျောင်းမှာ သင်ခဲ့ရသည့် Alogirthm course ကို မှီငြမ်းထားပါတယ်။ AVL Tree, Red-Black Tree တွေက အနည်းငယ် ရှုပ်ထွေးမှု ရှိပြီး interview တွေမှာလည်း မေးခဲ့သည့် အတွက် ဒီစာအုပ်မှာ ချန်ထားခဲ့ပါတယ်။

algorithm ဆိုတာ ပြဿနာ ဖြေရှင်းဖို့ နည်းလမ်း ပါပဲ။ တနည်းပြောရင် တွေးတော အဖြေရှာခြင်း ပါပဲ။ ဘယ်လို ပြဿနာ တွေကို ဘယ်လို ဖြေရှင်းလို့ရတယ်ဆိုတာ ကို နားလည် သဘောပေါက်လာသည့် အခါမှာ coding ရေးသည့် အခါမှာလည်း ပိုမို ကောင်းမွန်စွာ ရေးသားလာနိုင်မယ်။ နောက်တစ်ချက်က Big O ကို နားလည် လာသည့် အခါမှာ ဘာကြောင့် ဒီလို ရေးတာ ပိုကောင်းတယ်ဆိုတာ ကို သဘောပေါက်လာနိုင်ပါလိမ့်မယ်။

ဒီစာအုပ်မှာ Algorithm နှင့် သက်ဆိုင်ရာ အစ အဆုံး ပါဝင်ဖို့ရာ မဖြစ်နိုင်သည့် အတွက် လူသုံးများသည့် အပိုင်း interview တွေ မှာ အမေးများသည့် အပိုင်းတွေကို အဓိက ထည့်သွင်းရေးသားသွားပါမယ်။ ဒီစာအုပ်မှာ Java ကို ပဲ အသုံးပြုသွားပါမယ်။ Algorithm ပိုင်းကို ရှင်းပြဖို့ အတွက် Java language ကသာ အသင့်တော်ဆုံးပါပဲ။ ဒါကြောင့် လေ့လာသူ၏ စက်ထဲမှာ Java အဆင်သင့် ရှိနေဖို့ လိုပါတယ်။ ဒီစာအုပ် က Java စာအုပ်မဟုတ်သည့် အတွက် Java သွင်းနည်း Java run နည်းတွေကို ပါဝင်မှာ မဟုတ်ပါဘူး။ လေ့လာသူ က Java ကို ကိုယ့်စက်ထဲမှာ ဘယ်လို run ရသလဲ ဆိုတာ သိထားပြီးသား သူလို့ သတ်မှတ်ပြီး ရေးသားသွားမှာ ဖြစ်ပါတယ်။

ကျွန်တော် အမြဲယုံကြည်တာကတော့ Theory နှင့် အခြေခံ ပိုင်နိုင် သွားသည့် အခါ နောက် တဆင့် တက်ဖို့ ခက်ခဲမှု မရှိပါဘူး။ ဒီစာအုပ်ဟာ Mid Level Developer တွေ ကို အထောက်အကူ ပြုသည့် စာအုပ် တစ်အုပ် ဖြစ်လာမယ် လို့ မျှော်လင့်ပါတယ်။

အခန်း ၁ - Algorithms မိတ်ဆက်

Computer Science ဘာသာရပ်ကို လေ့လာလိုက်စားသည့် သူတွေ အတွက် Algorithm ဆိုတာ မသိမဖြစ် လေ့လာရမည့် ဘာသာရပ် တစ်ခု ဖြစ်ပါတယ်။ Developer တစ်ယောက်ရဲ့ အရည်အချင်း၊ သူ ဘယ်လောက်ထိ စဉ်းစားတွေးခေါ်နိုင်ပြီး coding ပြဿနာတွေကို ဖြေရှင်းနိုင်သလဲဆိုတာကို algorithm ပုစ္ဆာများဖြင့် interview များတွင် မကြာခဏ တိုင်းတာလေ့ရှိပါတယ်။

Algorithm ဆိုတာ ဘာလဲ

Algorithm ဆိုတာ ပြဿနာတစ်ခုကို ဖြေရှင်းရန်အတွက် စနစ်တကျ သတ်မှတ်ထားသော အဆင့်ဆင့် လုပ်ဆောင်ရမည့် ညွှန်ကြားချက်များဖြစ်ပါတယ်။ ဥပမာ အားဖြင့် ကျွန်တော်တို့ နေ့စဉ် ဘဝ တွေ မှာ ဟင်းချက်တာ ဖြစ်ဖြစ် ကားမောင်းတာ ဖြစ်ဖြစ် အဆင့်ဆင့် လုပ်ဆောင်ရပါတယ်။ ဥပမာ ဟင်းချက်နည်း တစ်ခု က Algorithm တစ်ခုပါပဲ။

သို့ပေမယ့် computer algorithms တွေဟာ အောက်ပါ ဂုဏ်သတ္တိတွေ ရှိရပါမယ်။

1. Input : လုပ်ဆောင်ရမည့် အချက်အလက်တွေ ရှိရမည်။
2. Definiteness : အဆင့်တိုင်းဟာ ရှင်းလင်း တိကျရမည်။
3. Finiteness: အဆုံးမရှိ ဘဲ လုပ်ဆောင်နေတာမျိုး မဟုတ်ဘဲ တစ်နေရာမှာ အလုပ်ပြီးမြောက်ပြီး ရပ်တန့် နေရမည်။
4. Output: အနည်းဆုံး ရလဒ် တစ်ခု ထွက်လာရမည်။
5. Effectiveness: လက်တွေ့ လုပ်ဆောင်လို့ရသည့် အဆင့်တွေ ဖြစ်ရမည်။

Algorithm ဘာကြောင့်အရေးကြီးလဲ

ယနေ့ခေတ်မှာ Software တွေဟာ ရှုပ်ထွေးလာပါတယ်။ ဥပမာ Facebook မှာ Friend Suggestion, Google Map မှာ route ပြတာ စသည့် စနစ်တွေ ရဲ့ နောက်ကွယ်မှာ အလွန် ရှုပ်ထွေးပြီး အရမ်းကို ကောင်းမွန်သည့် algorithm တွေ အလုပ်လုပ်နေပါတယ်။ ကျွန်တော်တို့တွေ က နည်းလမ်း မမှန်ဘဲ code တွေကို ရေးလိုက်မယ် ဆိုရင် process တစ်ခုက ပုံမှန်ထက် လုပ်ဆောင်ရမည့် အလုပ်ပမာဏ များလာပြီး နှေးသွားတာ ဒါမှမဟုတ် ပိုပြီး powerful ဖြစ်သည့် စက်တွေ လိုအပ်တာမျိုးတွေ ဖြစ်လာနိုင်တယ်။ ဒါကြောင့် algorithm ဟာ အလုပ်လုပ် ရုံတင်မကဘဲ အကောင်းဆုံး နဲ့ အမြန်ဆုံး အလုပ်လုပ်ဖို့ အတွက် လေ့လာရခြင်း ဖြစ်ပါတယ်။

အခန်း ၂ - Big-O, Time Complexity, Space Complexity

Algorithm ဘယ်လောက်ကောင်းသလဲဆိုတာကို စက္ကန့် နဲ့ မတိုင်းတာပါဘူး။ ဘာလို့လဲဆိုတော့ computer တစ်လုံး နဲ့ တစ်လုံးဟာ performance အမြန်နှုန်း မတူညီလို့ပါ။ ဒါကြောင့် အချက်အလက် ပမာဏ (N) များလာရင် algorithm က ဘယ်လောက်ထိ အလုပ်လုပ်ရမလဲ ဆိုတာကို သင်္ချာနည်းအရ **Asymptotic Notations** များ အသုံးပြုပြီး တိုင်းတာပါတယ်။

ဒီနေရာမှာ သဘောတရား **နှစ်ခု** ကို ခွဲမြင်ဖို့ အရေးကြီးပါတယ်။ ပထမက input အပေါ်မူတည်တဲ့ **အခြေအနေ (Best / Average / Worst Case)** ၊ ဒုတိယက growth rate bound ကို ဖော်ပြသည့် **Notation (O, Ω, Θ)** ပါ။

၁။ Input အခြေအနေ (Best / Average / Worst Case)

Algorithm တစ်ခုဟာ input ပေါ်မူတည်ပြီး အလုပ်လုပ်ရတဲ့ အကြိမ်အရေအတွက် ကွဲပြားနိုင်ပါတယ်။ ဒါကို အခြေအနေ (၃) မျိုး ခွဲပါတယ်။

- **Best Case (အကောင်းဆုံး):** ကံအကောင်းဆုံး input ။ ဥပမာ Linear Search နဲ့ Array `[5, 2, 9, 1, 7]` ထဲက `5` ကို ရှာရင် ပထမဆုံးနေရာမှာပဲ ချက်ချင်းတွေ့ပါတယ်။ အကြိမ် ၁ ကြိမ်ပဲ လုပ်ရတယ်။
- **Worst Case (အဆိုးဆုံး):** ကံအဆိုးဆုံး input ။ အပေါ်က Array ထဲမှာ `10` ကို ရှာရင် အကုန်ရှာပြီးမှ မတွေ့တာမျိုး ဖြစ်နိုင်ပါတယ်။ N ကြိမ် လုပ်ရပါတယ်။
- **Average Case (ပျမ်းမျှ):** input အမျိုးမျိုးအတွက် ပျမ်းမျှ ကြာချိန်။ Linear Search အတွက် ပျမ်းမျှ $N/2$ ကြိမ်လောက် လုပ်ရပါတယ်။

> Software Engineer တွေဟာ **အဆိုးဆုံး အခြေအနေ (Worst Case)** ကို ကြိုတင် မျှော်မှန်းထားရတဲ့အတွက် လက်တွေ့မှာ Worst Case ကို အဓိက သုံးကြပါတယ်။

၂။ Asymptotic Notations (O, Ω, Θ)

ဒီ notation (၃) ခုက input အခြေအနေ မဟုတ်ပါဘူး။ growth rate ကို **သင်္ချာနည်းအရ ချုပ်ဆို (bound)** တဲ့ နည်းတွေ ဖြစ်ပါတယ်။ အခြေအနေ တစ်ခုခု (best / average / worst) ရဲ့ ကြာချိန်ကို ဒီ notation တွေနဲ့ ဖော်ပြလို့ ရပါတယ်။

Big O Notation - O (Upper Bound)

ဒါကို **Upper Bound** လို့ ခေါ်ပါတယ်။ Algorithm တစ်ခုက အများဆုံး ဒီ growth rate ထက် မပိုနိုင်ဘူး ဆိုတဲ့ အပေါ်ဆုံး ကန့်သတ်ချက် ဖြစ်ပါတယ်။ ဥပမာ Linear Search ရဲ့ worst case ကို $O(N)$ လို့ ဖော်ပြပါတယ်။

Big Omega Notation - Ω (Lower Bound)

ဒါကို **Lower Bound** လို့ ခေါ်ပါတယ်။ Algorithm တစ်ခုက အနည်းဆုံး ဒီ growth rate တော့ ကြာမယ် ဆိုတဲ့ အောက်ဆုံး ကန့်သတ်ချက် ဖြစ်ပါတယ်။ ဥပမာ Linear Search ရဲ့ best case ကို $\Omega(1)$ လို့ ဖော်ပြပါတယ်။

Big Theta Notation - Θ (Tight Bound)

upper bound နဲ့ lower bound နှစ်ခုလုံးက တူညီတဲ့အခါ Θ notation ကို သုံးပါတယ်။

- **ဥပမာ:** Array `[5, 2, 9, 1, 7]` ထဲက ဂဏန်း အားလုံးရဲ့ ပေါင်းလဒ် (Sum) ကို ရှာမယ်ဆိုရင် ကံကောင်းတာ၊ ကံဆိုးတာ မရှိဘဲ ဂဏန်း (၅) လုံးစလုံးကို အမြဲ ပတ်ရပါတယ်။ best case ရော worst case ရော N ကြိမ် တူတူ ဖြစ်တဲ့အတွက် $\Theta(N)$ လို့ အတိအကျ ဆိုနိုင်ပါတယ်။

> **သတိ:** O (Ω, Θ) တွေက bound တွေ ဖြစ်ပြီး၊ best/average/worst တွေက input အခြေအနေ ဖြစ်ပါတယ်။ သဘောတရား နှစ်မျိုး မတူပါဘူး။ ဒါပေမယ့် လက်တွေ့မှာ worst case ကို Big O (O) နဲ့ ဖော်ပြလေ့ ရှိတဲ့အတွက် developer အများစုက "Big O = worst case" လို့ အလွယ်တကူ မှတ်ထားကြတာ ဖြစ်ပါတယ်။

ဥပမာ

ရန်ကုန် ကနေ မန္တလေး ကို ကားမောင်းသွားတယ် ဆိုပါစို့။ (real world example မဟုတ် ပဲ conceptual example ပါ။)

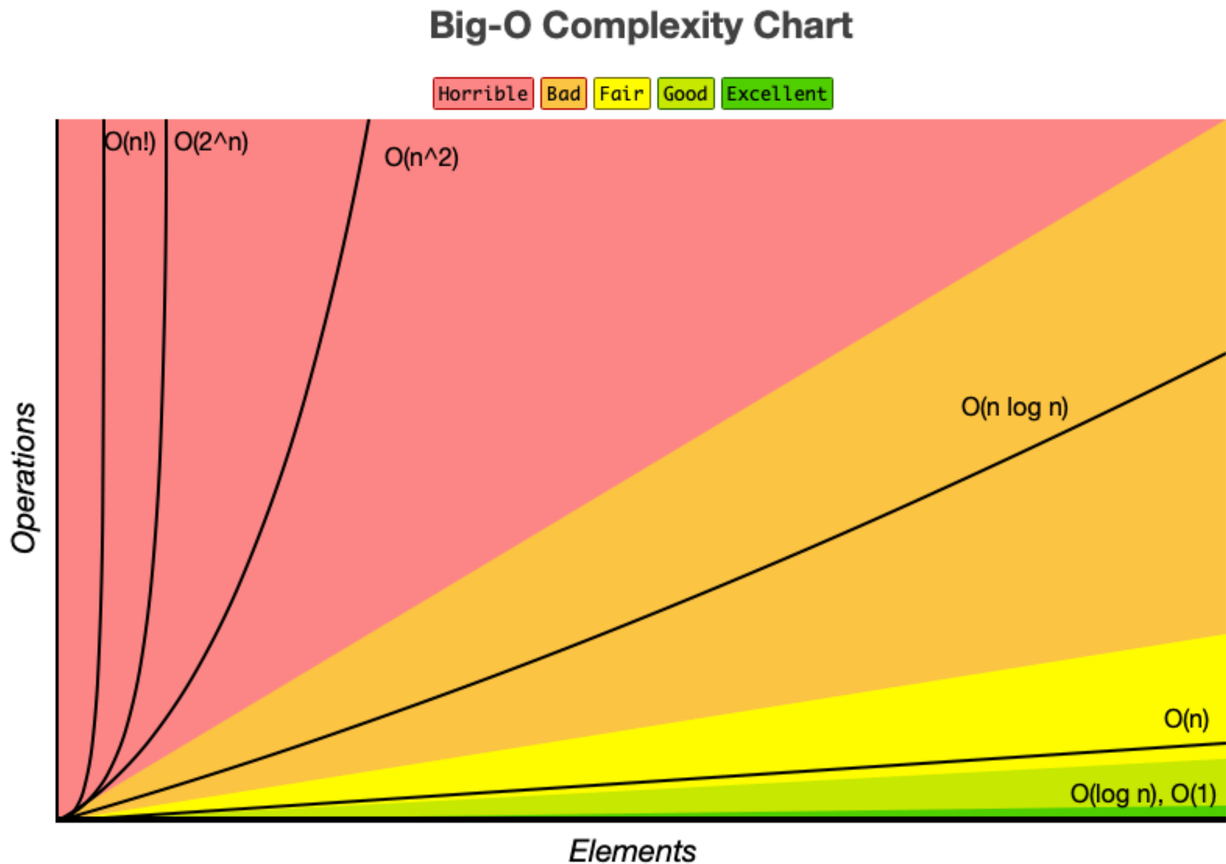
- **best-case runtime:** လမ်းမှာ လုံးဝကားမပိတ်၊ ရာသီဥတုကကောင်း၊ အမြန်ဆုံး အရှိန်နဲ့ သွားမယ် ဆိုရင် အနည်းဆုံး (၈) နာရီ ကြာမယ်။ ဒီ အနိမ့်ဆုံး အချိန်ကို Ω နဲ့ ဖော်ပြနိုင်ပါတယ်။
- **worst-case runtime:** လမ်းမှာ ကားပိတ်တာ၊ မိုးရွာတာ အဆိုးဆုံးကြုံရင်တောင် အဆိုးဆုံး အခြေအနေအဖြစ် (၁၁) နာရီခန့် ကြာနိုင်တယ်လို့ ယူဆပါစို့။ ဒီ အမြင့်ဆုံး အချိန်ကို O နဲ့ ဖော်ပြနိုင်ပါတယ်။
- **tight estimate:** အနိမ့်ဆုံး နဲ့ အမြင့်ဆုံး အချိန်ဟာ တူနေတဲ့ အခါ (ဥပမာ ပုံမှန် မောင်းရင် အမြဲ (၉) နာရီခန့်) အတိအကျ ခန့်မှန်းနိုင်ပါတယ်။ ဒါကို Θ နဲ့ ဖော်ပြနိုင်ပါတယ်။

အသုံးများတဲ့ Big O အမျိုးအစားများ

Computer Science မှာ Big O ဟာ လက်တွေ့ အသုံးအများဆုံးပါ။ အမျိုးအစားတွေကတော့

- $O(1)$ - Constant Time: Data ဘယ်လောက်များများ ကြာချိန် အတူတူပဲ။ ဥပမာ Array Index တစ်ခု ကို ယူတာ
- $O(N)$ - Linear Time : Data များလေလေ အချိန်ကြာလေလေ။ ဥပမာ Array ထဲမှာ အခန်း အစ ကနေ အဆုံး ထိ ရှာဖွေခြင်း

- $O(N \log N)$ - Linearithmic Time: O(N) ထက် အနည်းငယ် ပိုကြာသော်လည်း $O(N^2)$ ထက် အများကြီး ပိုမြန်တယ်။ ဥပမာ Merge Sort ကဲ့သို့သော Algorithm
- $O(N^2)$ - Quadratic Time: input size တိုးလာတာနဲ့အမျှ ကြာချိန်က နှစ်ထပ်ကိန်းအချိုးနဲ့ တိုးလာတယ်။ အဆင့်ဆင့် loop ပတ်နေတာမျိုး
- $O(\log N)$ - Logarithmic Time: Data များလာပေမယ့် ကြာချိန် အနည်းငယ်ပဲ တိုးလာခြင်း။ မြန်ဆန်သည့် စနစ်လို့ ဆိုနိုင်တယ်။



Big O တွက်ချက်ခြင်း

Big O Notation ကို တွက်ချက်သည့် အခါမှာ အဓိက Golden Rule ၃ ခု ရှိပါတယ်။

အဆင့်ဆင့် တွက်ချက်ခြင်း ဥပမာ

Algorithm ကို အဆင့်ဆင့် ဘယ်လို တွက်ချက်လဲဆိုတာ အောက်ပါ Java Code လေးနဲ့ အရင် လေ့လာကြည့်ရအောင်။

```
int sum = 0; // 1
int mul = 1; // 1
for (int i = 0; i < array.length; i++) { // N ကြိမ်
    sum = sum + array[i]; // N ကြိမ်
    mul = mul * array[i]; // N ကြိမ်
}
```

အပေါ်က code မှာ အစကိန်းသေ နှစ်ကြောင်းက ၁ ကြိမ်စီ အလုပ်လုပ်တယ်။ Loop က N ကြိမ် ပတ်တယ်။ Loop အထဲက ကုဒ်တွေကလည်း N ကြိမ်စီ အလုပ်လုပ်တယ်။
စုစုပေါင်းလုပ်ဆောင်ချက်ကို `1 + 1 + N + N + N = 2 + 3N` ဆိုပြီး အကြမ်းဖျင်း ယူဆလို့ရပါတယ်။

Big O Notation ကို သတ်မှတ်တဲ့အခါ ဒီလို ကိန်းဂဏန်းတွေထဲကနေ အဓိက **Golden Rule ၃ ခု** ကို အသုံးပြုပါတယ်။

၁။ ကိန်းသေများကို ပယ်ဖျက်ပါ

အပေါ်က တွက်လဒ် `2 + 3N` မှာ ကိန်းသေ (Constants) တွေဖြစ်တဲ့ အပေါင်း `2` နဲ့အမြှောက် `3` ကို ပယ်ဖျက်ပါတယ်။ ဒါကြောင့် အကျဉ်းချုံးလိုက်ရင် $O(N)$ လို့ပဲ သတ်မှတ်ပါတယ်။ Algorithm တစ်ခုက $O(2N)$ သို့မဟုတ် $O(N+100)$ အဆင့်တွေ ယူတယ်ဆိုရင်လည်း ကိန်းသေ တွေကို ပယ်ဖျက်ပြီး $O(N)$ လို့ပဲ သတ်မှတ်ပါတယ်။ အချက်အလက် ဘယ်လောက် ပွားလာလဲ ဆိုသည့် အချိုးအဆ ကို ပဲ အဓိက ထားလို့ပါ။

၂။ အဓိက မကျသည့် ကိန်းများကို ပယ်ဖျက်ပါ

Algorithm ရဲ့ ကြာချိန်က တွက်လိုက်လို့ $O(N^2+N)$ ဖြစ်နေခဲ့လျှင် N^2 က N ထက် အများကြီး ပို မြန်မြန် ကြီးထွားလာနိုင်သည့် အတွက် အရေးမပါတဲ့ N ကို ပယ်ဖျက်ပြီး $O(N^2)$ လို့ ပဲ ယူပါတယ်။

```
public void printNumbers(int[] array) {
    for (int i = 0; i < array.length; i++) { // O(N)
        System.out.println(array[i]);
    }

    for (int i = 0; i < array.length; i++) { // O(N^2)
        for (int j = 0; j < array.length; j++) {
            System.out.println(array[i] + array[j]);
        }
    }
}
```

အပေါ်က ဥပမာမှာ ပထမ Loop က $O(N)$ ကြာပြီး၊ ဒုတိယ Loop အထပ်က $O(N^2)$ ကြာပါတယ်။
စုစုပေါင်း Time Complexity က $O(N^2+N)$ ဖြစ်ပေမယ့်၊ N ဟာ တဖြည်းဖြည်း အရေးမပါတော့တဲ့ အတွက် ပယ်ဖျက်လိုက်ပြီး $O(N^2)$ လို့ပဲ သတ်မှတ်ပါတယ်။

၃။ ပေါင်းခြင်း နှင့် မြှောက်ခြင်း

အဆင့် A လုပ်ပြီးမှ အဆင့် B ကို ဆက်လုပ်တယ်။ အဲဒီလို case ဆိုရင် ပေါင်းပါတယ်။ $O(A+B)$

```
for (int i = 0; i < arrayA.length; i++) { // O(A)
    System.out.println(arrayA[i]);
}
for (int j = 0; j < arrayB.length; j++) { // O(B)
    System.out.println(arrayB[j]);
}
// Time Complexity: O(A + B)
```

အဆင့် A တစ်ခါလုပ်တိုင်း အဆင့် B ကို ထပ်ခါ ထပ်ခါ လုပ်ရတယ်။ ဥပမာ Nested Loop လိုမျိုး ဆိုရင် မြှောက်ပါတယ်။ $O(A \times B)$

```
for (int i = 0; i < arrayA.length; i++) { // O(A)
    for (int j = 0; j < arrayB.length; j++) { // O(B)
        System.out.println(arrayA[i] + arrayB[j]);
    }
}
// Time Complexity: O(A * B)
```

Time Complexity

Time Complexity ဆိုတာ အချက်အလက် ပမာဏ N အပေါ်မှာ မူတည်ပြီး Algorithm အလုပ်လုပ်ရသည့် အကြိမ်အရေ အတွက် ဘယ်လောက် များလဲဆိုတာ တိုင်းတာခြင်း ဖြစ်ပါတယ်။

$O(1)$ Constant Time

အချက်အလက် ဘယ်လောက်ပဲများများ အလုပ်လုပ်ရသည့် အချိန် အတူတူပါပဲ။

```
public void printFirstElement(int[] array) {

    // Array ထဲမှာ ဂဏန်း ၁၀ လုံးပဲရှိရှိ၊ သန်း ၁၀၀ ပဲရှိရှိ
    // ပထမဆုံး ဂဏန်းကို ယူဖို့ ကြာချိန်က အတူတူပါပဲ။
    System.out.println(array[0]); // O(1)
}
```

$O(N)$ Linear Time

အချက်အလက်ပမာဏ များလာတာနဲ့အမျှ ကြာချိန်ကလည်း တိုက်ရိုက် အချိုးကျ များလာပါတယ်။ Array က ၁၀ ဆ ကြီးလာရင် ကြာချိန် ၁၀ ဆ ပိုကြာပါမယ်။

```
public void printAllElements(int[] array) {
    // Loop က Array ရဲ့ အရွယ်အစား (N) အကြိမ် အရေအတွက်အတိုင်း အလုပ်လုပ်ပါတယ်။
    for (int i = 0; i < array.length; i++) { // O(N)
        System.out.println(array[i]);
    }
}
```

$O(N^2)$ Quadratic Time

အချက်အလက် ပမာဏ များလာသည်နှင့် အမျှ ကြာချိန် က နှစ်ထပ်ကိန်း (Square) အချိုးနဲ့ များလာပါတယ်။ များသော အားဖြင့် Loop နှစ်ထပ် (Nested Loop) တွေ မှာ တွေ့ရပါတယ်။

```
public void printAllPairs(int[] array) {
```

```
        // အပြင် Loop က N ကြိမ် အလုပ်လုပ်တယ်
        for (int i = 0; i < array.length; i++) {
            // အတွင်း Loop ကလည်း အပြင် Loop တစ်ခါပတ်တိုင်း N ကြိမ် ထပ်လုပ်တယ်
            for (int j = 0; j < array.length; j++) {
                System.out.println(array[i] + ", " + array[j]);
            }
        }
    }
    // Time complexity: O(N * N) = O(N^2)
```

$O(\log N)$ Logarithmic Time

အလွန်မြန်ဆန်သည့် Algorithm လို့ ဆိုလို့ရပါတယ်။ အချက်အလက်တွေ များလာပေမယ့် ကြာချိန် က အနည်းငယ်ပဲ တိုးလာပါတယ်။ အဆင့် တစ်ဆင့် လုပ်တိုင်း ကျန်နေသည့် အချက်အလက် တစ်ဝက်ကို ပယ်ဖျက်သွားနိုင်သည့် လုပ်ငန်းစဥ်တွေ ဖြစ်ပါတယ်။ ဥပမာ Binary Search လိုမျိုးပေါ့။

> **Algorithm မှာ Log ဆိုတာ**
>
> ပုံမှန် သင်္ချာမှာ log ဆိုရင် Base 10 သို့မဟုတ် Base e ကို ယူဆပါတယ်။ Computer Science မှာတော့ $\log N$ လို့ ရေးထားခဲ့ရင် Base 2 ($\log_2 N$)ဖြစ်ပါတယ်။

$\log_2 N$ ဆိုတာ ဘာလဲ ?

$\log_2 N$ ဆိုတာ 1 ရောက်အောင် 2 နဲ့ ဘယ်နှကြိမ်စားနိုင်လဲဆိုတာကို ဖော်ပြတာပါ။

N = 8 ဆိုပါစို့ ။ 8 ကို တစ်ဝက်စီ ပိုင်း ခဲ့ရင် ၃ ခါ ပိုင်းရပါတယ်။ 8 → 4 → 2 → 1 ။ တနည်းပြောရင် 8 = 2^3 ပါ။ အဲဒီ အစိပ္ပာယ်က $\log_2(8) = 3$ နဲ့ အတူတူပါပဲ။

ဥပမာ Array ထဲမှာ အခန်း အရေအတွက် 1024 ရှိတယ် ဆိုပါစို့။ N = 1024 ပါ။ Algorithm ရဲ့ Big O က $O(\log_2 N)$ ဆိုပါစို့။

$\log_2(1024) = 10$ ရပါတယ်။ ဒါကြောင့် ဒီ algorithm က အရမ်းမြန်တယ်။ 1024 အခန်းတောင် 10 ကြိမ်ပဲ အလုပ်လုပ်ရပါတယ်။ Time Complexity ကောင်းတယ်လို့ ဆိုရပါမယ်။

```
    public int binarySearch(int[] sortedArray, int target) {
        int left = 0;
        int right = sortedArray.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (sortedArray[mid] == target) {
                return mid;
            }

            // အဆင့်တစ်ဆင့်တိုင်းမှာ ရှာရမယ့်အပိုင်းကို တစ်ဝက်စီ လျှော့ချပစ်ပါတယ်။
            if (sortedArray[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
```

```
    }

    return -1;
}

// Time complexity: O(log N)
```

$O(N \log N)$ Linearithmic Time

$O(N)$ ထက် အနည်းငယ် ပိုကြာသော်လည်း $O(N^2)$ ထက် အများကြီး ပိုမြန်ပါတယ်။ အချက်အလက်တွေကို တစ်ဝက်စီ ပိုင်းခြားပြီးမှ ပြန်လည်ပေါင်းစပ်တဲ့ (Divide and Conquer) အယ်ဂိုရီသမ်တွေမှာ အများဆုံး တွေ့ရပါတယ်။ ထိရောက်တဲ့ Sorting Algorithm အများစုရဲ့ Time Complexity ဖြစ်ပါတယ်။ ဥပမာ Merge Sort က အခြေအနေ အားလုံးမှာ $O(N \log N)$ ဖြစ်ပါတယ်။ Quick Sort ကတော့ ပျမ်းမျှ (average) $O(N \log N)$ ဖြစ်ပေမယ့် worst case မှာ $O(N^2)$ အထိ ဆိုးနိုင်ပါတယ်။

```
// အသေးစိတ်ကို နောက်ပိုင်း လေ့လာရပါမည်။

public void sortArray(int[] array) {
    java.util.Arrays.sort(array);
}

// Time complexity: O(N log N)
```

> Sorting algorithm အများစု၏ average time complexity သည် O(N log N) ဖြစ်သည်။

$O(2^N)$ Exponential Time

အချက်အလက် တစ်ခုတိုးလာတိုင်း ကြာချိန်က ၂ ဆစီ ပွားသွားပါတယ်။ များသောအားဖြင့် Branch တွေ အများကြီးခွဲထွက်သွားတဲ့ Recursive အယ်ဂိုရီသမ်တွေမှာ တွေ့ရပါတယ်။

```
public int fibonacci(int n) {
    if (n <= 1) return n;
    // Function တစ်ခါခေါ်တိုင်း နောက်ထပ် Function ၂ ခုကို ပြန်ခွဲထွက်သွားပါတယ်
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Time complexity: O(2^N)
```

Space Complexity (မှတ်ဉာဏ် အသုံးပြုမှု)

Space Complexity ဆိုတာ Algorithm မှာ အလုပ်လုပ်ဖို့အတွက် Memory ဘယ်လောက် လိုအပ်လဲ ဆိုတာကို တိုင်းတာတာပါ။ Data (Input) ယူထားတဲ့ နေရာကိုတော့ ထည့်မတွက်ပါဘူး။ Data အသစ်ဖန်တီးတာ၊ Array အသစ် ဆောက်တာတွေကိုပဲ ထည့်တွက်ပါတယ်။

$O(1)$ Constant Space

Input Data ဘယ်လောက်များများ၊ memory အသစ်မလိုပါဘူး။

```
public int sumArray(int[] array) {
    int sum = 0; // Integer variable တစ်ခုစာပဲ နေရာယူပါတယ်။ (O(1) space)
    for (int i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}

// Space Complexity: O(1)
```

$O(N)$ Linear Space

Input Data ပမာဏနဲ့ အချိုးကျပြီး Memory လိုအပ်ပါတယ်။

```
public int[] copyArray(int[] array) {

    // မူလ Array ရဲ့ အရွယ်အစား N နဲ့တူညီတဲ့ Array အသစ်တစ်ခုကို တည်ဆောက်ပါတယ်။

    int[] newArray = new int[array.length]; // O(N) space

    for (int i = 0; i < array.length; i++) {
        newArray[i] = array[i];
    }

    return newArray;

}

// Space Complexity: O(N)
```

$O(N^2)$ - Quadratic Space

Input Data ကို မူတည်ပြီး 2D Array (သို့) Matrix တွေ တည်ဆောက်တဲ့အခါ တွေ့ရပါတယ်။

```
public int[][] createMatrix(int n) {

    // N x N အရွယ်အစားရှိတဲ့ Matrix အသစ် တည်ဆောက်တာ ဖြစ်ပါတယ်။

    int[][] matrix = new int[n][n]; // O(N * N) space

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = i + j;
        }
    }

    return matrix;

}
```

```
// Space Complexity: O(N^2)
```

Time Complexity နှင့် Space Complexity တို့၏ Trade-Off

လက်တွေ့မှာ အကောင်းဆုံး နဲ့ memory အနည်းဆုံး နှစ်ခု လုံး ရဖို့ မလွယ်ကူပါဘူး။ တစ်ခုကို လိုချင်လျှင် တစ်ခုကို အလျှော့ပေးရပါမယ်။ Trade Off လုပ်ရတာပေါ့။

- Time Complexity ကောင်းချင်လျှင် Space Complexity အားနည်းပါတယ်။ ဥပမာ Caching လိုမျိုး ပြန်တွက်စရာ မလိုပဲ သိမ်းထားသည့် ကိစ္စတွေမှာ Time Complexity ကောင်းပေမယ့် Space Complexity မကောင်းပါဘူး။
- Space Complexity ကောင်းချင်လျှင် Time Complexity အားနည်းပါတယ်။ တွက်ချက်မှု တွေကို ကြိုတင်သိမ်းထားမှု မရှိပဲ လိုမှ တွက်သလိုမျိုးပေါ့။

ဥပမာ - Array တစ်ခုတည်းမှာ နံပါတ်တွေ ထပ်နေတာ (Duplicates) ရှိမရှိ စစ်ဆေးခြင်း

နည်းလမ်း ၁။ Nested Loops သုံးခြင်း (ကြာချိန် နှေးသွားမည်၊ မှတ်ဉာဏ် သက်သာမည်)

```java
public boolean hasDuplicatesSlow(int[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = i + 1; j < array.length; j++) {
            if (array[i] == array[j]) return true;
        }
    }

    return false;

}

// Time: O(N^2) - ဂဏန်းတစ်လုံးကို ကျန်တဲ့ဂဏန်းတိုင်းနဲ့ လိုက်စစ်ရလို့ ကြာပါတယ်။
// Space: O(1) - Data အသစ်/Array အသစ် ထပ်မဆောက်လို့ မှတ်ဉာဏ် လုံးဝသက်သာပါတယ်။
```

နည်းလမ်း ၂။ HashSet သုံးခြင်း (Hash Table အကြောင်းကို နောက်ပိုင်း အခန်းတွေမှာ အသေးစိတ်တွေ့ရပါမည်)

```java
public boolean hasDuplicatesFast(int[] array) {

    HashSet<Integer> seen = new HashSet<>(); // Memory အသစ် ယူလိုက်ပါပြီ!
    for (int num : array) {
        if (seen.contains(num)) return true; // ရှာရတာ O(1) ဖြစ်လို့ မြန်ပါတယ်
        seen.add(num);
    }
    return false;
}

// Time: O(N) - Array ကို တစ်ခေါက်ပဲ ပတ်စရာလိုတဲ့အတွက် မြန်ပါတယ်။
// Space: O(N) - ဂဏန်းတွေကို HashSet ထဲအသစ်ပြန်ထည့်ရလို့ Memory (N) စာ ပိုယူသွားပါတယ်။
```

Production မှာ $O(N^2)$ ဘာကြောင့် အန္တရာယ်ရှိသလဲ

$O(N^2)$ algorithm ဟာ data နည်းတဲ့ အချိန် (test data) မှာ ပြဿနာ မတက်ပါဘူး။ ဒါပေမယ့် production မှာ data ကြီးလာတဲ့အခါ အလွန် ဆိုးရွားသွားနိုင်ပါတယ်။

- $N = 100$ ဆိုရင် $N^2 = 10,000$ — အဆင်ပြေပါသေးတယ်။
- $N = 10,000$ ဆိုရင် $N^2 = 100,000,000$ — စတင် နှေးလာပါပြီ။
- $N = 1,000,000$ ဆိုရင် $N^2 = 10^{12}$ — server လုံးဝ ရပ်သွား (timeout) နိုင်ပါတယ်။

တကယ့် ဥပမာ — user 1,000 ယောက်ပဲ ရှိစဉ်က ကောင်းနေတဲ့ feature တစ်ခုဟာ user သိန်းနဲ့ချီ များလာတဲ့အခါ ရုတ်တရက် နှေးကွေးသွားတတ်ပါတယ်။ ဒါကြောင့် data ကြီးနိုင်တဲ့ နေရာတွေမှာ $O(N^2)$ ကို ရှောင်သင့်ပါတယ်။

Real-World ဥပမာများ

Time Complexity ဟာ စာသင်ခန်း သီအိုရီ မဟုတ်ပါဘူး။ နေ့စဉ် development မှာ ဒီလို တွေ့ရပါတယ်။

- **Database records:** record သန်းနဲ့ချီ ထဲက item တစ်ခုကို index မပါဘဲ ရှာရင် $O(N)$ ဖြစ်ပြီး နှေးပါတယ်။ index (B-Tree) ပါရင် $O(\log N)$ နဲ့ မြန်ပါတယ်။
- **API response list:** API ကပြန်လာတဲ့ list ထဲမှာ item တစ်ခုစီကို loop နှစ်ထပ်နဲ့ နှိုင်းယှဉ်ရင် $O(N^2)$ ဖြစ်ပြီး response နှေးသွားပါတယ်။
- **Transaction list:** ဘဏ် transaction list ထဲက duplicate ရှာတာ၊ စုစုပေါင်း ပေါင်းတာတွေဟာ list အရွယ်အစား N ပေါ် တိုက်ရိုက် မူတည်ပါတယ်။

Solution နှစ်ခုကို ဘယ်လို နှိုင်းယှဉ်မလဲ

Algorithm နှစ်ခု ရှိရင် ဘယ်ဟာ ပိုကောင်းလဲ ဆုံးဖြတ်ဖို့ အောက်ပါ အဆင့်တွေနဲ့ နှိုင်းယှဉ်ပါ။

1. **Time Complexity နှိုင်းယှဉ်ပါ။** ဥပမာ $O(N)$ က $O(N^2)$ ထက် ပိုကောင်းပါတယ်။
2. **Space Complexity ကို ကြည့်ပါ။** Time တူရင် memory နည်းတဲ့ဟာ ပိုကောင်းပါတယ်။
3. **Data အရွယ်အစား (N) ကို စဉ်းစားပါ။** N သေးရင် ရိုးရှင်းတဲ့ $O(N^2)$ က လုံလောက်ပါတယ်။ N ကြီးရင် $O(N \log N)$ သို့ $O(N)$ လိုပါတယ်။
4. **Trade-off ကို ဆုံးဖြတ်ပါ။** အပေါ်က duplicate ဥပမာမှာ Nested Loop က $O(N^2)$ time / $O(1)$ space ၊ HashSet က $O(N)$ time / $O(N)$ space ။ Data ကြီးရင် HashSet ($O(N)$ time) ကို ရွေးတာ ပိုသင့်တော်ပါတယ်။

> **အကျဉ်းချုပ်:** "ပိုကောင်းတဲ့ solution" ဆိုတာ အမြဲ Big O နည်းတာ မဟုတ်ပါဘူး။ Data အရွယ်အစား နဲ့ memory ကန့်သတ်ချက် အပေါ်မူတည်ပြီး ဆုံးဖြတ်ရပါတယ်။

အခု ဆိုရင် အနည်းငယ် သဘောပေါက်ပြီလို့ ထင်ပါမယ်။ နောက်ပိုင်း အခန်းတွေ မှာ algorithm တိုင်း ကို Time Complexity နဲ့ Space Complexity ကို ဖော်ပြပေးသွားပါမယ်။ တချို့ sorting တွေမှာ တော့ အသေးစိတ် ပြန်လည် ရှင်းပြပါမယ်။ ထို့မှသာ နားလည် လွယ်ပါလိမ့်မယ်။

အခန်း ၃ - Random Access Memory, Array, String

Algorithm တွေဟာ data structure ပေါ်မှာ အခြေခံပြီး အလုပ်လုပ်ပါတယ်။ Data Structure တွေဟာ Random Access Memory (RAM) ပေါ်မှာ နေရာယူထားပါတယ်။ ဒါကြောင့် RAM သဘောတရား အခြေခံ ကို အနည်းငယ် နားလည် ထားဖို့ လိုပါတယ်။

RAM Architecture

RAM ကို ကြီးမားသည့် စာတိုက်ပုံး (P.O. BOX) လို့ မြင်ယောင်ကြည့်ပါ။ နောက်ပိုင်း မြန်မာနိုင်ငံက condo တွေမှာ ရှိသလို စာတိုက်ပုံး အကွက်လေး တွေ အများကြီး ရှိပါတယ်။ အကွက် တစ်ကွက် စီ က သူ့ရဲ့ သီးသန့် အိမ်လိပ်စာ အတွက် ပါပဲ။ Computer မှာလည်း Data ကို သိမ်းချင်သည့် အခါမှာ RAM ပေါ်မှာ သွားသိမ်းဖို့ လိုပါတယ်။ တနည်းပြောရင် အခန်းလွတ်နေသည့် address တစ်ခု မှာ သွားသိမ်းရ တာပေါ့။ အခန်း တစ်ခန်း စီ အတွက် Address တစ်ခု ဆီ ထားရှိသည့် သဘောပါ။

> ဥပမာ သင့်သူငယ်ချင်း ဆီကို စာပို့ချင် သူငယ်ချင်းနာမည် (variable name) သိရုံ နဲ့ မရပါဘူး။ သူ့ အိမ်လိပ်စာ (memory address) သိမှ ပို့လို့ရမှာပါ (memory ပေါ်မှာ တန်ဖိုးသိမ်း)။ Computer မှာ variable တစ်ခု ကြေငြာလိုက်သည် နှင့် RAM မှာ variable နဲ့ memory address ကို ချိတ် ပေးလိုက်သည့် သဘောပါ။

ဥပမာ - `Address 100` မှာ `တန်ဖိုး 50` ကို သိမ်းမယ် ဆိုပါစို့။ ပုံမှန်အားဖြင့် အောက်ပါဇယားအတိုင်း မြင် ယောင်နိုင်ပါတယ်။

Memory Address (လိပ်စာ)	Value (သိမ်းဆည်းထားသော တန်ဖိုး)	Description (ရှင်းလင်းချက်)
`0x0099`	0	အလွတ်
`0x0100`	50	Integer တန်ဖိုး 50 သိမ်းထားသည့်နေရာ
`0x0101`	'A'	Character 'A' သိမ်းထားသည့်နေရာ
`0x0102`	0	အလွတ်

> *(မှတ်ချက် - Memory Address များကို များသောအားဖြင့် Hexadecimal (16 လီစနစ်) ဥပမာ* `0x0100` *ဖြင့် ပြသလေ့ရှိပါတယ်။)*

Random Access ဆိုတာ ဘာလဲ

Random Access ဆိုတာကတော့ လိပ်စာ သိရင် ဘယ်နားမှာ ပဲ ရှိရှိ Access လုပ်လို့ရသည့် သဘောပါ။ ဥပမာ အားဖြင့် ကျွန်တော်တို့ ငယ်ငယ်က ကက်ဆက် ခွေ တွေမှာ သီချင်း နားထောင်ရင် နောက် တစ်ခု ကို ကျော်ချင်သည့် အခါမှာ ရှေ့ပိုင်းကို Fast Forward လုပ်ရပါတယ်။ တနည်းပြောရင် တိတ်ခွေက အရှေ့ ပိုင်းကို လိပ်ပြီး ကျော်သည့် သဘောပေါ့။ ဒါကို Sequential Access လို့ ခေါ်ပါတယ်။ ဒါပေမယ့် Youtube , Spotify နဲ့ computer မှာ သီချင်း နားထောင်ရင် ကြိုက်သည့် သီချင်းကို ကျော်လို့ ရသလို ကြိုက်သည့် အချိန်ကို ကျော်လို့ ရပါတယ်။ တနည်းပြောရင် တိကျသည့် နေရာကို တန်းသွားလို့ရတယ် ပေါ့။ ဒါကို Random Access လို့ ခေါ်ပါတယ်။

RAM မှာလည်း Address 0 ကို သွားတာ နဲ့ Address 1000 ကို သွားတာ ကြာချိန် အတူတူပါပဲ။ တနည်း ပြောရင် Memory Access ရဲ့ Time Complexity က $O(1)$ ဖြစ်ပါတယ်။

Stack နှင့် Heap

Stack နဲ့ Heap က အမြဲတန်း ရောထွေးနေတတ်တယ်။ memory အပိုင်း Stack and Heap နဲ့ Data Structure Stack and Heap က မတူပါဘူး။ တစ်ခါတစ်လေ interview တွေမှာ မင်း Stack နဲ့ Heap ကို သိလားလို့ မေးရင် သေချာအောင် ပြန်မေးရတယ်။ Memory က Stack နဲ့ Heap ကို မေးတာလား။ Data Structure က Stack နဲ့ Heap ကို မေးတာလားပေါ့။ အခုပြောမည့် အကြောင်းကတော့ Memory မှာ Data သိမ်းသည့် Stack နှင့် Heap ပါ။

Java လို Programming ဘာသာစကား အများစုမှာ Program တစ်ခု အလုပ်လုပ်ချိန် (Runtime) မှာ Memory ကို Stack/Heap ဆိုပြီး ရှင်းပြလေ့ရှိပါတယ်။ ဒီ chapter ထဲမှာလည်း နားလည်လွယ်အောင် အဲ့ဒီ model နဲ့ ဆက်ရှင်းပါမယ်။ ဒါပေမယ့် JVM specification က memory layout အတိအကျကို guarantee မပေးပါဘူး။

Stack Memory

Stack ဆိုတာက မြန်မြန် အလုပ်လုပ်နိုင်ပြီး စနစ်တကျ စီစဥ်ထားသည့် သေးငယ် သည့် memory နေရာ ပါ။

Stack Memory က ဘယ်လို အလုပ်လုပ်သလဲ ဆိုတော့ **LIFO ,** Last In , First Out (နောက်ဆုံး ဝင်တာ က အရင်ဆုံးပြန်ထွက်သည်) စနစ် နဲ့ အလုပ်လုပ်ပါတယ်။ စာအုပ် အဆင့်ဆင့် ထပ်ထားသလိုပေါ့။ အသစ်ထပ်တင်ရင် အပေါ်ဆုံးမှာ တင်ရတယ်။ ပြန်ယူသည့် အခါမှာလည်း အပေါ်ဆုံးက စာအုပ်ကို အ ရင် ယူရတယ်။

Function (Method) တစ်ခုခု ကို ခေါ်လိုက်သည့် အခါ အဲ့ဒီ Function အတွက် temporary နေရာ ဖန်တီး ပေးရတယ်လို့ စဉ်းစားနိုင်ပါတယ်။ Conceptually ဒီနေရာထဲမှာ local variable ဥပမာ `int x = 10;` , `double y = 5.5` စသည်တို့နဲ့ object ကို ညွှန်ပြတဲ့ reference value တွေ (`int[] arr` , `String s`) ကို သိမ်းထားတယ်လို့ နားလည်နိုင်ပါတယ်။

Function တစ်ခု အလုပ်လုပ် ပြီးသွားတာ သို့မဟုတ် Return ပြန်လိုက်သည့် အခါမှာ Stack Frame တစ် ခုလုံး Memory ပေါ်က အလိုအလျောက် ချက်ချင်း ဖျက်ပစ်လိုက်ပါတယ်။

Stack Memory က နေရာ သေးငယ်တယ်။ သိမ်းထားတာတွေကလည်း Primitive Data နဲ့ reference value တွေပဲ ဖြစ်သည့် အတွက်ကြောင့် အများကြီး မလိုဘူး။ Base case မရှိတဲ့ recursion တွေလိုမျိုး ခေါ်မိသည့် အခါမှာ `StackOverflowError` ဆိုသည့် ပြဿနာ တက်တတ်ပါတယ်။ တနည်းပြောရင် function တွေကို ဆင့်ကာ ဆင့် ကာ ခေါ်ယူပြီး ပြည့်သွားတာမျိုးပေါ့။

Heap

Heap ဆိုတာ runtime မှာ dynamically allocate လုပ်ထားတဲ့ objects တွေကို သိမ်းထားတယ်လို့ ရှင်းပြလေ့ရှိတဲ့ memory area ဖြစ်ပါတယ်။

Stack လိုမျိုး အစဥ်လိုက် စီနေဖို့ မလိုပါဘူး။ အလွတ်ရှိတဲ့ memory နေရာတွေမှာ allocate လုပ်ပြီး သိမ်းနိုင်ပါတယ်။

Java ကို conceptually ရှင်းမယ်ဆိုရင် Object တွေကို Heap ပေါ်မှာ သိမ်းတယ်လို့ သဘောထားနိုင်ပါတယ်။

ဥပမာ -

```
String name = "Mg Mg";
Scanner input = new Scanner(System.in);
int[] arr = new int[100];
```

Custom Class Object များ စသည်တို့ ဖြစ်ပါတယ်။

ဒီနေရာမှာ ရိုးရိုး သင်ကြားရေး model အရ local reference variables တွေက Stack ဘက်မှာ ရှိပြီး Object တွေက Heap ဘက်မှာ ရှိတယ်လို့ မြင်နိုင်ပါတယ်။

ဥပမာ -

```
String name = "Mg Mg";
```

ဒီ statement ကို run လိုက်တဲ့အခါ memory ထဲမှာ အောက်ပါအတိုင်း ဖြစ်ပါတယ်။

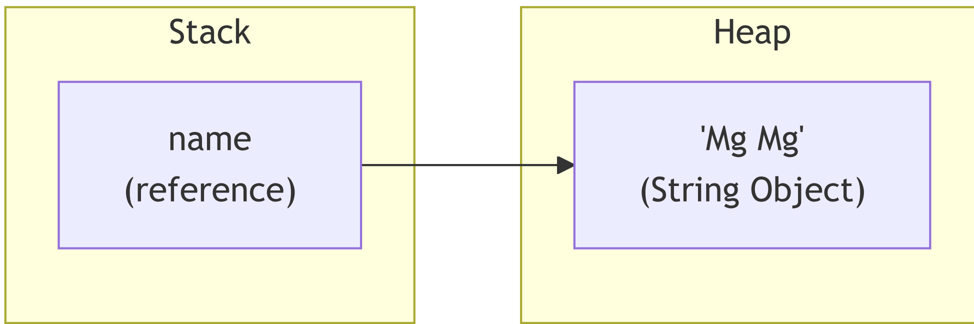
1. Stack

`name` ဆိုတဲ့ variable ကို **reference variable** အနေနဲ့ Stack side မှာ ရှိတယ်လို့ conceptual model နဲ့ မြင်နိုင်ပါတယ်။
ဒီ variable ထဲမှာ Object ကို မသိမ်းပါဘူး။ Object ကို ညွှန်တဲ့ **reference value** ကိုသာ သိမ်းထားပါတယ်။ Java က ဒီ reference ကို raw memory address အဖြစ် တိုက်ရိုက် မပြပေးပါဘူး။

2. Heap

`"Mg Mg"` ဆိုတဲ့ String Object ကတော့ Heap side မှာ ရှိတယ်လို့ နားလည်နိုင်ပါတယ်။
Stack ထဲက `name` variable က Heap ပေါ်က ဒီ Object ကို လှမ်းညွှန် (reference) လုပ်ထားတာ ဖြစ်ပါတယ်။

Conceptually memory ကို အောက်လိုမြင်နိုင်ပါတယ်။



အကယ်၍ reference variable မရှိတော့တဲ့အခါ (ဥပမာ function ပြီးသွားတဲ့အခါ) Stack ပေါ်က variable က ပျက်သွားနိုင်ပါတယ်။ ဒါပေမယ့် Heap ပေါ်က Object ကတော့ ချက်ချင်း မဖျက်ပါဘူး။

Java မှာတော့ အသုံးမပြုတော့တဲ့ Object တွေကို Garbage Collector ဆိုတဲ့ အလိုအလျောက် စနစ်က နောက်ကွယ်ကနေ ရှင်းလင်းပေးပါတယ်။

C/C++ တို့မှာဆိုရင်တော့ programmer က ကိုယ်တိုင် free() သို့မဟုတ် delete ကို ခေါ်ပြီး memory ကို ဖျက်ပေးရပါတယ်။

ဘာကြောင့် Object ကို function ပြီးတာနဲ့ အလိုအလျောက် မဖျက်တာလဲ ဆိုတော့ Heap ပေါ်က Object တစ်ခုကို နေရာအများကြီးကနေ တစ်ပြိုင်တည်း reference လုပ်ထားနိုင်လို့ ဖြစ်ပါတယ်။

Function တစ်ခု ပြီးသွားတာနဲ့ Object ကို ဖျက်လိုက်မယ်ဆိုရင် ကျန်နေတဲ့ အခြား reference တွေက အသုံးပြုတဲ့အခါ crash ဖြစ်နိုင်ပါတယ်။

ဒါကြောင့် Object ကို ဘယ်သူမှ reference မလုပ်တော့တဲ့အခါ (Unreferenced ဖြစ်တဲ့အခါ) မှသာ ဖျက်ပေးရပါတယ်။

Java မှာတော့ Garbage Collector က ဒီအလုပ်ကို အလိုအလျောက် လုပ်ပေးပါတယ်။

Stack နှင့် Heap အတူတကွ လုပ်ဆောင်ခြင်း

ကျွန်တော်တို့ အပေါ်မှာ ပြထားသည့် String ဥပမာ ကို ပြန်ကြည့်ရအောင်။

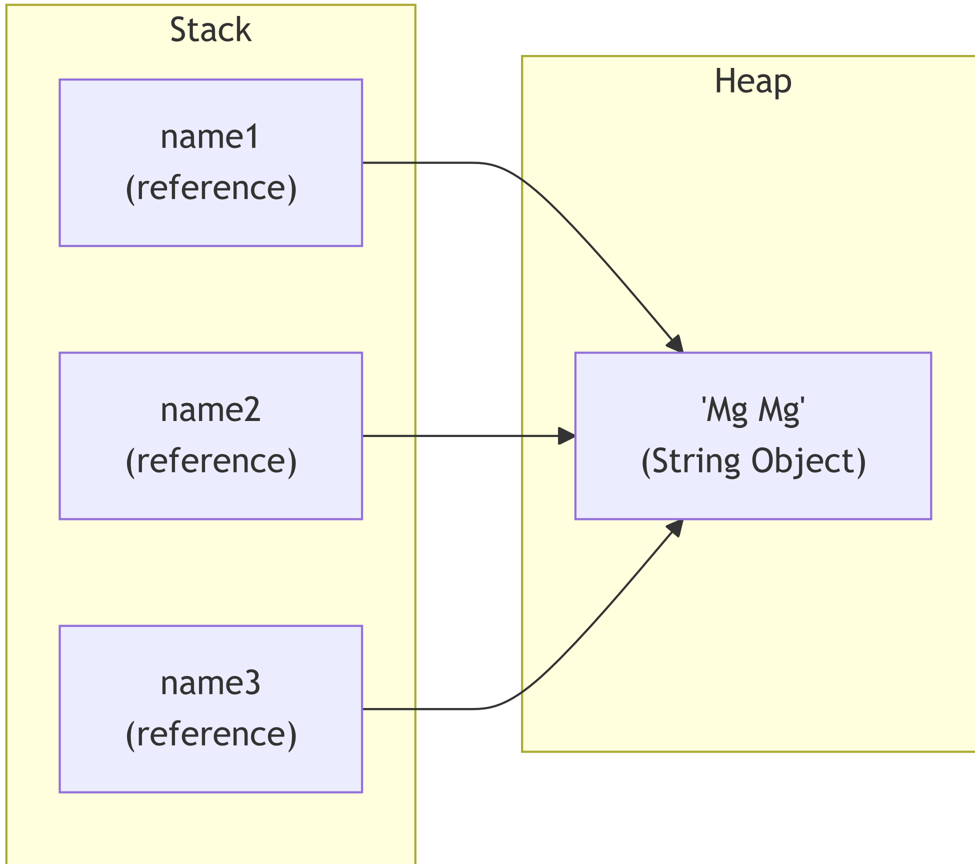
```
String name1 = "Mg Mg";
String name2 = "Mg Mg";
String name3 = "Mg Mg";
```

ဒီ code ကို run လိုက်တဲ့အခါ Java က memory ကို optimize လုပ်တဲ့အတွက် `"Mg Mg"` ဆိုတဲ့ String literal Object ကို **တစ်ခါပဲ ဖန်တီးပါတယ်**။ ဒါကို **String Pool** လို့ခေါ်ပါတယ်။ *(မှတ်ချက် - String Pool သည် Heap တစ်ခုလုံးကို ဆိုလိုခြင်းမဟုတ်ဘဲ Heap အတွင်းရှိ သီးခြား subset တစ်ခုသာ ဖြစ်ပါသည်။)*

Stack ပေါ်မှာတော့ `name1` , `name2` , `name3` ဆိုတဲ့ variables တွေ ရှိပါတယ်။ ဒီ variables တွေက **reference variables** ဖြစ်ပြီး Object ကို မသိမ်းပါဘူး။ Object ကို ညွှန်တဲ့ **reference value** ကိုသာ သိမ်းထားပါတယ်။

Heap ထဲမှာတော့ `"Mg Mg"` ဆိုတဲ့ **String Object တစ်ခုတည်းပဲ** ရှိပါတယ်။

Conceptually memory ကို အောက်လိုမြင်နိုင်ပါတယ်။



ဆိုလိုတာက `"Mg Mg"` ဆိုတဲ့ Object တစ်ခုတည်းကို **reference variable အများကြီးကနေ share လုပ်ပြီး အသုံးပြုနိုင်ပါတယ်**။

ဒီလို design လုပ်ထားတာက memory usage ကို လျှော့ချစေပြီး performance ကိုလည်း ပိုကောင်းစေပါတယ်။

အကယ်၍ Stack ပေါ်က variable တစ်ခုခု ပျက်သွားတယ်ဆိုရင် (ဥပမာ function ပြီးသွားတဲ့အခါ) ကျန်နေတဲ့ reference တွေ ရှိနေသေးတဲ့အတွက် Heap ပေါ်က Object ကို ချက်ချင်း မဖျက်ပါဘူး။

Java မှာတော့ **GC roots** တွေကနေ မရောက်နိုင်တော့တဲ့ Object (**unreachable object**) ဖြစ်သွားတဲ့အခါမှာ GC က ရှင်းလင်းနိုင်တဲ့ candidate ဖြစ်လာပါတယ်။

ဒါပေမယ့် unreachable ဖြစ်သွားတာနဲ့ ချက်ချင်း ဖျက်မယ်လို့ မဆိုလိုပါဘူး။ Garbage Collector က ဘယ်အချိန် run မလဲဆိုတာကို collector ရဲ့ heuristics နဲ့ runtime အခြေအနေပေါ် မူတည်ပြီး ဆုံးဖြတ်တာ ဖြစ်ပါတယ်။

တနည်းဆိုလျှင် Heap နဲ့ Stack ကလည်း အတူ တကွ တွဲပြီး အလုပ်လုပ်တာကို မြင်နိုင်ပါတယ်။

Function Argument Passing: String vs Object

Stack နဲ့ Heap ကို နားလည်ပြီးရင် Function ထဲကို variable တွေ ပို့လိုက်တဲ့အခါ ဘာကြောင့် တချို့ case တွေမှာ value မပြောင်းသလို မြင်ရပြီး တချို့ case တွေမှာ ပြောင်းသွားသလဲ ဆိုတာကို ဆက်ကြည့်လို့ ရပါတယ်။

Java မှာ **pass-by-value** ပဲ ရှိပါတယ်။ ဒါက အရေးကြီးဆုံး rule ပါ။

- Primitive type (`int` , `double` , `boolean`) တွေဆိုရင် value ကို တိုက်ရိုက် copy လုပ်ပြီး ပို့ပါတယ်။
- Object type (`String` , `User` , `Scanner` , `int[]`) တွေဆိုရင် reference value ကို copy လုပ်ပြီး ပို့ပါတယ်။

အဓိက မှတ်ထားရမယ့် အချက်က **Java က original variable ကို မပို့ပါဘူး။ copy တစ်ခုကိုပဲ ပို့တာ** ဖြစ်ပါတယ်။

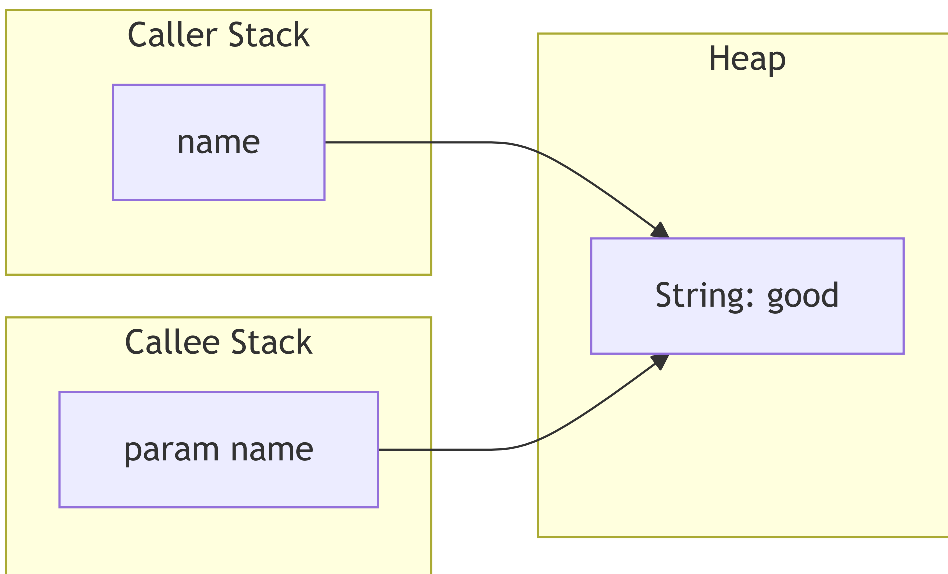
String ကို Function ထဲပို့တဲ့အခါ

ဥပမာ -

```
void updateName(String name) {
    name = "HELLO";
}

String name = "good";
updateName(name);
System.out.println(name);
```

Output က `good` ပဲ ဖြစ်ပါတယ်။

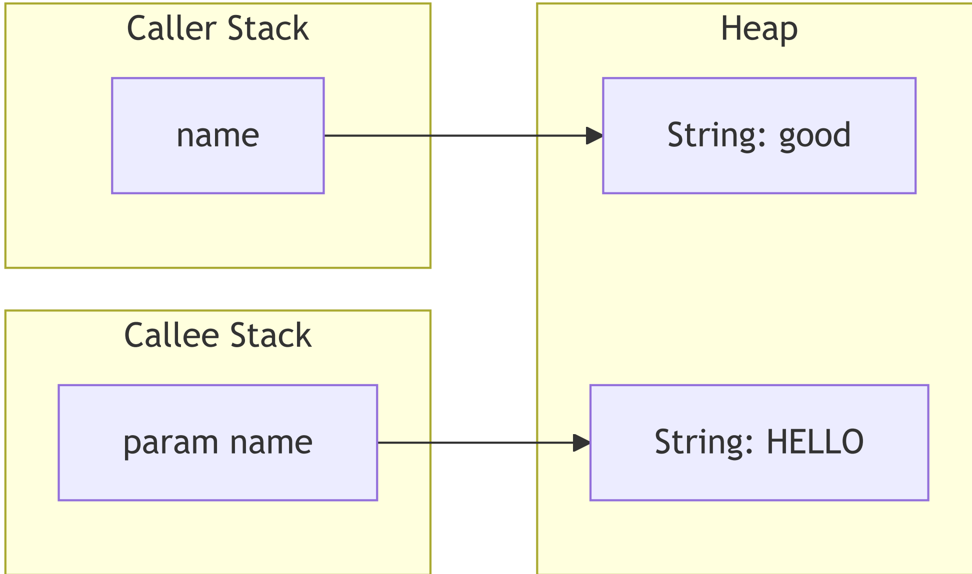
ဘာကြောင့်လဲဆိုတော့ Function ထဲက `name` ဟာ caller ထဲက `name` မဟုတ်ဘဲ **reference copy အသစ်** ဖြစ်ပါတယ်။ အစပိုင်းမှာ နှစ်ခုလုံးက `"good"` ကိုပဲ ညွှန်နေပါတယ်။



Function ထဲမှာ

```
name = "HELLO";
```

လို့ ရေးလိုက်တဲ့အခါ `"good"` String object ကို မပြင်ပါဘူး။ Function ထဲက local reference copy ကို `"HELLO"` ဆိုတဲ့ String object အသစ်ကို ညွှန်အောင် ပြောင်းလိုက်တာပါ။



Function ပြီးသွားတာနဲ့ Callee Stack ပေါ်က `param name` ပျက်သွားပြီး Caller ထဲက `name` ကတော့ `"good"` ကိုပဲ ဆက်ညွှန်နေပါတယ်။ ဒါကြောင့် output က `good` ဖြစ်တာပါ။

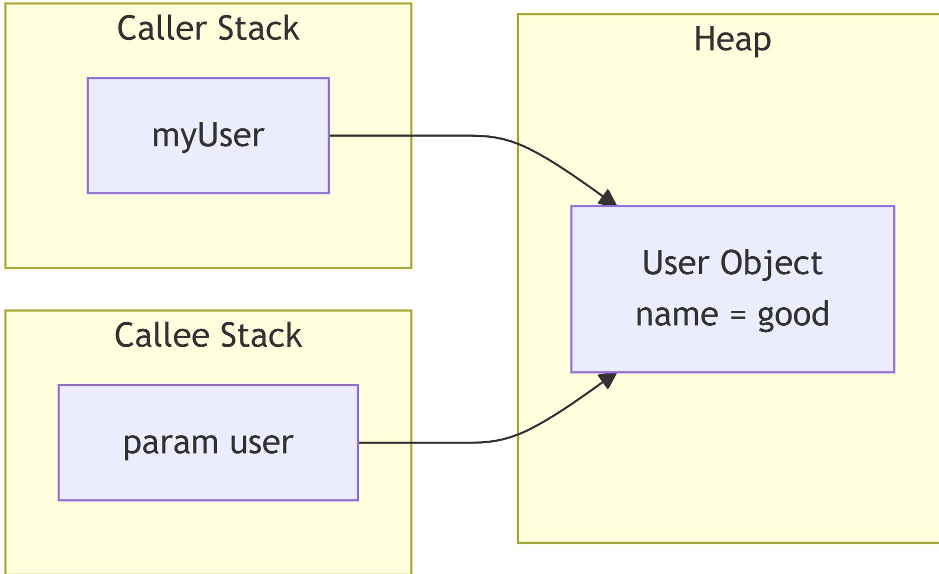
Object ကို Function ထဲပို့တဲ့အခါ

ဥပမာ -

```
class User {
    String name;
}

void update(User user) {
    user.name = "HELLO";
}

User myUser = new User();
myUser.name = "good";

update(myUser);
System.out.println(myUser.name);
```

ဒီ code ရဲ့ output က `HELLO` ဖြစ်ပါတယ်။

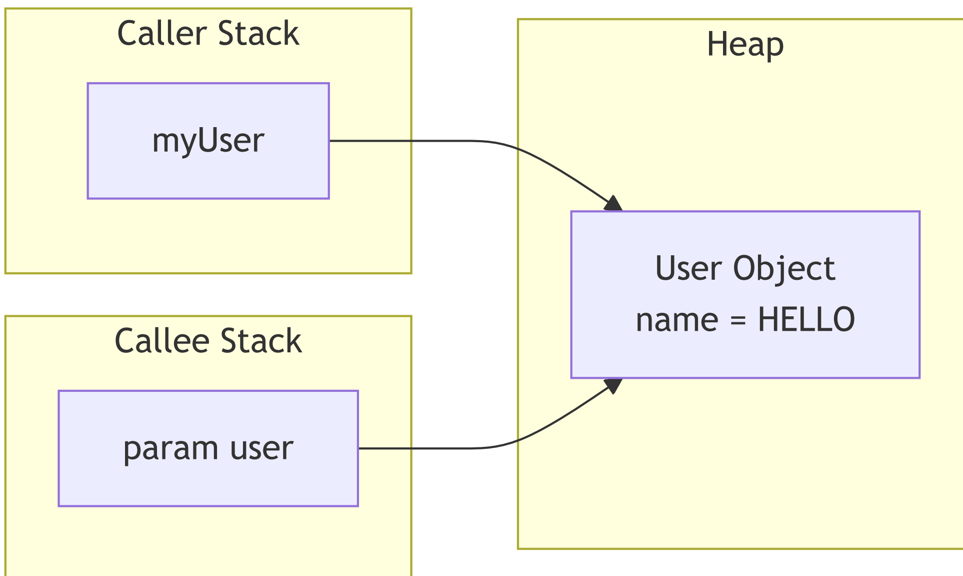
ဘာကြောင့်လဲဆိုတော့ Function ထဲကို ဝင်လာတဲ့ `user` ဟာ **reference copy** ဖြစ်ပေမယ့် Caller ထဲက `myUser` နဲ့ Heap ပေါ်က **object တစ်ခုတည်းကိုပဲ** ညွှန်နေပါတယ်။



Function ထဲမှာ

```
user.name = "HELLO";
```

လို့ ရေးလိုက်တဲ့အခါ reference ကို မပြောင်းဘဲ Heap ပေါ်က User object ထဲက `name` field ကို တိုက်ရိုက်ပြောင်းလိုက်တာ ဖြစ်ပါတယ်။



ဒါကြောင့် Function ပြီးသွားပြီးနောက် Caller ထဲက `myUser.name` ကို print လုပ်ရင် `HELLO` ထွက်လာတာပါ။

Mutable နှင့် Immutable

ဒီနေရာမှာ လူအများစု ရှုပ်သွားတာက **String က Object မဟုတ်ဘူးလား** ဆိုတာပါ။ အဖြေက **String ကလည်း Object ပဲ** ဖြစ်ပါတယ်။ ဒါပေမယ့် String နဲ့ User object က behavior မတူတာက **mutable / immutable** ကွာလို့ ဖြစ်ပါတယ်။

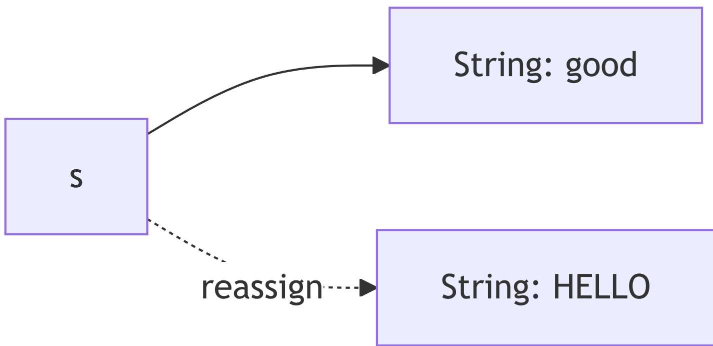
Immutable ဆိုတာ ဘာလဲ

Immutable object ဆိုတာ create လုပ်ပြီးနောက် internal value ကို မပြင်နိုင်တဲ့ object ဖြစ်ပါတယ်။ Value ပြောင်းချင်ရင် object အသစ်ဖန်တီးရပါတယ်။

Java မှာ `String` က immutable ဖြစ်ပါတယ်။

```
String s = "good";
s = "HELLO";
```

ဒီ code မှာ `"good"` String object ကို မပြင်ပါဘူး။ `"HELLO"` String object အသစ်ကို ယူညွှန်လိုက်တာ ဖြစ်ပါတယ်။



တကယ့် memory semantics အရ ပြောရရင် variable `s` က အရင် `"good"` ကို ညွှန်နေရာမှ `"HELLO"` ကို ပြန်ညွှန်သွားတာ ဖြစ်ပါတယ်။ `"good"` object ကို ကိုယ်တိုင် ပြင်လိုက်တာ မဟုတ်ပါဘူး။

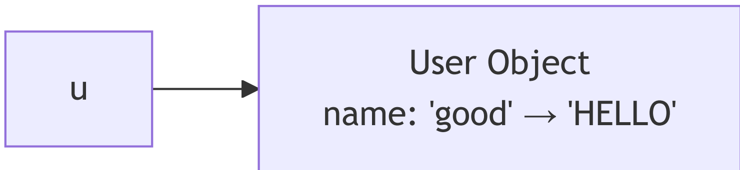
Mutable ဆိုတာ ဘာလဲ

Mutable object ဆိုတာ create လုပ်ပြီးနောက် internal state ကို ပြင်နိုင်တဲ့ object ဖြစ်ပါတယ်။

ဥပမာ custom class တစ်ခု -

```
class User {
    String name;
}

User u = new User();
u.name = "good";
u.name = "HELLO";
```

ဒီနေရာမှာ User object အသစ် မဖန်တီးပါဘူး။ Heap ပေါ်က object တစ်ခုတည်းထဲက field value ကို update လုပ်တာပါ။



Reassign လုပ်တာနှင့် Object ကိုပြင်တာ မတူဘူး

ဥပမာ ဒီ code ကို ကြည့်ပါ -

```
void update(User user) {
    user = new User();
    user.name = "HELLO";
}
```

ဒီ code မှာ output မပြောင်းပါဘူး။ ဘာကြောင့်လဲဆိုတော့ `user = new User();` က Function ထဲက local reference copy ကို object အသစ်ဆီ ပြောင်းလိုက်တာပဲ ဖြစ်ပါတယ်။ Caller ထဲက `myUser` reference ကို မထိခိုက်ပါဘူး။

ဒါကြောင့် Java Function parameter တွေကို ရှင်းလင်းစွာ နားလည်ဖို့အတွက် အောက်ပါ rule နှစ်ခုကို မှတ်ထားရင် လုံလောက်ပါတယ်။

1. Java က **pass-by-value** ပဲ ရှိတယ်။
2. **reference ကိုပြောင်းတာ** နဲ့ **reference ညွှန်ထားတဲ့ object ကိုပြောင်းတာ** မတူဘူး။

အခြားသော Programming Language များရှိ Memory Management

Stack နဲ့ Heap ကို **Java တစ်ခုတည်းကပဲ သုံးတာ မဟုတ်ပါဘူး**။ Programming Language အများစုဟာ ဒီ Memory ခွဲဝေမှုစနစ် (Stack & Heap) ကိုပဲ အခြေခံပြီး အလုပ်လုပ်ကြပါတယ်။ ဒါပေမယ့် နောက်ကွယ်ကနေ Memory ကို ဘယ်သူက ဘယ်လို ရှင်းလင်းပေးလဲ (Memory Management Strategy) ဆိုတာပေါ် မူတည်ပြီး ကွာခြားသွားပါတယ်-

1. **Tracing GC ကို အဓိက အသုံးပြုသော Runtime များ (Java, C#, JavaScript):**

- ဒီ runtime တွေမှာ Developer က `free()` / `delete` လို memory free code ကို ကိုယ်တိုင် မရေးရပါဘူး။
- အသုံးမပြုတော့တဲ့ object တွေကို runtime ရဲ့ **Garbage Collector (GC)** က နောက်ကွယ်ကနေ ရှင်းလင်းပေးပါတယ်။

1. **TypeScript:**

- TypeScript က JavaScript ပေါ်မှာ syntax/type system တင်ထားတာဖြစ်ပြီး ကိုယ်ပိုင် runtime သို့မဟုတ် GC မရှိပါဘူး။
- ဒါကြောင့် TypeScript program တွေရဲ့ memory behavior က JavaScript runtime (ဥပမာ V8) ပေါ် မူတည်ပါတယ်။

1. **Reference Counting + GC ပေါင်းစပ်အသုံးပြုသော Runtime များ (Python, PHP):**

- Python နဲ့ PHP တို့မှာ object lifetime ကို reference counting နဲ့အဓိက စောင့်ကြည့်ပြီး cycle ဖြစ်နေတဲ့ object တွေအတွက် GC ကို ထပ်မံသုံးပါတယ်။

- အဲ့ဒါကြောင့် Java/Go လို tracing GC-only model နဲ့ တစ်ပုံစံတည်း မဟုတ်ပါဘူး။

1. **GC + Escape Analysis (Go):**

- Go (Golang) ဟာ GC ကို သုံးတဲ့ ဘာသာစကားဖြစ်ပေမယ့် **Escape Analysis** ကိုလည်း ပေါင်းစပ်အသုံးပြုပါတယ်။
- Compile လုပ်တဲ့အချိန်မှာ "ဒီ value က function scope အပြင်ကို ထွက်သွားမလား" ဆိုတာကို ဆုံးဖြတ်ပါတယ်။
- Escape မလုပ်တဲ့ value တချို့ကို GC ကို မပေးဘဲ stack-like storage ပေါ်မှာပဲ ထားနိုင်တာကြောင့် performance ပိုကောင်းစေပါတယ်။

1. **Manual Memory Management (C, C++):**

- Stack ကို Function ပြီးရင် အလိုအလျောက် ရှင်းပေးတာက အတူတူပါပဲ။
- ဒါပေမယ့် Heap ပေါ်မှာ နေရာယူထားတဲ့ Memory ကိုတော့ GC က အလိုအလျောက် မရှင်းပေးပါဘူး။ Developer ကိုယ်တိုင် Code ရေးပြီး (`free()` , `delete`) မဖြစ်မနေ ပြန်လွှတ်ပေးရပါတယ်။ မလွှတ်ပေးမိရင် **Memory Leak** (မှတ်ဉာဏ်မလွှတ်ဘဲ ပိတ်ဆို့နေမှု) ဆိုတဲ့ ပြဿနာကြီး ဖြစ်ပါတယ်။

1. **Ownership/Borrowing Model (Rust):**

- Rust ဟာ GC လည်း မသုံးပါဘူး၊ အဲ့ဒီအတွက် မြန်ဆန်ပါတယ်။ ဒါပေမယ့် C/C++ လို ကိုယ်တိုင် လိုက်ဖျက်ပေးစရာလည်း မလိုပါဘူး။
- Compiler ကနေ **Ownership စည်းမျဉ်းတွေ** (ဒီ Memory ကို ဘယ်သူက ပိုင်တယ်ဆိုတာ) ကို တင်းတင်းကျပ်ကျပ် သတ်မှတ်ပေးထားပါတယ်။ အဲ့ဒီ ပိုင်ရှင် Variable သက်တမ်းကုန်သွားတာနဲ့ (Out of Scope ဖြစ်တာနဲ့) Heap ပေါ်က Data ကို Compiler ကနေ အလိုအလျောက် Code (Drop) ပြန်ထည့်ပေးပြီး ဖျက်ပစ်ပါတယ်။
- ဒီစနစ်ကြောင့် dangling pointer လို ပြဿနာတွေကို များစွာ လျော့ချပေးနိုင်ပေမယ့် Memory Leak လုံးဝမဖြစ်ဘူးလို့တော့ မဆိုနိုင်ပါဘူး။ ဥပမာ reference cycle တချို့ကြောင့် leak ဖြစ်နိုင်သေးပါတယ်။

Reference နှင့် Pointer

C/C++ လို ဘာသာစကားတွေမှာ "Pointer" ဆိုတဲ့ စကားလုံးကို ကြားဖူးပါလိမ့်မယ်။ Pointer ဆိုတာ raw memory address ကို ကိုင်ထားပြီး dereference လုပ်ကာ data ကို တိုက်ရိုက် သွားဖတ်/သွားပြင်နိုင်တဲ့ အရာပါ။ Java မှာတော့ Pointer တွေကို တိုက်ရိုက် ကိုင်တွယ်ခွင့် မပေးထားပါဘူး။ အဲဒီအစား **Reference** ဆိုတဲ့ abstraction နဲ့ object ကို ညွှန်ခိုင်းထားတာ ဖြစ်ပါတယ်။

Pointer ကို တိုက်ရိုက်မြင်ရအောင် Objective-C code နဲ့ အရင်ကြည့်ရအောင်။

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
```

```
        int age = 20;
        int *ptr = &age; // ptr က age ရဲ့ memory address ကို သိမ်းထားတဲ့ pointer

        NSLog(@"age value      = %d", age);
        NSLog(@"age address    = %p", &age);
        NSLog(@"ptr value      = %p", ptr);
        NSLog(@"*ptr value     = %d", *ptr);

        *ptr = 25; // pointer ကနေတစ်ဆင့် age ရဲ့ value ကို ပြင်လိုက်တာ
        NSLog(@"updated age    = %d", age);
    }
    return 0;
}
```

ဒီ code ထဲမှာ-

- `&age` က `age` ရဲ့ **memory address** ကို ယူပေးတာပါ။
- `int *ptr` က **address ကို သိမ်းမယ့် pointer variable** ဖြစ်ပါတယ်။
- `*ptr` က pointer ညွှန်နေတဲ့ နေရာထဲက **တန်ဖိုးအစစ်** ကို သွားဖတ်တာ၊ သွားပြင်တာပါ။

ဒါကြောင့် `*ptr = 25;` လို့ ရေးလိုက်တာနဲ့ `age` ရဲ့ value ကပါ `25` ဖြစ်သွားပါတယ်။ Pointer က address ကို သိမ်းထားပြီး၊ dereference (`*ptr`) လုပ်တဲ့အခါ အဲ့ဒီ address ထဲက data ကို တိုက်ရိုက် သွားထိတာပါ။

Java မှာတော့ ဒီလို `*ptr` , `&age` လို syntax တွေကို မသုံးရပါဘူး။ ဒါပေမယ့် Object တွေကို ကိုင်တွယ်တဲ့အခါ Reference ဆိုတဲ့ အလားတူ သဘောတရားနဲ့ အလုပ်လုပ်နေပါတယ်။ အရေးကြီးတာက Java reference ကို raw address လို့ မမြင်သင့်ဘဲ JVM က စီမံထားတဲ့ opaque handle တစ်ခုလိုပဲ နားလည်သင့်ပါတယ်။

Java reference ကိုလည်း အောက်က code နဲ့ တိုက်ရိုက် နှိုင်းယှဉ်ကြည့်နိုင်ပါတယ်။

```
class User {
    String name;
}

public class Main {
    public static void main(String[] args) {
        User user1 = new User();
        user1.name = "Mg Mg";

        User user2 = user1; // object ကို copy ကူးတာ မဟုတ်ဘဲ reference ကိုပဲ copy လုပ်တာပါ။
        user2.name = "Aung Aung";

        System.out.println(user1.name); // Output: Aung Aung
        System.out.println(user2.name); // Output: Aung Aung
    }
}
```

ဒီနေရာမှာ `user1` နဲ့ `user2` က variable ၂ ခု ဖြစ်ပေမယ့် conceptual model အရ `User` object တစ်ခုတည်းကိုပဲ ညွှန်နေပါတယ်။ ဒါကြောင့် `user2` ကနေ `name` ကို ပြင်လိုက်တာနဲ့ `user1` ကနေလည်း အဲ့ဒီ ပြောင်းလဲမှုကို မြင်ရတာပါ။

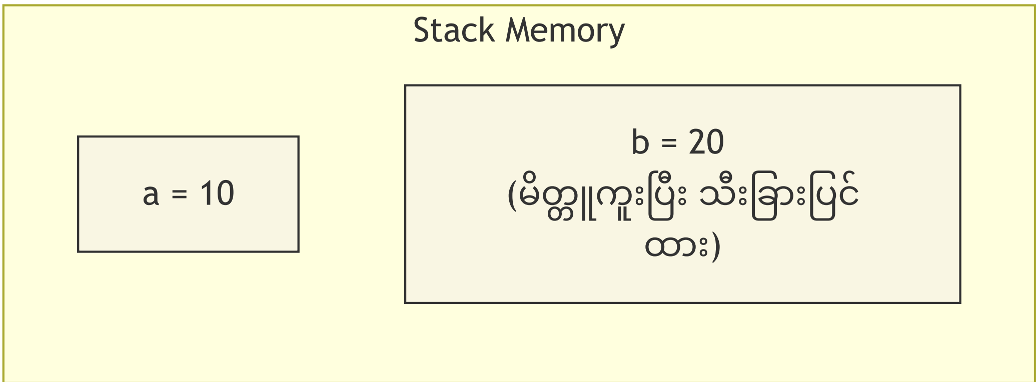
Primitives (ရိုးရိုး Data တန်ဖိုးများ)

Primitive ဆိုတာ Programming Language ကနေ အခြေခံအကျဆုံး ကြိုတင်သတ်မှတ်ထားတဲ့ ရိုးရှင်းတဲ့ Data အမျိုးအစားတွေပါ။ Java မှာ Primitive Types (၈) မျိုး ရှိပါတယ်။ အဲ့ဒါတွေကတော့-

1. **ဂဏန်းပြည့်များ:** `byte` , `short` , `int` , `long`
2. **ဒသမကိန်းများ:** `float` , `double`
3. **အက္ခရာတစ်လုံး:** `char`
4. **အမှန်/အမှား:** `boolean`

ဒီ Primitive variables တွေဟာ အရွယ်အစား ပုံသေ သတ်မှတ်ထားပြီးသားဖြစ်လို့ local variable အနေနဲ့ သင်ကြားရေး model ထဲမှာ **တန်ဖိုးအစစ်ကို တိုက်ရိုက် ကိုင်ထားတယ်** လို့ နားလည်နိုင်ပါတယ်။ ဒါပေမယ့် Java implementation အမှန်တကယ်က အမြဲ stack ပေါ်မှာပဲ သိမ်းရမယ်လို့ မဆိုလိုပါဘူး။ ဥပမာ object field သို့မဟုတ် array element အဖြစ် ရှိနေတဲ့ primitive data က heap object အတွင်းမှာလည်း ရှိနိုင်ပါတယ်။

```
int a = 10;
int b = a; // 'a' ရဲ့ တန်ဖိုး (10) ကို 'b' ထဲ လုံးဝ "မိတ္တူ" ကူးထည့်လိုက်ပါတယ်။
b = 20;    // 'b' ကို ပြင်လိုက်ပေမယ့် 'a' ကတော့ 10 အတိုင်းပဲ ဆက်ရှိနေပါတယ်။ (အချင်းချင်း မသက်ဆိုင်တော့ပါ)
System.out.println(a); // Output: 10
```



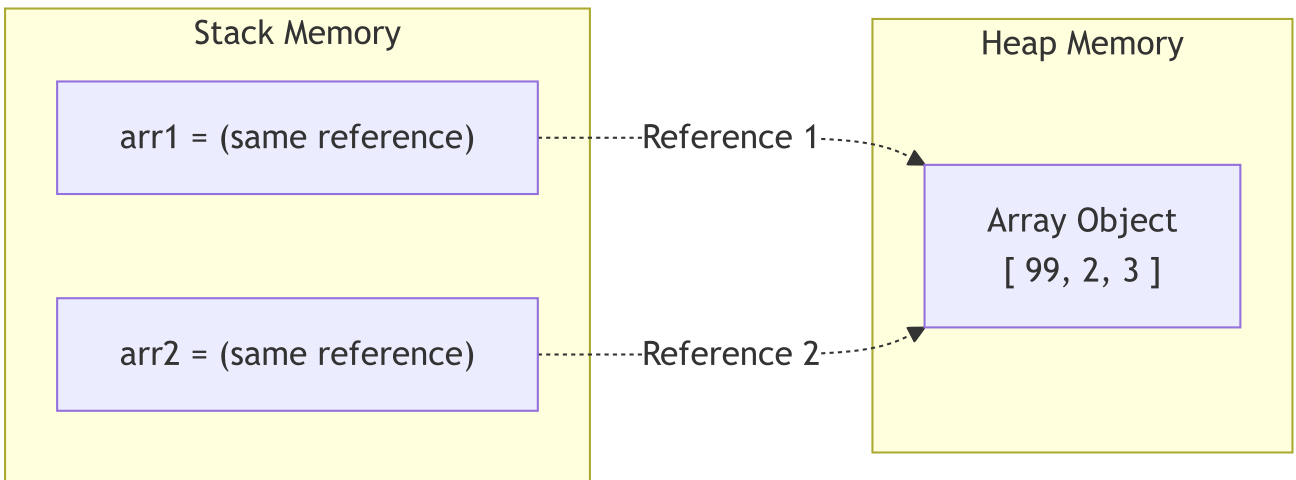
Reference Types (Objects/Arrays)

Primitive အမျိုးအစား ၈ မျိုးကလွဲရင် ကျန်တဲ့ Data တွေ အားလုံးဟာ Reference Types (ရည်ညွှန်းလိပ်စာများ) ဖြစ်ပါတယ်။

- Arrays အားလုံး (Primitive type data လေးတွေ စုထားတဲ့ Array ဖြစ်နေရင်တောင်မှ)
- Objects တွေ အားလုံး (ဥပမာ `String` , တခြား ကိုယ်တိုင်ရေးထားတဲ့ Classes တွေ `new Scanner()` , `new CustomClass()`)

ဒီအမျိုးအစားတွေကို Java teaching model ထဲမှာတော့ **object အစစ်ကို heap ဘက်မှာရှိတယ်**, variable ကတော့ အဲ့ဒီ object ကို ညွှန်တဲ့ reference value ကို ကိုင်ထားတယ်လို့ ရှင်းပြလေ့ရှိပါတယ်။ ဒါပေမယ့် ဒီ reference value ကို raw memory address အတိအကျလို့ မမြင်သင့်ပါဘူး။ Representation က ကွဲပြားနိုင်ပါတယ်။

```
int[] arr1 = {1, 2, 3}; // (Heap ပေါ်မှာ Array အစစ်ကြီး သွားဆောက်ပြီး၊ arr1 ထဲမှာ အဲ့ဒီ array ကို ညွှန်တဲ့ reference value ရှိပါတယ်။)
int[] arr2 = arr1;      // (arr1 ထဲက reference value ကို မိတ္တူကူးပေးလိုက်တာပါ။ ဒါကြောင့် Remote Control ၂ ခုက Array အစစ် တစ်ခုတည်းကို ညွှန်ပြနေပါတယ်။)

arr2[0] = 99; // Remote 'arr2' ကိုသုံးပြီး Heap ပေါ်က အစစ်ကြီးရဲ့ ပထမနေရာကို ပြင်လိုက်ပါတယ်။

// arr1 ကလည်း အတူတူပဲ ညွှန်ပြနေတဲ့ Remote ဖြစ်နေတဲ့အတွက် arr1 ရဲ့ တန်ဖိုးပါ ပြောင်းသွားပါတယ်။
System.out.println(arr1[0]); // Output: 99
```



> *(မှတ်ချက် - အပေါ်က diagram က conceptual model သာ ဖြစ်ပါတယ်။ Java program ထဲက reference value ကို raw hex memory address အနေနဲ့ တိုက်ရိုက် မြင်ရတာ မဟုတ်ပါဘူး။)*

> **ဥပမာ**
>
> - **Primitives:** ခဲတံတစ်ချောင်း ဝယ်တယ်။ သူငယ်ချင်းကို တစ်ချောင်း ထပ်ဝယ်ပေးလိုက်တယ် (မိတ္တူကူးလိုက်တယ်)။ သူငယ်ချင်းက သူ့ခဲတံကို ချိုးပစ်လိုက်ပေမယ့် ကိုယ့်ခဲတံက အကောင်းအတိုင်းပဲ ရှိနေပါတယ်။
> - **References:** အိမ်ဆောက်ပြီး သော့ တစ်ချောင်း ထုတ်တယ်။ အဲ့ဒီ သော့ကို ပွားပြီး သူငယ်ချင်းကို ပေးလိုက်တယ်။ သူငယ်ချင်းက အဲ့ဒီသော့နဲ့ အိမ်ထဲဝင်ပြီး တီဗွီကို ဖျက်ဆီးလိုက်ရင်၊ ကိုယ်ဝင်ကြည့်တဲ့အခါမှာလည်း တီဗွီက ပျက်စီးနေမှာပါပဲ။ (ဘာလို့လဲဆိုတော့ အိမ်က တစ်လုံးတည်း ဖြစ်နေပြီး နှစ်ယောက်စလုံးက တစ်အိမ်တည်းကို ညွှန်ပြနေလို့ပါ။)

ဒီ Memory Address ညွှန်ပြတဲ့ သဘောတရားကို သေချာနားလည်ရင် Array တွေ ဘာကြောင့် ဒီလို အလုပ်လုပ်တယ်ဆိုတာကို ရှင်းရှင်းလင်းလင်း မြင်လာမှာ ဖြစ်ပါတယ်။

Array ဆိုတာ ဘာလဲ

Array ဆိုတာ Data တွေ တည်ဆောက်သည့်အထဲမှာ အခြေခံ အကျဆုံး နဲ့ အသုံးအများဆုံး Structure တစ်ခုပါ။ သူက Data တွေကို RAM မှာ တစ်ဆက်တည်း သိမ်းဆည်းထားတာ ဖြစ်ပါတယ်။ Array မှာ *(မှတ်ချက် - Java မှာ `int[]` လို primitive array ဆိုရင် တန်ဖိုးတွေကိုယ်တိုင် array object အတွင်းမှာ ဆက်တိုက်ရှိပြီး၊ `String[]` လို object array ဆိုရင်တော့ object အစစ်တွေ မဟုတ်ဘဲ object တွေကို ညွှန်တဲ့ reference value တွေကသာ ဆက်တိုက်ရှိပါတယ်။ JVM ကလည်း physical RAM ပေါ်က နေရာချထားပုံကို abstraction လုပ်ထားပါတယ်။)*

- Static Array
- Dynamic Array

ဆိုပြီးရှိပါတယ်။

Static Array

Static Array ဆိုတာကတော့ ယူမည့် အခန်း အရေအတွက် ကို ကြေငြာပြီးရင် ပြန်ပြင်မရတော့ပါဘူး။ ယူမည့် အခန်းအရေအတွက်ကို အရင် ကြေငြာရပါတယ်။ Java မှာ ဆိုရင် `int [] arr = new int[10];` လို့ ရေးပါတယ်။ ထပ်ချဲ့လို့ မရသည့် အတွက် Static Array လို့ ခေါ်ပါတယ်။

Array က ဘာလို့ မြန်တာလဲ ? ဥပမာ Array ရဲ့ အစ (base address) က `1000` ဖြစ်ပြီး `int` တစ်ခု က 4 bytes ဆီ နေရာယူတယ် ဆိုပါစို့။ `arr[5]` ၆ ခု မြောက်အခန်း ကို လိုချင်ရင် loop ပတ်ပြီး ရှာနေဖို့ မလိုပါဘူး။

```
Address = BaseAddress + (Index * DataSize)
Address = 1000 + (5 * 4) = 1020
```

ဒါဆိုရင် address `1020` ကို တန်းသွားလိုက်တာနဲ့ data ရပါတယ်။ *(`1000` က ရှင်းအောင်သုံးထားတဲ့ ဥပမာ decimal address ဖြစ်ပြီး၊ တကယ့် memory address တွေကို hexadecimal ဥပမာ `0x1000` နဲ့ ပြသလေ့ရှိပါတယ်။)*

ဒါကြောင့် index သိရင် တန်ဖိုးကို **ဖတ်တာ (read)** နဲ့ **ပြင်တာ (write)** က $O(1)$ ပဲ ကြာပါတယ်။ ဒါပေမယ့် ဒီ $O(1)$ က index နဲ့ တိုက်ရိုက် access/update လုပ်တာ ကိုသာ ဆိုလိုပါတယ်။ နောက်မှာ ပြမယ့် အလယ်တစ်နေရာမှာ **ထည့်တာ (insert)** ၊ **ဖျက်တာ (delete)** ကတော့ $O(1)$ မဟုတ်ဘဲ ပိုကြာပါတယ်။

```java
import java.util.Arrays;

public class ArrayExample {
    public static void main(String[] args) {
        // အခန်း ၆ ခန်းပါသော Array တည်ဆောက်ခြင်း (Data ၅ ခုပဲ ထည့်ထားပါမည်)
        int[] numbers = new int[6];
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;
        // numbers[5] သည် တန်ဖိုးမထည့်ရသေးသဖြင့် 0 ဖြစ်နေပါမည်။
```

```
            System.out.println("Before Insertion: " + Arrays.toString(numbers));
        }
    }
```

Array ၏ Index နှင့် Zero-based

Array ထဲက အခန်းတွေကို ရေတွက်တဲ့အခါ `1` ကနေ မဟုတ်ဘဲ `0` ကနေ စတင်ရေတွက်ပါတယ်။ ဒါကို **zero-based indexing** လို့ ခေါ်ပါတယ်။ ဒါကြောင့်-

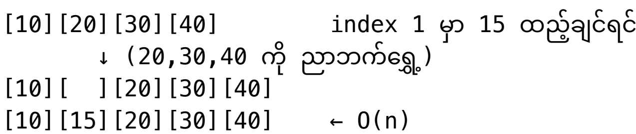
- ပထမဆုံးအခန်း = `arr[0]`
- ဒုတိယအခန်း = `arr[1]`
- အခန်း `n` ခုပါတဲ့ Array ရဲ့ နောက်ဆုံးအခန်း = `arr[n-1]`

ဘာကြောင့် `0` ကနေ စသလဲဆိုတော့ Index က base address ကနေ ဘယ်လောက် ဝေးတယ်ဆိုတဲ့ **offset** ဖြစ်လို့ပါ။ ပထမဆုံးအခန်းက base address မှာ တည့်တည့်ရှိတဲ့အတွက် offset က `0` ဖြစ်ပါတယ် (`Address = 1000 + (0 × 4) = 1000`)။

Array မှာ Insert နှင့် Delete ကုန်ကျစရိတ်

Read/Write က $O(1)$ ဖြစ်ပေမယ့် Array အလယ်တစ်နေရာမှာ data **ထည့် (insert)** ၊ **ဖျက် (delete)** ဖို့ဆိုရင်တော့ စျေးကြီးပါတယ်။ Array က data တွေ တစ်ဆက်တည်း ရှိနေဖို့ လိုတဲ့အတွက်ကြောင့်ပါ။

- **အလယ် (သို့) အစမှာ Insert:** ထည့်မယ့်နေရာ နောက်က element အကုန်လုံးကို တစ်နေရာစီ ညာဘက် ရွှေ့ပေးရတယ်။ ဒါကြောင့် အဆိုးဆုံး case မှာ $O(n)$ ဖြစ်ပါတယ်။
- **အလယ် (သို့) အစမှာ Delete:** ဖျက်လိုက်တဲ့ နေရာက ဟာသွားတာကို ဖို့ဖို့ နောက်က element တွေကို ဘယ်ဘက် ပြန်ရွှေ့ပေးရတယ်။ ဒါလည်း $O(n)$ ဖြစ်ပါတယ်။
- **အဆုံး (နောက်ဆုံးနေရာ) မှာ Insert/Delete:** element ရွှေ့စရာ မလိုတဲ့အတွက် $O(1)$ ဖြစ်ပါတယ် (နေရာလွတ်ရှိမှ)။



ဒါကြောင့် Array ရဲ့ operation cost ကို အောက်လို မှတ်ထားနိုင်ပါတယ်-

Operation	Time Complexity
Index နဲ့ Access (read)	$O(1)$
Index နဲ့ Update (write)	$O(1)$
အဆုံးမှာ Insert (append)	$O(1)$ *

အလယ်/အစမှာ Insert	$O(n)$
အလယ်/အစမှာ Delete	$O(n)$
Search (value ရှာ)	$O(n)$

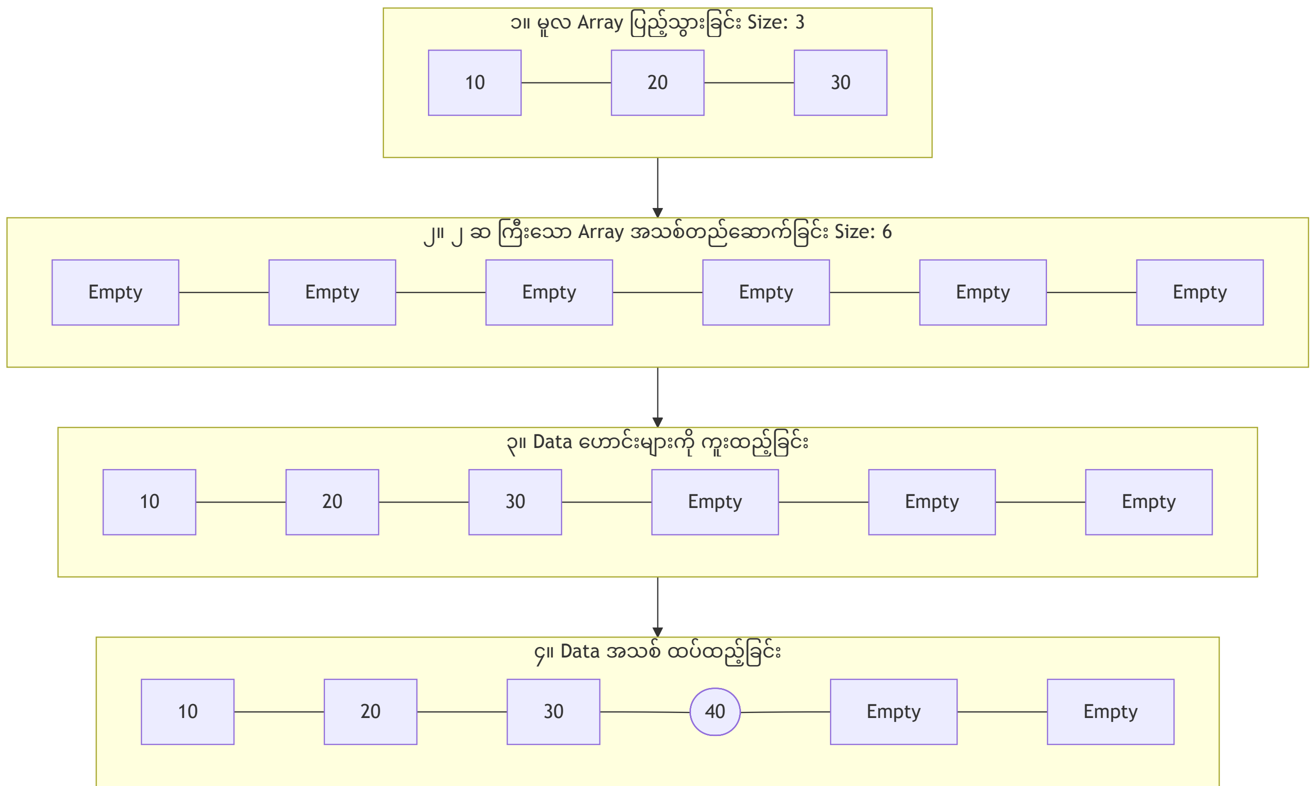
> \ Append မှတ်ချက်:* Static Array ဟာ အရွယ်အစား ပုံသေဖြစ်တဲ့အတွက် သတ်မှတ်ထားတဲ့ length ထက် ကျော်ပြီး တကယ် append လုပ်လို့ မရပါဘူး။ နေရာလွတ် (capacity) ကို ကြို ယူထားပြီး logical size ကို သီးခြား track လုပ်ထားမှသာ အဆုံးမှာ ထည့်တာ $O(1)$ ဖြစ်ပါ တယ်။ ဘယ်လောက်ဆန့်မလဲ ကြိုမသိဘဲ ထည့်ချင်သလောက် ထည့်လို့ရတဲ့ append ($O(1)$ **amortized**) ကိုတော့ အောက်က Dynamic Array မှာ ရရှိပါတယ်။

Dynamic Array

Static Array တွေမှာ အခန်းကို ကြိုသတ်မှတ်ရသည့် ပြဿနာ ဖြေရှင်းဖို့ Dynamic Array တွေ ပေါ်ပေါက်လာပါတယ်။ Java မှာ `ArrayList` လို့ လူသိများပါတယ်။ သူကတော့ Data တွေကို ထည့်ချင် သလောက် ထည့်လို့ရပြီး Size ကို အလိုလျောက်ညှိပေးသွားပါတယ်။

Dynamic Array အလုပ်လုပ်ပုံကတော့

1. Initial Capacity : စစချင်းမှာ အခန်း အနည်းငယ် (ဥပမာ ၅ ခန်း) ပါသည့် Array တစ်ခု ကို နေရာယူ လိုက်တယ်။
2. Checking Space: Data အသစ်တစ်ခု ထည့်လိုက်တိုင်း နေရာလောက်သေးလား စစ်တယ်။
3. Resizing: အကယ်၍ နေရာပြည့်သွားပြီဆိုရင်
    1. အရင် Array ထက် ၂ ဆ ကြီးသည့် Array တစ်ခု ထပ်ဆောက်လိုက်တယ်။
    2. အရင် Array အဟောင်းက Data တွေကို အသစ်ထဲ တစ်ခန်းဆီ Copy လုပ်တယ်။
    3. ပြီးမှ အဟောင်း ကို ဖျက်ချပြီး အသစ်ကို ဆက်သုံးပါတယ်။




```java
import java.util.ArrayList;

public class DynamicArrayExample {
    public static void main(String[] args) {
        // Dynamic Array တည်ဆောက်ခြင်း (Size ကြိုပြောစရာမလို)
        ArrayList<Integer> list = new ArrayList<>();

        // Data ထည့်ခြင်း - O(1) [Amortized]
        list.add(10);
        list.add(20);
        list.add(30);

        // အလယ်မှာ ထည့်ခြင်း - O(n) [နောက်ကကောင်တွေ ရွှေ့ရလို့]
        list.add(1, 15);

        // Data ဖတ်ခြင်း - O(1)
        System.out.println("Element at index 1: " + list.get(1));

        // လက်ရှိ ဘယ်နှစ်ခု ရှိနေပြီလဲ
        System.out.println("Current size: " + list.size());
    }
}
```

Dynamic Array ဟာ resize ပြန်လုပ်သည့် အချိန်မှာ နှေးသွားနိုင်ပါတယ်။ ဒါကြောင့် size အတိအကျ သိလျှင် Static Array ကို အသုံးပြုသင့်ပါတယ်။

String ဆိုတာ ဘာလဲ

String ဆိုတာ စာလုံး (character) တွေ အစဉ်လိုက် စုထားတာ ဖြစ်ပါတယ်။ Conceptually အားဖြင့် String ကို Array လိုပဲ index နဲ့ access လုပ်လို့ရတဲ့ character/code unit sequence တစ်ခုလို့ မြင်နိုင်ပါတယ်။ ဒါကြောင့် `charAt(i)` နဲ့ တစ်လုံးချင်းစီကို $O(1)$ နဲ့ ယူနိုင်ပါတယ်။

> *(မှတ်ချက် - အတွင်းပိုင်း implementation ကတော့ Java version အလိုက် ကွဲပြားပါတယ်။ ခေတ်သစ် Java မှာ String ကို `char[]` အစား `byte[]` (Compact Strings) နဲ့ သိမ်းတာမျိုး ရှိနိုင်ပါတယ်။ ထို့အပြင် `charAt()` က ပြန်ပေးတဲ့ `char` က **UTF-16 code unit** တစ်ခုဖြစ်ပြီး user မြင်ရတဲ့ စာလုံးတစ်လုံးနဲ့ အမြဲ တူချင်မှ တူပါမယ်။ ဥပမာ မြန်မာစာလို script မှာ မြင်ရတဲ့ စာလုံး/syllable တစ်လုံးဟာ code point/code unit များစွာနဲ့ ဖွဲ့စည်းထားနိုင်ပါတယ်။)*

```
String s = "HELLO";
//          H  E  L  L  O
// index    0  1  2  3  4

System.out.println(s.charAt(0)); // H
System.out.println(s.charAt(4)); // O
System.out.println(s.length());  // 5
```

ဆိုလိုတာက `"HELLO"` ကို conceptually အောက်လို character array အဖြစ် မြင်နိုင်ပါတယ်-

```
index :  0    1    2    3    4
char  : 'H'  'E'  'L'  'L'  'O'
```

String Immutability

အပေါ်က Mutable/Immutable အပိုင်းမှာ ပြောခဲ့သလို Java မှာ `String` က **immutable** ဖြစ်ပါတယ်။ ဆိုလိုတာက String တစ်ခု ဖန်တီးပြီးရင် အထဲက character တွေကို တိုက်ရိုက် ပြန်ပြင်လို့ မရပါဘူး။

```
String s = "HELLO";
// s.charAt(0) = 'J';    // ❌ ဒီလိုလုပ်လို့ မရပါဘူး
s = s.replace('H', 'J'); // String အသစ် "JELLO" ကို ဖန်တီးပြီး s ကို ပြန်ညွှန်တာ
```

ဒါကြောင့် String ကို ထပ်ခါထပ်ခါ ပေါင်း (concatenate) လုပ်ရင် အကြိမ်တိုင်း String object အသစ် ဖန်တီးနေရပြီး ဈေးကြီးနိုင်ပါတယ်။ String အများကြီး ပြောင်းလဲဖို့ လိုရင် Java မှာ `StringBuilder` (mutable) ကို သုံးသင့်ပါတယ်။

```
StringBuilder sb = new StringBuilder();
sb.append("HE");
sb.append("LLO");
String result = sb.toString(); // "HELLO"
```

Common Operations နှင့် ကုန်ကျစရိတ်

Array/String မှာ အသုံးများတဲ့ operation တွေနဲ့ သူတို့ရဲ့ time complexity-

Operation	ရှင်းလင်းချက်	Time Complexity
Access	`arr[i]` index နဲ့ ဖတ်ခြင်း	$O(1)$
Update	`arr[i] = x` index နဲ့ ပြင်ခြင်း	$O(1)$
Append (Dynamic Array)	အဆုံးမှာ ထည့်ခြင်း	$O(1)$ amortized
Insert/Delete (အလယ်)	element ရွှေ့ရခြင်း	$O(n)$
Slice (subarray/substring)	အပိုင်းတစ်ခု ဖြတ်ယူ၍ အသစ်ကူးခြင်း	$O(k)$ (k = ဖြတ်ယူသော အရှယ်)
Search	value တစ်ခု ရှာခြင်း	$O(n)$

> **Append vs Slice မှတ်ချက်:** Append ($O(1)$ amortized) က နေရာရှိရင် တန်းထည့်ရုံပါပဲ။ Slice ကတော့ ဖြတ်ယူတဲ့ element အရေအတွက်အလိုက် အသစ်ကူးရတာကြောင့် $O(k)$ ကုန်ပါတယ်။

Real-world Examples

Array နဲ့ String ကို နေ့စဉ် software development မှာ နေရာတိုင်းနီးပါး တွေ့ရပါတယ်-

- **Product list:** e-commerce app မှာ ကုန်ပစ္စည်းတွေကို array/list အဖြစ် သိမ်းပြီး index နဲ့ ဖော်ပြတယ်။
- **Transaction list:** banking app မှာ ငွေသွင်း/ထုတ် မှတ်တမ်းတွေကို list အဖြစ် စီထားတယ်။
- **CSV row:** CSV file တစ်ကြောင်းကို `","` နဲ့ split လုပ်လိုက်ရင် field တွေပါတဲ့ array ရတယ် (ဥပမာ `["Mg Mg","20","Yangon"]`)။

Common Problems

ဒီအခန်းက concept တွေကို လေ့ကျင့်ဖို့ classic problem အချို့-

၁။ Max/Min ရှာခြင်း

```
int[] nums = {3, 7, 1, 9, 4};
int max = nums[0];
int min = nums[0];
for (int i = 1; i < nums.length; i++) {
    if (nums[i] > max) max = nums[i];
    if (nums[i] < min) min = nums[i];
}
System.out.println(max); // 9
System.out.println(min); // 1
```

Array တစ်ခေါက် (one pass) လှည့်ရုံနဲ့ max နဲ့ min နှစ်ခုလုံး ရတဲ့အတွက် $O(n)$ ဖြစ်ပါတယ်။

၂။ String Reverse လုပ်ခြင်း

```
String s = "HELLO";
StringBuilder sb = new StringBuilder(s);
System.out.println(sb.reverse().toString()); // OLLEH
```

String က immutable ဖြစ်လို့ `StringBuilder` သုံးတာ ပိုသင့်ပါတယ်။ $O(n)$ ဖြစ်ပါတယ်။

၃။ Character ရေတွက်ခြင်း (Count characters)

```
String s = "banana";
int[] count = new int[26]; // a-z
for (char c : s.toCharArray()) {
    count[c - 'a']++;
}
System.out.println(count['a' - 'a']); // 'a' ၃ လုံး
```

စာလုံးတစ်လုံးချင်း လှည့်ရတဲ့အတွက် $O(n)$ ဖြစ်ပါတယ်။

၄။ Duplicate များ ဖယ်ရှားခြင်း (Remove duplicates)

```
int[] nums = {1, 2, 2, 3, 3, 3};
java.util.LinkedHashSet<Integer> set = new java.util.LinkedHashSet<>();
for (int n : nums) set.add(n);
System.out.println(set); // [1, 2, 3]
```

Set ကို သုံးပြီး ထပ်နေတဲ့ တန်ဖိုးတွေကို ဖယ်လိုက်တာ ဖြစ်ပါတယ်။ (Set အကြောင်းကို နောက်အခန်းမှာ ဆက်ဖတ်ပါ။)

အခန်း ၄ - Hash Table / Dictionary

Hashing

Hashing ဆိုတာ Data တစ်ခုကို သတ်မှတ်ထားသော Function တစ်ခု (Hash Function) မှတဆင့် ဖြတ်ကာ သိမ်းဆည်းမည့် နေရာ (Index) တွေ ရှာဖွေသည့် နည်းစနစ်ပါ။ ရလဒ်ကတော့ $O(1)$ ဖြင့် Data ထည့်ခြင်း၊ ရှာဖွေခြင်း၊ ဖျက်ခြင်း ပြုလုပ်နိုင်ခြင်း ဖြစ်ပါတယ်။

Hash Function

Hash Function ဆိုတာ Key တစ်ခုကို Integer (Index) တစ်ခုအဖြစ် ပြောင်းပေးသည့် Function ပါ။ ဥပမာ —

```
index = key % tableSize
// key=25, tableSize=10  =>  index = 25 % 10 = 5
```

Hash Function ကောင်းတစ်ခုမှာ အောက်ပါ ဂုဏ်သတ္တိ (properties) တွေ ရှိရပါမယ်။

- **Deterministic** — တူညီသော Input ကို အမြဲ တူညီသော Output ထုတ်ပေးရမည်
- **Uniform Distribution** — Data တွေကို Array တစ်ခုလုံးမှာ ညီညာအောင် ဖြန့်ကျက်ပေးရမည်
- **Fast Computation** — Hash တွက်ချက်ချိန် $O(1)$ ဖြစ်ရမည်

Hash Table

Hash Table ဆိုတာ Hash Function ကို အသုံးပြု၍ Key-Value Pair တွေ သိမ်းသည့် Data Structure (Concept) ပါ။ လက်တွေ့အသုံးပြုရာမှာတော့ Hash Table နဲ့ Hash Map ဆိုတဲ့ စကားလုံးနှစ်ခုကို သဘောတရား အတူတူပဲလို့ မှတ်ယူပြီး အပြန်အလှန် (interchangeably) သုံးလေ့ရှိပါတယ်။

Java မှာဆိုရင် `HashMap` ဆိုတာ Hash Table ရဲ့ အဓိက Implementation တစ်ခု ဖြစ်ပါတယ်။

> မှတ်ချက် - Java မှာ `Hashtable` ဆိုတဲ့ Thread-safe ဖြစ်တဲ့ class အဟောင်းတစ်ခု ရှိပေမယ့်၊ ယခုအခါမှာတော့ ပိုမြန်တဲ့ `HashMap` ကိုသာ အများဆုံး အသုံးပြုကြပါတယ်။

Concept အရ **Hash Table** လို့ ခေါ်ပြီး၊ ကုဒ်ရေးတဲ့အခါ**Hash Map** လို့ ခေါ်ပါတယ်။

```java
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
```

```
        // Data ထည့်ခြင်း — O(1) average
        map.put("apple", 3);
        map.put("banana", 5);
        map.put("cherry", 1);

        // Data ရှာဖွေခြင်း — O(1) average
        System.out.println(map.get("apple")); // 3

        // Key ရှိမရှိ စစ်ဆေးခြင်း
        System.out.println(map.containsKey("banana")); // true

        // Data ဖျက်ခြင်း — O(1) average
        map.remove("cherry");

        // Key အားလုံး Iterate လုပ်ခြင်း — O(n)
        for (String key : map.keySet()) {
            System.out.println(key + " -> " + map.get(key));
        }
    }
}
```

Collision

Key နှစ်ခု မတူညီသော်လည်း Hash Function မှ တူညီသော Index ထုတ်ပေးသောအခါ Collision(ထပ်မိခြင်း) ဖြစ်ပါသည်။ ဥပမာ `key=5` နှင့် `key=15` တို့ကို `tableSize=10` ဖြင့် hash လုပ်ပါက နှစ်ခုလုံး `index 5` ကို ရရှိသည်။ Collision ကို ဖြေရှင်းသည့် နည်းလမ်းများမှာ —

- **Chaining** — တူညီသော Index တွင် Linked List ဖြင့် Data များ ချိတ်ဆက် သိမ်းဆည်းခြင်း (Linked List အကြောင်းကို အခန်း ၈ တွင် ဖော်ပြပါမည်)
- **Open Addressing** — Collision ဖြစ်ပါက အနီးစပ်ဆုံး လွတ်နေသော Index ကို ရှာဖွေ သိမ်းဆည်းခြင်း

Open Addressing (နေရာ ရှာဖွေ သိမ်းဆည်းခြင်း)

Collision ဖြစ်ပါက Linked List မသုံးဘဲ Table ထဲတွင်ပင် နောက် index ကို ရှာ၍ သိမ်းပါသည်။ နည်းလမ်း ၃ မျိုး ကို အသုံးပြုသည်။

- Linear Probing — collision ဖြစ်ရင် index + 1, +2, +3 အတိုင်း တစ်ခုချင်း ရှာခြင်း
- Quadratic Probing — $+1^2$, $+2^2$, $+3^2$ ကွာဟချက်ဖြင့် ရှာခြင်း (clustering လျော့ချစေသည်)
- Double Hashing — ဒုတိယ hash function တစ်ခု သုံး၍ ကွာဟချက် ရှာခြင်း

တို့ ဖြစ်သည်။

HashSet

`HashSet` ဆိုတာ `HashMap` ရဲ့ Key များသာ သိမ်းသည့် Version တစ်ခုပါ (Value မသိမ်း)။ ထပ်တလဲလဲ ရောက်လာသော Element များကို အလိုအလျောက် ဖယ်ရှားပေးသောကြောင့် Unique Element များ ထိန်းသိမ်းရာတွင် အသုံးဝင်သည်။

```java
import java.util.HashSet;

HashSet<String> set = new HashSet<>();
set.add("apple");
set.add("apple");  // ဒုတိယ "apple" ထပ်မထည့်ဘူး
set.add("banana");
System.out.println(set.contains("apple")); // true — O(1)
System.out.println(set.size());            // 2
```

Time Complexity

Operation	Average Case	Worst Case
put / add	$O(1)$	$O(n)$
get / contains	$O(1)$	$O(n)$
remove	$O(1)$	$O(n)$
size	$O(1)$	$O(1)$
iterate	$O(n)$	$O(n)$

> **မှတ်ချက်:** Worst Case $O(n)$ ဆိုသည်မှာ Hash Collision အများကြီး ဖြစ်ပေါ်သောကြောင့် ဖြစ်ပါသည်။ ကောင်းမွန်သော Hash Function ဖြင့် Collision ကို နည်းစေ၍ Average $O(1)$ ကို ထိန်းသိမ်းနိုင်ပါသည်။

Hashing အသုံးပြုသည့် အခြေအနေများ

- **Duplicate Detection** — HashSet ဖြင့် O(n) ကြာသော ကြည့်ရှုမှုကို $O(1)$ ဖြစ်စေနိုင်သည်
- **Frequency Count** — HashMap ဖြင့် Key=Element, Value=Count ထားကာ $O(n)$ တစ်ကြိမ် ဖြတ်ကာ ရေတွက်နိုင်သည်
- **Two Sum ကဲ့သို့သော ပြဿနာများ** — Array ကို Loop ပတ်ကာ Complement ကို HashMap တွင် စစ်ဆေးခြင်း
- **Cache / Memoization** — Caching သဘောတရားအရ တွက်ချက်မှု ရလဒ်များကို HashMap တွင် သိမ်းကာ ထပ်တွက်မနေစေရ

Real-world မှာ ဘယ်လိုသုံးလဲ

Hash Table ကို နေ့စဉ် coding မှာ ဘယ်လိုသုံးလဲ ကြည့်ရအောင်။ Backend / Frontend development မှာ အသုံးအများဆုံးက **lookup** ပါ။

၁။ ID နဲ့ Object ရှာခြင်း (Lookup) — Database ကနေ user list ဆွဲထုတ်ပြီး `userId` နဲ့ ထပ်ခါထပ်ခါ ရှာရတဲ့အခါ List ကို loop ပတ်ရင် တစ်ကြိမ်လျှင် $O(n)$ ဖြစ်တယ်။ Map ထဲ တစ်ခါထည့်ထားရင် ရှာတိုင်း $O(1)$ ဖြစ်သွားတယ်။

```
// List ထဲ loop ပတ်ရှာ → ရှာတိုင်း O(n)
// Map ထဲ index လုပ်ထား → ရှာတိုင်း O(1)
Map<Integer, User> userById = new HashMap<>();
for (User u : users) {
    userById.put(u.getId(), u);
}

User u = userById.get(42); // O(1)
```

အလားတူ `productId → product` , `permission → allowed?` တို့ကိုလည်း Map နဲ့ index လုပ်ထားလေ့ရှိတယ်။

၂။ Status code အလိုက် API error ရေတွက်ခြင်း (Frequency Count) — Log တွေထဲက error တွေကို status code အလိုက် ဘယ်နှစ်ကြိမ် ဖြစ်လဲ ရေတွက်ဖို့ Map ကို `key=statusCode` , `value=count` ထားပြီး တစ်ကြိမ်ဖြတ်ရုံပါပဲ။

```
Map<Integer, Integer> errorCount = new HashMap<>();
for (int status : responseStatuses) {
    if (status >= 400) {
        errorCount.merge(status, 1, Integer::sum); // count[status] += 1
    }
}
// {404: 12, 500: 3, ...}
```

ဒီ "Map နဲ့ ရေတွက်တယ်" ဆိုတဲ့ pattern က နောက်က **Valid Anagram**, **Top K Frequent** ပုစ္ဆာတွေမှာ ပြန်တွေ့ရမှာပါ။

၃။ တွက်ပြီးသားကို ပြန်မတွက်ခြင်း (Cache) — တူညီတဲ့ input အတွက် ထပ်ခါထပ်ခါ တွက်နေမယ့်အစား ရလဒ်ကို Map ထဲ သိမ်းထားလိုက်တယ်။ ဒုတိယအခါ ခေါ်ရင် Map ထဲက တန်းထုတ်ပေးနိုင်ပါတယ်။

```
Map<Integer, Long> cache = new HashMap<>();

long compute(int n) {
    if (cache.containsKey(n)) return cache.get(n); // cache hit — O(1)

    long result = expensiveWork(n);  // ပထမအကြိမ်မှာသာ တွက်တယ်
    cache.put(n, result);
    return result;
}
```

ဒါက Dynamic Programming အခန်းမှာ ပြောမယ့် **Memoization** ရဲ့ အခြေခံပါပဲ။

Questions

Array နှင့် Hash Table , Hash Set တို့ကို သိပြီ ဆိုလျှင် ကျွန်တော်တို့ တချို့ ပုစ္ဆာ တွေကို ကြည့်ရအောင်။

Contains Duplicate

Array ထဲတွင် duplicate ရှိမရှိ ရှာပါ။

ဥပမာ

```
Input: nums = [1, 2, 3, 3]

Output: true

Input: nums = [1, 2, 3, 4]

Output: false
```

> Time နှင့် Space complexity နှစ်ခုလုံး $O(n)$ ဖြစ်ရမည်။

ဒီ ပုစ္ဆာက array ကို စလေ့လာသည့် သူတွေ အတွက် အကောင်းဆုံး လေ့ကျင့်လို့ရတာပဲ။ သတိထားရမည့် အချက်က O(n) ဖြစ်နေတာပဲ။

ပုံမှန် ဆိုလျှင် အခန်း တစ်ခု ကို ယူ ၊ အစ ကနေ အကုန်တိုက်သည့် ပုံစံ ဖြင့်သွားရသည်။ ဒါဆိုရင် $O(n^2)$ ဖြစ်သွားပါမယ်။ $O(n)$ ရအောင် `HashSet` ကို သုံးနိုင်ပါတယ်။

```java
public boolean hasDuplicate(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int n : nums) seen.add(n);
    return seen.size() < nums.length;
}
```

Array တွေ အကုန်လုံးကို Set ထဲထည့်လိုက်တယ်။ ပြီးရင် Set size နဲ့ array size ယှဥ်တယ်။ မတူရင် duplicate ရှိတယ် ဆိုတာ သိနိုင်ပါတယ်။

Valid Anagram

An **anagram** ဆိုတာကတော့ string ထဲမှာ ပါသည့် characters တွေကနေ နောက်ထပ် string ထဲမှာ လည်း ပါနေသည့် သဘောပါ။ ဥပမာ

```
Input: s = "racecar", t = "carrace"

Output: true

Input: s = "jar", t = "jam"

Output: false
```

> $O(n+m)$ Time Complexity ဖြစ်ရမည်။

ဒီ ပုစ္ဆာ ဆိုရင် အဖြေ နှစ်မျိုး ရှိနိုင်တယ်။ Space Complexity က ဘယ်လောက် လိုချင်သလဲ။ $O(1)$ သာဖြစ်ခဲ့ရင် character က fix length (a-z) ဖြစ်မယ်။ character က latin တွေ ပါ ပါလာနိုင်လား။ ပါလာနိုင်ရင်တော့ dynamic ဖြစ်ပြီး $O(1)$ နဲ့ မရနိုင်တော့ဘူး။ Space complexity က $O(n)$ မှ အဆင်ပြေမယ်။

အရင်ဆုံး `a-z` ပဲ ရှိမယ် သတ်မှတ်ပြီး ရေးကြည့်ရအောင်။

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;

    int[] count = new int[26]; //a မှ z အထိ

    for (int i = 0; i < s.length(); i++) {
        count[s.charAt(i) - 'a']++;//a-a = 0 , b-a = 1
        count[t.charAt(i) - 'a']--;
    }
    //count ထဲဝင် a-z count ဖြစ်သွားပါပြီ။
    for (int c : count) {
        if (c != 0) return false;
    }

    return true;
}
```

ဒီ Code Idea ကတော့ ရိုးရှင်းပါတယ်။ `a-z` အခန်း ယူထားတယ်။ `s` ထဲမှာ `a` ပါရင် အခန်း ကို `+1` လုပ်လိုက်တယ်။ `t` ထဲမှာ `a` ပါရင် `-1` လုပ်လိုက်တယ်။ ဒါဆိုရင် အခန်းက မူရင်း အတိုင်း `0` ပြန် ဖြစ်သွားမှာပေါ့။ တကယ်လို့ `s` မှာ `a` ပါပြီး `t` မှာ `a` မပါရင် အခန်းထဲမှာ `0` ပြန်မဖြစ်ပါဘူး။ ဒါဆို ရင် anagram မဖြစ်တာ သေချာသွားပြီ။

`s.charAt(i) - 'a'` ဆိုသည့် သဘောကတော့ `a - a = 0` ဖြစ်သည့် သဘောပါ။ `b - a` ဆိုရင် 1 ပါ။ တနည်းပြောရင် အခန်း 0 ကနေ 25 ထိ တွက်သည့် သဘောပါပဲ။

အကယ်၍ character က `a-z` အပြင် တခြား `latin character` တွေ ပါပါလာခဲ့ရင် array size fix က အဆင်မပြေတော့ပါဘူး။ အဲဒီ အတွက် HashMap ကို ပြောင်းသုံးပါမယ်။

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;

    Map<Character, Integer> count = new HashMap<>();

    for (int i = 0; i < s.length(); i++) {
        count.merge(s.charAt(i), 1, Integer::sum);
        count.merge(t.charAt(i), -1, Integer::sum);
    }

    for (int val : count.values()) {
        if (val != 0) return false;
    }

    return true;
}
```

Dictionary key ကို သုံးပြုပြီး +1 , -1 လုပ်သွားတာပါပဲ။ loop ပြီးသွားသည့် အခါမှာ အကုန် 0 တွေ ဖြစ်နေရမှာပါ။ `count.merge` ဆိုတာကတော့ လက်ရှိ ရှိနေသည့် value ကို ပေါင်းပေးတာပါ။

merge မသုံးပဲ put သုံးခဲ့ရင်တော့ အခုလို ရေးရပါမယ်။

```
count.put(s.charAt(i), count.getOrDefault(s.charAt(i), 0) + 1);
```

Two Sum

Given an array of integers `nums` and an integer `target` , return *indices of the two numbers such that they add up to* `target` .

You may assume that each input would have ***exactly*** **one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: numbers = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because numbers[0] + numbers[1] == 9, we return [0, 1].
```

Example 2:

```
Input: numbers = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: numbers = [3,3], target = 6
Output: [0,1]
```

Constraints:

- `2 <= nums.length <= 10^4`
- `-10^9 <= nums[i] <= 10^9`
- `-10^9 <= target <= 10^9`
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?

ဒီ ပုစ္ဆာ က ကြည့်လိုက်တာနဲ့ ရိုးရှင်းပါတယ်။ ပုံမှန် သမာရိုးကျ နည်းလမ်း စဉ်းစားရင် loop နှစ်ခါ ပတ်လိုက်ရင် ရပါပြီ။ ဒါပေမယ့် time complexity က $O(n^2)$ ဖြစ်နေပါတယ်။ ဒါကြောင့် ကျွန်တော်တို့တွေ ပုံမှန် သုံးနေကျ Hash Table နဲ့ ပဲ စဉ်းစားပါမယ်။

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> seen = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];

            if (seen.containsKey(complement)) {
                return new int[] { seen.get(complement), i };
            }

            seen.put(nums[i], i);
        }

        return new int[] {}; // unreachable given valid input
    }
}
```

Group Anagram

Given an array of strings `strs` , group the anagrams together. You can return the answer in **any order**.

Example 1:

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

Explanation:

- There is no string in strs that can be rearranged to form `"bat"` .
- The strings `"nat"` and `"tan"` are anagrams as they can be rearranged to form each other.
- The strings `"ate"` , `"eat"` , and `"tea"` are anagrams as they can be rearranged to form each other.

Example 2:

```
Input: strs = [""]

Output: [[""]]
```

Example 3:

```
Input: strs = ["a"]

Output: [["a"]]
```

ဒီပုစ္ဆာမှာ Anagram ဖြစ်တယ်ဆိုရင် သိနိုင်မယ့် နည်းက သက်ဆိုင်ရာ String တွေကို Sort လုပ်လိုက်တာပါပဲ။ ဥပမာ `eat` ရော `tea` ရော `ate` ရော အားလုံးကို sort လုပ်လိုက်ရင် `aet` ဆိုပြီး ရလာပါမယ်။ ဒီတော့ `aet` ကို Map ရဲ့ Key အနေနဲ့ သုံးပြီး တူရာတွေကို Group ဖွဲ့လို့ ရပါတယ်။

ဒါပေမယ့် Sort လုပ်မယ်ဆိုရင် Time Complexity က $O(m \cdot n \log n)$ (m က string အရေအတွက်၊ n က string တစ်ခုရဲ့ ပျမ်းမျှ အရှည်) ဖြစ်သွားပါမယ်။
ပိုမြန်တဲ့ နည်းလမ်းကတော့ Array Counting နည်းနဲ့ Group ဖွဲ့တာပါ။ `a-z` အထိ character ၂၆ လုံးအတွက် Array ဆောက်ပြီး ရေတွက်လိုက်မယ်ဆိုရင် Time Complexity ကို $O(m \cdot n)$ အထိ လျှော့ချလို့ ရပါတယ်။

```java
import java.util.*;

class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();

        for (String s : strs) {
            // character 26 လုံးအတွက် count မှတ်ရန် array
            char[] hash = new char[26];
            for (char c : s.toCharArray()) {
                hash[c - 'a']++;
            }

            // ယင်း Array ကို String အနေနဲ့ ပြောင်းပြီး Key အဖြစ်သုံးခြင်း
            String key = new String(hash);

            map.putIfAbsent(key, new ArrayList<>());
            map.get(key).add(s);
        }

        return new ArrayList<>(map.values());
    }
}
```

Top K Frequent Element

Given an integer array `nums` and an integer `k` , return the `k` most frequent elements. You may return the answer in **any order**.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Constraints:

- Your algorithm's time complexity must be better than $O(n \log n)$, where n is the array's size.

ဒီပုစ္ဆာမှာ element တွေ ဘယ်နှစ်ကြိမ် ပါလဲဆိုတာကို အရင်ဆုံး `HashMap` နဲ့ ရေတွက်လိုက်ပါမယ်။ အဲဒီနောက် အကြိမ်ရေ အများဆုံး k ခုကို ရှာရမှာပါ။ Constraints အရ $O(n \log n)$ ထက်မြန်ရမယ် ဆိုတော့ ပုံမှန် Sort လုပ်တဲ့နည်းကို သုံးလို့ မရပါဘူး။

ဒီအခါမှာ Bucket Sort သဘောတရားကို အသုံးပြုနိုင်ပါတယ်။ Array တစ်ခုတည်ဆောက်ပြီး အဲဒီ Array ရဲ့ Index ကို "ပါဝင်တဲ့အကြိမ်ရေ (Frequency)" အနေနဲ့ သတ်မှတ်ပါမယ်။ Array ရဲ့ Values ကတော့ အဲဒီ အကြိမ်ရေပြည့်တဲ့ နံပါတ်တွေရဲ့ List ဖြစ်ပါမယ်။ သုံးမယ့် Time Complexity ကတော့ $O(n)$ ပါ။ Space Complexity လည်း $O(n)$ ပါပဲ။

```java
import java.util.*;

class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> count = new HashMap<>();
        // တစ်ခုချင်းစီ ဘယ်နှစ်ကြိမ်ပါလဲ ရေတွက်ခြင်း
        for (int num : nums) {
            count.merge(num, 1, Integer::sum);
        }

        // ကြိမ်ရေ (Frequency) ကို Index အဖြစ်သုံးရန် Array တည်ဆောက်ခြင်း
        List<Integer>[] buckets = new List[nums.length + 1];
        for (int i = 0; i < buckets.length; i++) {
            buckets[i] = new ArrayList<>();
        }

        // Map ထဲက Data တွေကို ကြိမ်ရေနဲ့ ကိုက်ညီတဲ့ အခန်းထဲ သွားထည့်ခြင်း
        for (int key : count.keySet()) {
            buckets[count.get(key)].add(key);
        }

        // နောက်ဆုံးကနေ ပြန်ဖတ်ပြီး အကြိမ်ရေများတဲ့ k ခုကို ပြန်ပေးခြင်း
        int[] result = new int[k];
        int index = 0;
        for (int i = buckets.length - 1; i >= 0 && index < k; i--) {
            for (int num : buckets[i]) {
                result[index++] = num;
                if (index == k) return result;
            }
        }
        return result;
    }
}
```

Valid Sudoku

Determine if a `9x9` Sudoku board is valid.

Rules:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3x3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Sudoku board တစ်ခု မှန်ကန်မှုရှိမရှိ စစ်ဆေးဖို့အတွက် အခြေအနေ ၃ ခုကို စစ်ရပါမယ်။ လိုင်း (Row) ထဲမှာ ဂဏန်းတွေ ထပ်နေလား၊ တိုင် (Column) ထဲမှာ ထပ်နေလား၊ `3x3` အကွက်လေး (Sub-box) ထဲမှာရော ထပ်နေလား ဆိုတာပါ။

ဒီအတွက် `HashSet` ကို သုံးပြီး လွယ်လွယ်ကူကူ ဖြေရှင်းနိုင်ပါတယ်။ HashSet ထဲကစာသားကို သေချာဖွဲ့စည်းပေးလိုက်ပါမယ်။ ဥပမာ `"5 found in row 0"` , `"5 found in column 2"` , သို့မဟုတ် `"5 found in sub-box 0-0"` လို့ String လေးတွေ တည်ဆောက်ပြီး Set ထဲ ထည့်လိုက်ပါမယ်။ `HashSet.add()` က ပါပြီးသား (duplicate) ဆိုရင် `false` Return ပြန်ပေးတဲ့အတွက် အထပ်ပါနေတာကို အလွယ်တကူ သိနိုင်ပါတယ်။

```
import java.util.HashSet;
import java.util.Set;

class Solution {
    public boolean isValidSudoku(char[][] board) {
        Set<String> seen = new HashSet<>();

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                char current = board[i][j];
                if (current != '.') {
                    // String တွေ တည်ဆောက်ပြီး Set ထဲထည့်ရန် ကြိုးစားခြင်း
                    if (!seen.add(current + " found in row " + i) ||
                        !seen.add(current + " found in col " + j) ||
                        !seen.add(current + " found in sub-box " + (i / 3) + "-" + (j /
3))) {

                        return false;
                        // သတ်မှတ်ထားတဲ့ လိုင်း, တိုင်, သို့မဟုတ် အကွက်လေး ထဲမှာ အဆိုပါဂဏန်း ပါနေခဲ့
ပြီ ဆိုတာကို ပြသသည်
                    }
                }
            }
        }
        return true;
    }
}
```

Longest Consecutive Sequence

Given an unsorted array of integers `nums` , return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in $O(n)$ time.

Example 1:

```
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length
is 4.
```

ဆက်တိုက်ဖြစ်နေတဲ့ ဂဏန်းစဉ် (Sequence) အရှည်ဆုံးကို ရှာရမှာပါ။ ဥပမာ `1, 2, 3, 4` ပါရင် အရှည် ၄ ဆိုပြီး အဖြေထုတ်ပေးရမှာပါ။ Constraints မှာ $O(n)$ နဲ့ တွက်ရမယ် ဆိုတာကြောင့် Array ကို Sort လုပ်ရင် $O(n \log n)$ ဖြစ်သွားမှာမို့လို့ Sort လုပ်ခွင့် မရှိပါဘူး။

$O(n)$ နဲ့ ရဖို့အတွက် array ထဲက element တွေ အကုန်လုံးကို `HashSet` ထဲ အရင်ထည့်လိုက်ပါမယ်။ ပြီးရင် number တစ်ခုချင်းစီဟာ sequence တစ်ခုရဲ့ အစ (Start of Sequence) ဟုတ်မဟုတ် စစ်ဆေးပါမယ်။
ဘယ်လိုစစ်မလဲ ဆိုတော့ သက်ဆိုင်ရာနံပါတ် `num` ရဲ့ အရှေ့နံပါတ် `num - 1` က HashSet ထဲမှာ မရှိဘူး ဆိုရင် ဒါဟာ အစပဲ ဖြစ်ပါတယ်။ ကဲ အစကို ရပြီ ဆိုတာနဲ့ အဲဒီနံပါတ်ကနေ စပြီး `num + 1` , `num + 2` တွေ Set ထဲ ရှိမရှိ ဆက်တိုက် ရှာသွားလိုက်ယုံပါပဲ။

```java
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int longestConsecutive(int[] nums) {
        // ထပ်နေတာတွေ ဖယ်ပြီး အလွယ်တကူ စစ်နိုင်အောင် HashSet သုံးခြင်း
        Set<Integer> set = new HashSet<>();
        for (int num : nums) {
            set.add(num);
        }

        int longestStreak = 0;

        for (int num : set) {
            // Sequence ရဲ့ အစ ဟုတ်မဟုတ် စစ်ဆေးခြင်း
            if (!set.contains(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                // နောက်နံပါတ်တွေ ဆက်တိုက် ရှိမရှိ စစ်ဆေးခြင်း
                while (set.contains(currentNum + 1)) {
                    currentNum += 1;
                    currentStreak += 1;
                }

                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }
}
```

အခန်း ၅ - Two Pointers

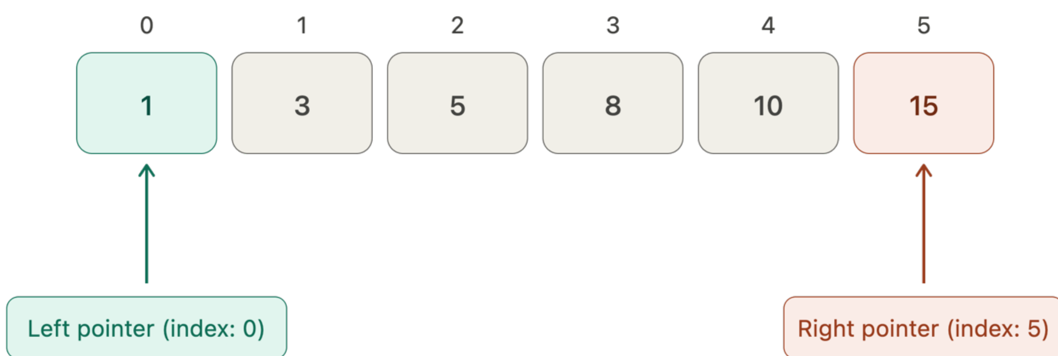
Two Pointers ဆိုတာ သီးသန့် Data Structure တစ်ခု မဟုတ်ပါဘူး။ Array, String နဲ့ Linked List တွေမှာ တွေ့ရလေ့ရှိတဲ့ ပြဿနာတွေကို ဖြေရှင်းရာမှာ အသုံးပြုတဲ့ **နည်းလမ်း (Algorithm Technique)** တစ်ခု ဖြစ်ပါတယ်။ အထူးသဖြင့် ပုံမှန် Nested Loop နဲ့ $O(n^2)$ ကြာနိုင်တဲ့ ရှာဖွေမှု ပြဿနာတွေကို $O(n)$ သို့မဟုတ် $O(n \log n)$ အထိ လျှော့ချနိုင်တဲ့ နေရာမှာ အလွန် အသုံးဝင်ပါတယ်။

Two Pointers အလုပ်လုပ်ပုံကွဲများ

အဓိကအားဖြင့် Pointer ၂ ခုကို သုံးပြီး Data ကို ဖတ်ရှုတာ ဖြစ်ပါတယ်။ အသုံးအများဆုံး ပုံစံ ၂ မျိုး ရှိပါတယ်။

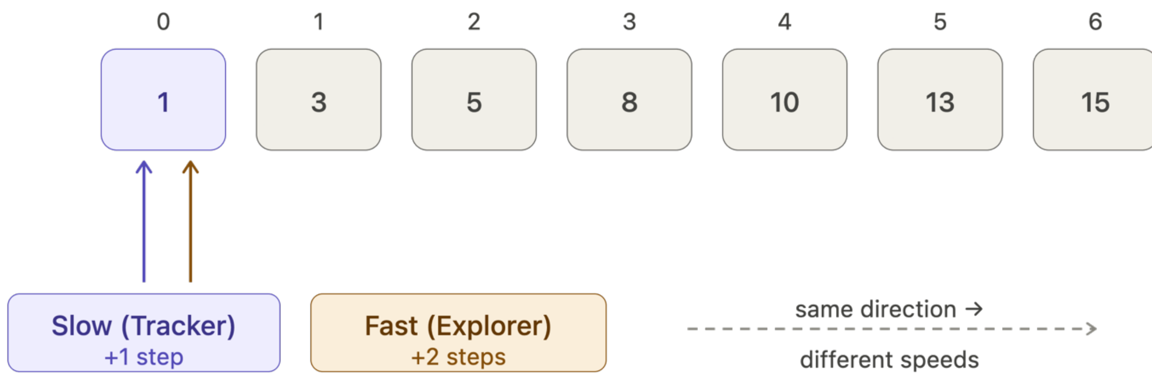
၁။ Opposite Direction (ဆန့်ကျင်ဘက်သို့ သွားခြင်း)

Pointer တစ်ခုကို အစ (`left = 0`) မှာ ထားပြီး၊ နောက်တစ်ခုကို အဆုံး (`right = length - 1`) မှာ ထားပါတယ်။ ပြီးရင် ပြဿနာရဲ့ အခြေအနေပေါ် မူတည်ပြီး အချင်းချင်း ဆုံတဲ့အထိ ရှေ့တိုး/နောက်ဆုတ် လုပ်သွားတဲ့ နည်းပါ။ ဒီနည်းလမ်းကို အများအားဖြင့် **Sorted (စီစဉ်ပြီးသား) Array** တွေမှာ အများဆုံး သုံးပါတယ်။



၂။ Same Direction (ဦးတည်ရာ တူညီခြင်း - Fast & Slow Pointers)

Pointer ၂ ခုလုံး အစကနေပဲ စပါတယ်။ ဒါပေမယ့် တစ်ခုက အရှေ့ကနေ မြန်မြန်သွားပြီး (Fast pointer / Explorer) နောက်တစ်ခုက နှေးနှေးသွားတဲ့ (Slow pointer / Tracker) နည်းပါ။ မြန်တဲ့ကောင်က အရှေ့ကနေ လိုအပ်တဲ့ အခြေအနေကို လိုက်ရှာပေးပြီး၊ နှေးတဲ့ကောင်က လိုချင်တဲ့ နေရာရောက်တဲ့အခါ အလုပ်လုပ်တဲ့ သဘောမျိုးပါ။ တနည်းအားဖြင့် နောက်ပိုင်းတွေ့ရမည့် Sliding Window Technique မှာလည်း Same Direction သွားတဲ့ Two Pointers သဘောတရားကိုပဲ သုံးပါတယ်။



ဘယ်အချိန်မှာ Two Pointers သုံးမလဲ

အောက်ပါ လက္ခဏာတွေ တွေ့ရင် Two Pointers ကို စဉ်းစားသင့်ပါတယ်။

- Array သို့မဟုတ် String ထဲက **အတွဲ (pair)** ဒါမှမဟုတ် **အစု (triplet)** ကို ရှာရတဲ့အခါ
- Data က **Sorted (စီထားပြီးသား)** ဖြစ်နေတဲ့အခါ — အစ/အဆုံးကနေ ချဉ်းကပ်လို့ ရတယ်
- Nested loop $O(n^2)$ နဲ့ ဖြေလို့ရပေမယ့် ပိုမြန်အောင် လုပ်ချင်တဲ့အခါ
- **Extra space မယူရ** ($O(1)$) လို့ ကန့်သတ်ထားတဲ့အခါ — HashMap သုံးလို့ မရတော့လို့

ပိုပြီး real-world feeling ရအောင် ဒီလိုချဲ့ရေးနိုင်ပါတယ်။

Real-world မှာ ဘယ်လိုသုံးလဲ

Two Pointers က Interview အတွက် Algorithm တစ်ခုလို့ပဲ ထင်ရပေမယ့် Production System တွေမှာ နေ့စဉ် သုံးနေတဲ့ Pattern တစ်ခုပါ။ အထူးသဖြင့် **sorted data** တွေ၊ **streaming data** တွေ၊ **large dataset** တွေကို memory များများ မသုံးဘဲ process လုပ်တဲ့အခါ အသုံးဝင်တယ်။

- **Sorted list နှစ်ခု ပေါင်းခြင်း (Merge)** — Database query နှစ်ခုက timestamp အလိုက် စီထားပြီးသား result တွေကို activity feed တစ်ခုအဖြစ် ပေါင်းတဲ့အခါ pointer နှစ်ခုနဲ့ တစ်ကြိမ်စီ ရှေ့တိုးသွားရုံပဲ လိုတယ်။ Data တစ်ခုစီကို တစ်ခါပဲ ဖတ်ရတဲ့အတွက် $O(n + m)$ နဲ့ အလုပ်ပြီးတယ်။ Merge Sort ရဲ့ merge step ကလည်း ဒီအယူအဆပဲ ဖြစ်တယ်။

- **Sorted list နှစ်ခု နှိုင်းယှဉ်ခြင်း** — System version နှစ်ခုကြား ပြောင်းလဲသွားတဲ့ record တွေကို ရှာခြင်း၊ user ID နှစ်စုရဲ့ intersection / difference ထုတ်ခြင်း၊ inventory list နှစ်ခုကို reconcile လုပ်ခြင်း စတဲ့နေရာတွေမှာ အသုံးများတယ်။ Hash Table မသုံးဘဲ memory သက်သာစွာနဲ့ scan တစ်ကြိမ်တည်း လုပ်နိုင်တယ်။

- **Duplicate ဖယ်ခြင်း (In-place Deduplication)** — Log file, user list, transaction record စတဲ့ sorted data တွေထဲက duplicate တွေကို array အသစ်မဆောက်ဘဲ ဖယ်ရှားနိုင်တယ်။ Slow pointer က valid data ရေးမယ့်နေရာကို ညွှန်ပြီး fast pointer က data ကို scan လုပ်တဲ့ pattern ကို database engine တွေ၊ ETL pipeline တွေမှာ မကြာခဏ တွေ့ရတယ်။

- **Sliding Window Analysis** — Time-series data, monitoring metric, stock price, sensor data တွေမှာ အချိန်ကာလတစ်ခုအတွင်း average, maximum, unique count စတာတွေကို တွက်ချက်တဲ့အခါ left/right pointer နှစ်ခုနဲ့ window ကို ရွှေ့သွားတယ်။ Log analytics နဲ့ observability platform တွေမှာ အသုံးများတဲ့ technique ဖြစ်တယ်။
- **Streaming Data Processing** — Kafka, Kinesis, RabbitMQ ကဲ့သို့ data stream တွေကို ဖတ်တဲ့အခါ record အားလုံး memory ထဲ မတင်နိုင်ဘူး။ Pointer နှစ်ခု သို့မဟုတ် iterator နှစ်ခုနဲ့ stream နှစ်ခုကို တစ်ပြိုင်နက်တည်း ဖတ်ပြီး compare, merge, join လုပ်ကြတယ်။
- **Large File Comparison** — Git, diff tool, synchronization software တွေက file version နှစ်ခုကို line-by-line နှိုင်းယှဉ်တဲ့အခါ pointer နှစ်ခုကို အသုံးပြုတယ်။ File တစ်ခုလုံး memory ထဲ တင်စရာမလိုဘဲ sequential scan လုပ်နိုင်လို့ performance ကောင်းတယ်။
- **Network & Log Processing** — Access log, event log, audit log တွေက အချိန်အစဉ်အတိုင်း စီထားလေ့ရှိတယ်။ Service A ရဲ့ log နဲ့ Service B ရဲ့ log ကို correlation လုပ်ပြီး request flow ကို ချိတ်ဆက်ကြည့်တဲ့အခါ timestamp အလိုက် pointer နှစ်ခုနဲ့ လိုက်ဖတ်တာက အထိရောက်ဆုံး နည်းလမ်းတစ်ခု ဖြစ်တယ်။

Two Pointers ရဲ့ အဓိက အားသာချက်က **nested loop ကို linear scan အဖြစ် ပြောင်းပေးနိုင်ခြင်း** ဖြစ်တယ်။ Data က sorted ဖြစ်နေပြီဆိုရင် Hash Map မလိုဘဲ memory သက်သာစွာနဲ့ $O(n)$ သို့မဟုတ် $O(n+m)$ အတွင်း ပြဿနာအများစုကို ဖြေရှင်းနိုင်တယ်။

ဒီ pattern တွေက အောက်က ပုစ္ဆာတွေမှာ ပြန်တွေ့ရမှာပါ။

Questions

Two Pointers ကို ပိုမို နားလည်သွားစေဖို့ လက်တွေ့ အသုံးချရမယ့် လေ့ကျင့်ခန်း ပုစ္ဆာ တချို့ကို ကြည့်ရအောင်။

Reverse String

Character array `s` တစ်ခုကို နေရာပို မယူဘဲ (in-place) ပြောင်းပြန် လှန်ပါ။

Example:

```
Input:  s = ['h','e','l','l','o']
Output: ['o','l','l','e','h']
```

ဒါက Two Pointers (Opposite Direction) ရဲ့ အရိုးရှင်းဆုံး ဥပမာပါ။ Left ကို အစ၊ Right ကို အဆုံးမှာ ထားပြီး အချင်းချင်း လဲ (swap) လိုက်တယ်။ ပြီးရင် နှစ်ခုလုံး အလယ်ဆီ တိုး/ဆုတ် သွားလိုက်ရုံပါပဲ။ Time $O(n)$၊ Space $O(1)$ ဖြစ်ပါတယ်။

```
class Solution {
    public void reverseString(char[] s) {
        int left = 0;
```

```
        int right = s.length - 1;

        while (left < right) {
            char temp = s[left];
            s[left] = s[right];
            s[right] = temp;

            left++;
            right--;
        }
    }
}
```

Remove Duplicates from Sorted Array

(အငယ်မှအကြီး) စီထားပြီးသား array `nums` ထဲက ထပ်နေတဲ့ ဂဏန်းတွေကို **နေရာပို မယူဘဲ (in-place)** ဖယ်ပြီး၊ ကျန်တဲ့ unique element အရေအတွက်ကို ပြန်ပေးပါ။

Example:

```
Input:  nums = [1,1,2,2,3]
Output: 3,  nums = [1,2,3,...]
```

ဒါက **Same Direction (Slow & Fast)** ရဲ့ classic ဥပမာပါ။ Array က စီထားပြီးသား ဖြစ်တဲ့အတွက် ထပ်နေတဲ့ ဂဏန်းတွေဟာ ဘေးချင်းကပ်လျက် ရှိနေမှာပါ။

- **Fast pointer** က array တစ်ခုလုံးကို ဖြတ်သွားပြီး element တိုင်းကို ကြည့်တယ်။
- **Slow pointer** က unique element တွေကို သိမ်းမယ့် နေရာကို ထိန်းတယ်။ Fast က အသစ်တစ်ခု (ရှေ့ကနဲ့ မတူတာ) တွေ့မှ slow ကို တိုးပြီး အဲဒီနေရာမှာ ရေးချတယ်။

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int slow = 0; // unique element တွေ ဆုံးတဲ့ နေရာ

        for (int fast = 1; fast < nums.length; fast++) {
            // ရှေ့က element နဲ့ မတူမှ အသစ်လို့ သတ်မှတ်သည်
            if (nums[fast] != nums[slow]) {
                slow++;
                nums[slow] = nums[fast];
            }
        }

        return slow + 1; // အရေအတွက် = နောက်ဆုံး index + 1
    }
}
```

Valid Palindrome

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string `s`, return `true` if it is a **palindrome**, or `false` otherwise.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

Palindrome ဆိုတာ ရှေ့ကဖတ်ဖတ်၊ နောက်ကဖတ်ဖတ် အတူတူပဲ ဖြစ်နေတဲ့ စာသားကို ခေါ်တာပါ။ ဒီပုစ္ဆာမှာ စာလုံးအကြီးအသေး မခွဲခြားတဲ့အပြင်၊ စာသား (a-z) နဲ့ ဂဏန်း (0-9) ကလွဲပြီး ကျန်တဲ့ သင်္ကေတ (Symbols) တွေ အားလုံးကို ဖယ်ထုတ်/ကျော်သွားပြီး စစ်ရမှာပါ။

ဒီပြဿနာကို ဖြေရှင်းဖို့ Opposite Direction သုံးတဲ့ Two Pointers နည်းလမ်းက အကောင်းဆုံးပါ။ Left pointer ကို အစမှာထားပြီး Right pointer ကို အဆုံးမှာ ထားပါမယ်။ မလိုအပ်တဲ့ သင်္ကေတတွေ တွေ့ရင် ကျော်သွားပါမယ်။ နှစ်ခုလုံးက စာလုံးစစ်စစ် ဖြစ်တဲ့အခါ တူ/မတူ စစ်ဆေးပါမယ်။

> $O(n)$ time complexity, space complexity $O(1)$ ဖြစ်ရပါမည်။

```
class Solution {
    public boolean isPalindrome(String s) {
        int left = 0;
        int right = s.length() - 1;

        while (left < right) {
            // ဘယ်ဘက် pointer က စာလုံး/ဂဏန်း မဟုတ်ရင် ကျော်သွားမည်
            while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
                left++;
            }
            // ညာဘက် pointer က စာလုံး/ဂဏန်း မဟုတ်ရင် ကျော်သွားမည်
            while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
                right--;
            }

            // အကြီးအသေး မခွဲဘဲ တိုက်စစ်မည်
            if (Character.toLowerCase(s.charAt(left)) !=
Character.toLowerCase(s.charAt(right))) {
                return false;
            }

            // တူရင် ရှေ့ဆက်သွားမည်
            left++;
            right--;
```

```
        }

        return true;
    }
}
```

Two Sum II - Input Array Is Sorted

Given a **1-indexed** array of integers `numbers` that is already **sorted in non-decreasing order**, find two numbers such that they add up to a specific `target` number.

Return the indices of the two numbers, `index1` and `index2`, added by one as an integer array `[index1, index2]` of length 2.

Your solution must use only constant extra space.

Example 1:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].
```

ဒီပုစ္ဆာက Array & Hash Table အခန်း က Two Sum နဲ့ တူပါတယ်။ ဒါပေမယ့် ကွာခြားချက်တွေက—

၁။ Array က (အငယ်မှအကြီး) စီထားပြီးသား ဖြစ်နေတာ။

၂။ နေရာပို (Extra Space) လုံးဝ မယူရဘူး ($O(1)$ Space) ဆိုတာပါ။ Memory အသစ် မယူရတဲ့ အတွက် HashMap ကို သုံးလို့ မရတော့ပါဘူး။

Array က စီထားပြီးသား ဖြစ်တဲ့အတွက် Two Pointers (Opposite Direction) ကို သုံးပြီး အလွယ်တကူ ဖြေရှင်းနိုင်ပါတယ်။ အစ (အငယ်ဆုံးတန်ဖိုး) နဲ့ အဆုံး (အကြီးဆုံးတန်ဖိုး) ကို ပေါင်းကြည့်ပါမယ်။

- ပေါင်းလဒ်က `target` ထက် ကြီးနေရင်၊ အဖြေကို သေးအောင် လုပ်ဖို့ လိုတဲ့အတွက် Right Pointer ကို တစ်ခန်း လျှော့ပါမယ်။
- ပေါင်းလဒ်က `target` ထက် ငယ်နေရင်၊ အဖြေကို ကြီးအောင် လုပ်ဖို့ လိုတဲ့အတွက် Left Pointer ကို တစ်ခန်း တိုးပါမယ်။

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int left = 0;
        int right = numbers.length - 1;

        while (left < right) {
            int currentSum = numbers[left] + numbers[right];

            if (currentSum == target) {
                // ပြဿနာက 1-indexed (index 1 ကနေစသည်) လို့ ဆိုထားသည့်အတွက် 1 ပေါင်းပေးရ
သည်
                return new int[] { left + 1, right + 1 };
            } else if (currentSum > target) {
                // ပေါင်းလဒ် များနေလျှင် တန်ဖိုးနည်းသွားစေရန် right ကို လျှော့သည်
```

```
                right--;
            } else {
                // ပေါင်းလဒ် နည်းနေလျှင် တန်ဖိုးများလာစေရန် left ကို တိုးသည်
                left++;
            }
        }

        return new int[] {};
    }
}
```

3Sum

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
```

ဒီပုစ္ဆာက ဂဏန်း ၃ လုံး ပေါင်းရင် 0 ရမယ့် အစုလေးတွေကို ရှာခိုင်းတာပါ။ ပုံမှန်ဆို Loop ၃ ထပ် ပတ်ရမှာမို့လို့ $O(n^3)$ ဖြစ်သွားပါမယ်။
ပိုကောင်းတဲ့ ဖြေရှင်းနည်းကတော့ Array ကို အရင်ဆုံး Sort လုပ်လိုက်ပါမယ်။ Sort လုပ်လိုက်တဲ့အတွက် တူညီတဲ့ ဂဏန်းတွေဟာ ဘေးကပ်လျက် ရောက်သွားမှာဖြစ်ပြီး၊ Duplicate ဖြစ်နေတာတွေကို အလွယ်တကူ ကျော်သွား (Skip) လို့ ရသွားပါမယ်။
ပြီးရင် ဂဏန်းတစ်လုံးကို အသေထား (Loop ပတ်) ပြီး၊ ကျန်တဲ့ အပိုင်းကို "Two Sum II" မှာ သုံးခဲ့တဲ့ Two Pointers နည်းနဲ့ ဆက်ရှာသွားယုံပါပဲ။ Time Complexity $O(n^2)$ နဲ့ ရနိုင်ပါတယ်။

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        // Two pointer သုံးရန်နှင့် Duplicate များ အလွယ်တကူဖယ်နိုင်ရန် Sort အရင်လုပ်သည်
        Arrays.sort(nums);

        for (int i = 0; i < nums.length; i++) {
            // Duplicate ဖြစ်နေလျှင် ကျော်သွားမည်
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }

            int left = i + 1;
            int right = nums.length - 1;

            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
```

```
                if (sum > 0) {
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    res.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    left++;
                    // Duplicate အတွဲများ ထပ်မပါစေရန်
                    while (left < right && nums[left] == nums[left - 1]) {
                        left++;
                    }
                }
            }
        }
        return res;
    }
}
```

Container With Most Water

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the ith line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Example 1:

```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this
case, the max area of water (blue section) the container can contain is 49.
```

ဒီပုစ္ဆာမှာ တိုင်လေးတွေကြားမှာ ရေ အများဆုံး လှောင်နိုင်မယ့် အကျယ်ပြန့်ဆုံး (Area အများဆုံး) ဧရိယာ ကို ရှာရမှာပါ။
ရေလှောင်နိုင်တဲ့ ပမာဏ = "အကျယ် (Width) × ရေလှောင်နိုင်တဲ့ အမြင့်" ဖြစ်ပါတယ်။
ရေလှောင်နိုင်တဲ့ အမြင့် ဆိုတာ တိုင်နှစ်ခုစလုံးမှာ နိမ့်တဲ့တိုင် ရဲ့ အမြင့်ကို ယူရမှာပါ။ နိမ့်တဲ့အမြင့်ထက် ကျော်ရင် ရေတွေ လျှံကျသွားမှာ ဖြစ်လို့ပါ။

Two Pointers ကို အစနဲ့ အဆုံးမှာ ချထားပါမယ်။ အစနဲ့ အဆုံး ဆိုတော့ အကျယ် (Width) အကြီးဆုံး အခြေအနေကနေ စတာပါ။
ရေဆိုတာ နိမ့်တဲ့တိုင်ကိုပဲ အခြေခံတာ ဖြစ်တဲ့အတွက်၊ နိမ့်နေတဲ့ တိုင်ဘက်က Pointer ကို ရွှေ့ပြီး ပိုမြင့်မယ့်တိုင်များ တွေ့မလား ဆိုတာကို ဆက်ရှာပါမယ်။

```
class Solution {
    public int maxArea(int[] height) {
        int left = 0;
        int right = height.length - 1;
        int maxArea = 0;
```

```
        while (left < right) {
            // ရေလှောင်နိုင်သည့် အမြင့်သည် အနိမ့်ဆုံးတိုင်ပေါ် မူတည်သည်
            int minHeight = Math.min(height[left], height[right]);
            // အကျယ်
            int width = right - left;
            // ဧရိယာ (Area)
            int area = minHeight * width;
            maxArea = Math.max(maxArea, area);

            // ပိုမြင့်သည့် တိုင်ကို ရှာရန် နိမ့်သောဘက်မှ Pointer ကို ရွှေ့သည်
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }
}
```

Trapping Rain Water

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being
trapped.
```

ဒီပုစ္ဆာက "Container With Most Water" နဲ့ ဆင်ပေမယ့် ပိုခက်ပါတယ်။ လမ်းတွေ/မြေပြင်တွေ ခွက်နေတဲ့ အခါ ရေဘယ်လောက် အိုင်နိုင်လဲ ဆိုတာကို တွက်ရမှာပါ။
တိုင်တစ်တိုင်ရဲ့ အပေါ်မှာ ရေ ဘယ်လောက်တင်မလဲ ဆိုတာဟာ သူ့ရဲ့ **ဘယ်ဘက်မှာ ရှိတဲ့ အမြင့်ဆုံးတိုင် (MaxLeft)** နဲ့ **ညာဘက်မှာ ရှိတဲ့ အမြင့်ဆုံးတိုင် (MaxRight)** တွေထဲက နိမ့်တဲ့အမြင့် `min(MaxLeft, MaxRight)` ကနေ လက်ရှိ တိုင်အမြင့် ကို နှုတ်လိုက်တဲ့ ပမာဏပါပဲ။

နေရာပိုမယူဘဲ အချိန် $O(n)$ ၊ Space $O(1)$ နဲ့ တွက်ဖို့အတွက် Two Pointers ကို သုံးနိုင်ပါတယ်။
Left နဲ့ Right ကို အစဆုံး ချထားပြီး လမ်းလျှောက်ပါမယ်။ MaxLeft နဲ့ MaxRight ကို မှတ်သွားပါမယ်။ အကယ်၍ ကိုယ့်ရဲ့ MaxLeft က MaxRight ထက် ငယ်နေရင် ဘယ်ဘက်ကရေအိုင်ဖို့ သေချာနေပြီ ဖြစ်တဲ့အတွက် (ရေလျှံမသွားနိုင်ပါ) ဘယ်ဘက်ကို တွက်ပြီး တိုးပါတယ်။ ပြောင်းပြန်ဆိုရင်တော့ ညာဘက်ကို တွက်ပြီး လျှော့ပါမယ်။

```
class Solution {
    public int trap(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }
```

```
        int left = 0;
        int right = height.length - 1;

        int maxLeft = height[left];
        int maxRight = height[right];

        int totalWater = 0;

        while (left < right) {
            // maxLeft က ပိုငယ်နေလျှင် ဘယ်ဘက်ရှိ ရေကို တွက်မည်
            if (maxLeft < maxRight) {
                left++;
                maxLeft = Math.max(maxLeft, height[left]);
                totalWater += maxLeft - height[left];
            }
            // maxRight က ပိုငယ်နေလျှင် ညာဘက်ရှိ ရေကို တွက်မည်
            else {
                right--;
                maxRight = Math.max(maxRight, height[right]);
                totalWater += maxRight - height[right];
            }
        }

        return totalWater;
    }
}
```

အခန်း ၆ - Sliding Window

Sliding Window (ဆလိုက်ဒင်းဝင်းဒိုး) ဆိုတာ Array ဒါမှမဟုတ် String တွေပေါ်မှာ Subarray သို့မဟုတ် Substring တွေနဲ့ ပတ်သက်ပြီး ရှာဖွေတွက်ချက်ရတဲ့ ပုစ္ဆာတွေကို အလွန် လျင်မြန်စွာ ဖြေရှင်းနိုင်တဲ့ **နည်းလမ်း (Algorithm Technique)** တစ်ခု ဖြစ်ပါတယ်။

ပုံမှန်အားဖြင့် sub-elements တွေကို Nested Loops (Loop ပတ်တာ နှစ်ထပ်) နဲ့ တွက်ချက်မယ်ဆိုရင် Time Complexity ဟာ $O(n^2)$ ကြာမြင့်တတ်ပေမယ့် Sliding Window ကို စနစ်တကျ အသုံးပြုမယ်ဆိုရင် အချိန်ကို $O(n)$ သို့မဟုတ် Linear Time အထိ သိသိသာသာ လျှော့ချနိုင်ပါတယ်။

Sliding Window ၏ အဓိက အယူအဆ (Core Intuition)

Sliding Window ကို နားလည်ဖို့ အလွယ်ဆုံးနည်းက **အရွယ်အစားတူတဲ့ အပိုင်းလေးတွေကို ဘယ်လိုမြန်မြန် တွက်မလဲ** ဆိုတာကနေ စပါမယ်။

ဥပမာ `nums = [5, 2, -1, 0, 3]` ဖြစ်ပြီး၊ တစ်ခါကြည့်မယ့် အပိုင်းအရွယ်အစား `k = 3` ဖြစ်တယ်ဆိုပါစို့။ မေးခွန်းက **ဆက်တိုက်ရှိတဲ့ ဂဏန်း ၃ လုံးအုပ်စုတွေထဲမှာ ပေါင်းလဒ်အများဆုံးက ဘယ်လောက်လဲ** ဆိုတာပါ။

အရင်ဆုံး ဖြစ်နိုင်တဲ့ အုပ်စုတွေကို ရှင်းရှင်းလင်းလင်း ကြည့်ပါ။

အုပ်စု	ပေါင်းလဒ်
`[5, 2, -1]`	`5 + 2 + (-1) = 6`
`[2, -1, 0]`	`2 + (-1) + 0 = 1`
`[-1, 0, 3]`	`(-1) + 0 + 3 = 2`

ရိုးရိုးနည်းလမ်းနဲ့ဆိုရင် အုပ်စုတစ်ခုချင်းစီကို အစကနေ ပြန်ပေါင်းရပါတယ်။ ဒါကြောင့် `[5, 2, -1]` ပြီးလို့ `[2, -1, 0]` ကိုတွက်တဲ့အခါ `2` နဲ့ `-1` ကို ထပ်ပေါင်းနေရပါတယ်။

Sliding Window နည်းလမ်းကတော့ အဲ့ဒီ ထပ်တွက်နေတဲ့အလုပ်ကို လျှော့တာပါ။ ပထမအုပ်စုရဲ့ ပေါင်းလဒ်ကို `6` လို့ သိပြီးသားဆိုရင် နောက်အုပ်စုကို အစကနေ ပြန်မပေါင်းတော့ပါဘူး။

အဆင့်	လုပ်ဆောင်ချက်	Window Sum
စတင်	`[5, 2, -1]` ကို ပေါင်းသည်	`6`
တစ်လှမ်းရွေ့	ဘယ်ဘက်ဆုံး `5` ထွက်၊ ညာဘက်အသစ် `0` ဝင်	`6 - 5 + 0 = 1`
နောက်တစ်လှမ်းရွေ့	ဘယ်ဘက်ဆုံး `2` ထွက်၊ ညာဘက်အသစ် `3` ဝင်	`1 - 2 + 3 = 2`

ဒီလို **အရင် window ရဲ့ အဖြေကို သုံးပြီး နောက် window ရဲ့ အဖြေကို တွက်တာ** ကို Sliding Window လို့ ခေါ်ပါတယ်။

```
nextWindowSum = currentWindowSum − ထွက်သွားသောတန်ဖိုး + ဝင်လာသောတန်ဖိုး
```

အုပ်စုတိုင်းကို အစကနေ ပြန်ပေါင်းရင် $O(n \times k)$ ကြာနိုင်ပေမယ့် Sliding Window နဲ့ဆိုရင် ညာဘက်သို့ တစ်ခါချင်း ရွှေ့သွားရုံဖြစ်လို့ $O(n)$ အထိ လျှော့နိုင်ပါတယ်။

ဝင်းဒိုး အမျိုးအစားများ (Types of Sliding Windows)

Sliding Window ပြဿနာတွေကို အဓိကအားဖြင့် အမျိုးအစား ၂ မျိုး ခွဲခြားနိုင်ပါတယ်။

၁။ Fixed-size Window (ပုံသေအရွယ်အစား ဝင်းဒိုး)

ဝင်းဒိုးရဲ့ အကျယ်ဟာ ပုစ္ဆာမှာ ပေးထားတဲ့အတိုင်း အမြဲတမ်း ပုံသေဖြစ်နေတတ်ပါတယ်။ (ဥပမာ- size က k လို့ ပေးထားခြင်း)။

- **လုပ်ဆောင်ပုံ:** ပထမဆုံး size k ရှိတဲ့ window တစ်ခု ဆောက်ပါ။ ပြီးရင် ညာဘက်အစွန်းကို တစ်ဆင့်တိုးပြီး ဘယ်ဘက်အစွန်းကိုပါ တစ်ဆင့်လိုက်တိုးကာ window အရွယ်အစား k အတိုင်း မပြောင်းလဲဘဲ ညာဘက်သို့ ဆလိုက်တိုက် ရွှေ့သွားပါ။

၂။ Dynamic-size Window (ပြောင်းလဲနိုင်သော ဝင်းဒိုး)

ဝင်းဒိုးရဲ့ အရွယ်အစားက သတ်မှတ်ထားတဲ့ အခြေအနေပေါ်မူတည်ပြီး အတိုးအလျှော့ ရှိပါတယ်။

- **လုပ်ဆောင်ပုံ:** `left` နှင့် `right` pointer နှစ်ခုကို သုံးပြီး `right` ကို ရွှေ့ကာ ဝင်းဒိုးကို ချဲ့သွားပါ။ အကယ်၍ သတ်မှတ်ချက် ကျော်လွန်သွားတဲ့အခါ `left` ကို ညာဘက်သို့ ရွှေ့ပြီး ဝင်းဒိုးကို ပြန်ကျဉ်းစေပါတယ်။ Window မှန်ကန်နေတဲ့အချိန်တိုင်း အဖြေကို update လုပ်သွားပါတယ်။

Two Pointers နဲ့ Sliding Window ဘာကွာလဲ

Sliding Window က Two Pointers (Same Direction) ရဲ့ အထူးပုံစံ တစ်ခုပါ။ နှစ်ခုလုံး `left` / `right` pointer သုံးပေမယ့် ရည်ရွယ်ချက် ကွဲပါတယ်။

	Two Pointers	Sliding Window
အာရုံစိုက်တာ	pointer **၂ ခုရဲ့ နေရာ** (ဥပမာ- အတွဲ၊ palindrome)	pointer ၂ ခုကြားက **အပိုင်း (window) တစ်ခုလုံး**
ပုံမှန် data	Sorted array များတွင် အများဆုံး	Sorted ဖြစ်ရန် မလို
တွက်ချက်မှု	pointer ၂ ခုနေရာက တန်ဖိုးကိုသာ ကြည့်လေ့ရှိ	window ထဲက sum / count / frequency ကို ထိန်းသိမ်း
ရှာဖွေပုံ	တစ်ခါတည်း အဖြေ ဆုံတတ်	window ကို **ချဲ့ (expand) / ကျဉ်း (shrink)** လုပ်ရင်း

		အဖြေ update

အလွယ်မှတ်ရင် — **pointer ၂ ခုကြားက အပိုင်းတစ်ခုလုံးကို စဉ်းစားဖို့ လိုရင် Sliding Window၊** pointer ၂ ခုရဲ့ နေရာကသာ အရေးကြီးရင် Two Pointers ပါ။

Real-world မှာ ဘယ်လိုသုံးလဲ

Sliding Window က "**ဆက်တိုက် N ခု**" သို့မဟုတ် "**လတ်တလော အချိန်ပိုင်း**" ပြဿနာတွေမှာ နေ့စဉ် တွေ့ရပါတယ်။

- **Rate limiting** — user တစ်ယောက် "၁ မိနစ်အတွင်း request ဘယ်နှစ်ကြိမ်" လုပ်လဲ ရေတွက်ဖို့ — အချိန် window တစ်ခုထဲက request count ကို ထိန်းတယ်။
- **လတ်တလော metrics** — "နောက်ဆုံး ၇ ရက်အတွင်း visitor အများဆုံး", "နောက်ဆုံး transaction ၁၀ ခု ပျမ်းမျှ" — fixed window။
- **Streaming / log analytics** — moving average၊ time-series ရဲ့ window aggregate။
- **Longest active session** — condition မပျက်သရွေ့ window ချဲ့သွားတဲ့ dynamic window။

ဒီ pattern တွေ အကုန်လုံးက အောက်က ပုစ္ဆာတွေနဲ့ သဘောတရားချင်း တူပါတယ်။

Questions

Sliding Window ကို ကျွမ်းကျင်စွာ အသုံးချတတ်စေဖို့ အမေးအများဆုံး မေးခွန်းပုံစံတွေကို တစ်ဆင့်ချင်း လေ့လာကြည့်ရအောင်။ အလွယ်ဆုံး Fixed-size window ကနေ စပါမယ်။

၁။ Maximum Sum Subarray of Size K

ဂဏန်း array `nums` တစ်ခုနဲ့ window size `k` ပေးထားတယ်။ ဆက်တိုက်ရှိတဲ့ ဂဏန်း `k` လုံးအုပ်စုတွေထဲက **ပေါင်းလဒ်အများဆုံးကို** ရှာပါ။

Example:

```
Input:  nums = [2, 1, 5, 1, 3, 2], k = 3
Output: 9
Explanation: [5, 1, 3] ၏ ပေါင်းလဒ် 9 သည် အများဆုံး ဖြစ်သည်။
```

ရှင်းလင်းချက်

ဒါက အပေါ်က Core Intuition မှာ ပြောခဲ့တဲ့ **Fixed-size window** အတိအကျပါ။ window တိုင်းကို အစကနေ ပြန်မပေါင်းဘဲ — **ထွက်သွားတဲ့တန်ဖိုး နှုတ်၊ ဝင်လာတဲ့တန်ဖိုး ပေါင်း** လုပ်ရုံပါ။ ဒါကြောင့် $O(n)$ ဖြစ်တယ်။

> **Time Complexity:** $O(n)$ — array တစ်ပတ်ပဲ ပတ်ရသည်။
> **Space Complexity:** $O(1)$

```java
class Solution {
    public int maxSumSubarray(int[] nums, int k) {
        int windowSum = 0;
        // ပထမဆုံး window k လုံးကို ပေါင်းသည်
        for (int i = 0; i < k; i++) {
            windowSum += nums[i];
        }

        int maxSum = windowSum;
        // window ကို တစ်လှမ်းချင်း ရွှေ့သည် – အဟောင်းနှုတ်၊ အသစ်ပေါင်း
        for (int right = k; right < nums.length; right++) {
            windowSum += nums[right] - nums[right - k];
            maxSum = Math.max(maxSum, windowSum);
        }

        return maxSum;
    }
}
```

၂။ Best Time to Buy and Sell Stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0` .

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before
you sell.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

ရှင်းလင်းချက်

ရှယ်ယာ (Stock) ဈေးနှုန်းတွေကို နေ့အလိုက် ပေးထားတဲ့ Array တစ်ခု ဖြစ်ပါတယ်။ အကျိုးအမြတ်အများဆုံး (Maximum Profit) ရအောင် ဘယ်နေ့မှာ ဝယ်ပြီး ဘယ်နေ့မှာ ပြန်ရောင်းရမလဲဆိုတာ ရှာရမှာပါ။ စည်းကမ်းချက်ကတော့ မရောင်းခင် အရင်ဝယ်ရမှာ ဖြစ်လို့ နောက်နေ့ဈေးနှုန်းထဲကနေ အရင်နေ့ဈေးနှုန်းကိုပဲ နှုတ်လို့ ရပါမယ်။

ဒီပုစ္ဆာကို Sliding Window / Two Pointers သဘောတရားနဲ့ စဉ်းစားနိုင်ပါတယ်။

- `left` pointer က ဝယ်တဲ့နေ့ (Buy Day) ကို ညွှန်ပြပြီး၊ `right` pointer က ရောင်းတဲ့နေ့ (Sell Day) ကို ညွှန်ပြပါမယ်။
- အမြဲတမ်း `prices[left]` (ဝယ်ဈေး) က `prices[right]` (ရောင်းဈေး) ထက် သက်သာနေစေချင်ပါတယ်။
- အကယ်၍ `prices[left] < prices[right]` ဖြစ်နေရင် အကျိုးအမြတ် ရနိုင်တဲ့အတွက် profit ကို တွက်ချက်ပြီး `maxProfit` ကို Update လုပ်ပါမယ်။
- အကယ်၍ `prices[left] >= prices[right]` ဖြစ်သွားရင် (ဆိုလိုတာက ရောင်းဈေးက ဝယ်ဈေးထက် ပိုသက်သာနေရင်)၊ အဲ့ဒီ `right` နေ့ဟာ ပိုပြီးဈေးသက်သာတဲ့ ဝယ်စရာနေ့အသစ် ဖြစ်သွားပါပြီ။ ဒါကြောင့် ဝယ်တဲ့နေ့ pointer ကို `left = right` ဆိုပြီး ချက်ချင်း ရွှေ့လိုက်ပါမယ်။
- ခြေလှမ်းတိုင်းမှာ `right` ကို တစ်လှမ်းချင်း တိုးသွားပါမယ်။

> **Time Complexity:** $O(n)$ - Array ကို တစ်ခေါက်ပဲ ပတ်လို့ဖြစ်ပါတယ်။
> **Space Complexity:** $O(1)$ - memory ပိုသုံးစရာ မလိုလို့ ဖြစ်ပါတယ်။

Java Solution

```
class Solution {
    public int maxProfit(int[] prices) {
        int left = 0; // Buy pointer
        int right = 1; // Sell pointer
        int maxProfit = 0;

        while (right < prices.length) {
            // ဝယ်ဈေးထက် ရောင်းဈေးက ပိုကြီးနေလျှင် (အမြတ်ရလျှင်)
            if (prices[left] < prices[right]) {
                int currentProfit = prices[right] - prices[left];
                maxProfit = Math.max(maxProfit, currentProfit);
            } else {
                // ရောင်းဈေးက ဝယ်ဈေးထက် ပိုသက်သာသွားပါက ထိုရက်ကို ဝယ်ရက်အသစ်အဖြစ် သတ်မှတ်
သည်
                left = right;
            }
            right++; // ရောင်းရက်ကို တစ်ရက်တိုးသည်
        }

        return maxProfit;
    }
}
```

၃။ Longest Substring Without Repeating Characters

Given a string `s` , find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

ရှင်းလင်းချက်

ပေးထားတဲ့ string ထဲမှာ တူညီတဲ့ စာလုံးထပ်ပြီး ပါဝင်ခြင်းမရှိတဲ့ (No repeating characters) အရှည်ဆုံး substring ရဲ့ အလျားကို ရှာရမှာ ဖြစ်ပါတယ်။

ဒါကို **Dynamic Size Sliding Window** နဲ့ ဖြေရှင်းနိုင်ပါတယ်။

- `left` နဲ့ `right` pointer နှစ်ခု သုံးပြီး ဝင်းဒိုးတစ်ခု ဖန်တီးပါမယ်။
- `HashSet` (သို့မဟုတ် character status tracking array) တစ်ခု သုံးပြီး လက်ရှိ ဝင်းဒိုးထဲမှာ ရှိနေတဲ့ စာလုံးတွေကို မှတ်သားထားပါမယ်။
- `right` pointer ကို ညာဘက်သို့ တစ်လုံးချင်း တိုးပြီး ဝင်းဒိုးကို ချဲ့သွားပါမယ်။
  - ဝင်လာတဲ့ စာလုံးက `HashSet` ထဲမှာ မရှိသေးရင် `HashSet` ထဲ ထည့်ပြီး ဝင်းဒိုးအကျယ်ကို `maxLength = Math.max(maxLength, right - left + 1)` ဆိုပြီး တွက်ပါမယ်။
  - ဝင်လာတဲ့ စာလုံးက `HashSet` ထဲမှာ ရှိပြီးသား ဖြစ်နေရင် (Duplicate တွေ့ရင်)၊ အဲ့ဒီ စာလုံး ထပ်နေတာ မပျောက်မချင်း ဝင်းဒိုးရဲ့ ဘယ်ဘက်ဆုံးက စာလုံးတွေကို `HashSet` ထဲကနေ ဖယ်ထုတ်ပြီး `left` pointer ကို ညာဘက်သို့ တိုးရွှေ့ (ဝင်းဒိုးကို ကျဉ်း) ပေးရပါမယ်။

> **Time Complexity:** $O(n)$ - Pointer တစ်ခုချင်းစီဟာ အများဆုံး Array အလျားအတိုင်း တစ်ကြိမ်သာ သွားရလို့ ဖြစ်ပါတယ်။
> **Space Complexity:** $O(\min(m, n))$ - `m` သည် character set အရွယ်အစားဖြစ်ပြီး၊ window ထဲရှိ unique character များကို သိမ်းထားရန် ဖြစ်သည်။

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int lengthOfLongestSubstring(String s) {
        Set<Character> charSet = new HashSet<>();
        int left = 0;
        int maxLength = 0;

        for (int right = 0; right < s.length(); right++) {
```

```
            char c = s.charAt(right);

            // စာလုံးအသစ်သည် လက်ရှိ window ထဲတွင် ရှိပြီးသားဖြစ်နေပါက
            // ၎င်းစာလုံး ပျောက်သွားသည်အထိ left pointer ကို ရွှေ့ပြီး window ကို ကျဉ်းစေသည်
            while (charSet.contains(c)) {
                charSet.remove(s.charAt(left));
                left++;
            }

            // စာလုံးအသစ်အား window ထဲ ထည့်သွင်းသည်
            charSet.add(c);
            // အရှည်ဆုံး အလျားကို တွက်သည်
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

၄။ Longest Repeating Character Replacement

You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

Example 1:

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating character, which is 4.
There may exists other ways to achieve this answer too.
```

ရှင်းလင်းချက်

String တစ်ခု ပေးထားပြီး စာလုံးတွေကို အများဆုံး k ကြိမ်အထိ တခြားစာလုံးတွေနဲ့ အစားထိုး ပြောင်းလဲခွင့် ရှိပါတယ်။ အစားထိုးပြီးတဲ့နောက် တူညီတဲ့စာလုံးတွေချည်းပဲ ရှိနေတဲ့ အရှည်ဆုံး Substring အလျားကို မေးတာ ဖြစ်ပါတယ်။

ဒီပုစ္ဆာကို ဖြေရှင်းဖို့ Sliding Window နဲ့ Frequency Map (သို့မဟုတ် character counting array) ကို တွဲသုံးရပါမယ်။

- လက်ရှိ ဝင်းဒိုး `[left, right]` ထဲမှာ ရှိတဲ့ စာလုံးတွေရဲ့ ကြိမ်နှုန်း (Frequency) ကို သိမ်းထားပါမယ်။
- ဝင်းဒိုးထဲမှာ အများဆုံးပါဝင်တဲ့ စာလုံးရဲ့ ကြိမ်နှုန်းကို `maxCount` လို့ ခေါ်ပါမယ်။
- ဝင်းဒိုးတစ်ခုလုံးရဲ့ အလျား `(right - left + 1)` ထဲကနေ `maxCount` ကို နှုတ်လိုက်ရင် ကျန်တဲ့ စာလုံးတွေဟာ **ငါတို့ အစားထိုးပြောင်းလဲပစ်ရမယ့် စာလုံးအရေအတွက်** ဖြစ်ပါတယ်။
- အဲ့ဒီ ပြောင်းလဲရမယ့် အရေအတွက်ဟာ k ထက် သာလွန်သွားရင် (`(right - left + 1) - maxCount > k`)၊ ဒီဝင်းဒိုးဟာ မမှန်ကန်တော့တဲ့အတွက် `left` က စာလုံးရဲ့ ကြိမ်နှုန်းကို လျှော့ပြီး `left` ကို ညာဘက်သို့ တိုးရွှေ့ရပါမယ်။
- ဝင်းဒိုး မှန်ကန်နေသရွေ့ အရှည်ဆုံးအလျားကို Update လုပ်သွားပါမယ်။

> **Time Complexity:** $O(n)$ - string တစ်ပတ်ပဲ ပတ်ရလို့ဖြစ်ပါတယ်။
> **Space Complexity:** $O(26) = O(1)$ - uppercase English character ၂၆ လုံးအတွက်သာ counting array သိမ်းလို့ ဖြစ်ပါတယ်။

Java Solution

```java
class Solution {
    public int characterReplacement(String s, int k) {
        int[] count = new int[26]; // English alphabet size
        int maxCount = 0; // လက်ရှိ window ထဲရှိ အများဆုံးပါသော စာလုံး၏ ကြိမ်နှုန်း
        int left = 0;
        int maxLength = 0;

        for (int right = 0; right < s.length(); right++) {
            // ဝင်လာသော စာလုံး၏ ကြိမ်နှုန်းကို တိုးသည်
            count[s.charAt(right) - 'A']++;
            // လက်ရှိ window ထဲရှိ အများဆုံးပါဝင်သော character count ကို update လုပ်သည်
            maxCount = Math.max(maxCount, count[s.charAt(right) - 'A']);

            // ပြောင်းလဲရန် လိုအပ်သော စာလုံးအရေအတွက်သည် k ထက် ကျော်လွန်နေပါက window အား ကျဉ်း
စေသည်
            int windowSize = right - left + 1;
            if (windowSize - maxCount > k) {
                count[s.charAt(left) - 'A']--;
                left++;
            }

            // မှန်ကန်သော window အရွယ်အစားကို အဖြေအဖြစ် တွက်သည်
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

၅။ Permutation in String

Given two strings `s1` and `s2`, return `true` *if* `s2` *contains a permutation of* `s1`, *or* `false` *otherwise*.

In other words, return `true` if one of `s1` 's permutations is the substring of `s2` .

Example 1:

```
Input: s1 = "ab", s2 = "eidbaooo"
Output: true
Explanation: s2 contains one permutation of s1 ("ba").
```

Example 2:

```
Input: s1 = "ab", s2 = "eidboaoo"
Output: false
```

ရှင်းလင်းချက်

String `s2` ထဲမှာ `s1` ရဲ့ Permutation (စာလုံးစီစဉ်မှု ပုံစံကွဲ) တစ်ခုခု Substring အဖြစ် ပါဝင်နေသလား စစ်ဆေးရမှာ ဖြစ်ပါတယ်။ Permutation ဆိုတာ စာလုံးတွေရဲ့ နေရာလွဲနေပေမယ့် ပါဝင်တဲ့ စာလုံးတွေနဲ့ သူတို့ရဲ့ ကြိမ်နှုန်းတွေ တူညီနေတာကို ဆိုလိုပါတယ်။

ဒါကြောင့် ဒါဟာ **Fixed-size Sliding Window (ပုံသေအရွယ်အစား ဝင်းဒိုး)** ပြဿနာ ဖြစ်ပါတယ်။

- ဝင်းဒိုးရဲ့ အရွယ်အစားဟာ `s1.length()` အတိုင်း ပုံသေ ဖြစ်ပါတယ်။
- အရင်ဦးဆုံး `s1` ထဲမှာ ပါတဲ့ စာလုံးတွေရဲ့ frequency count ကို array မှာ သိမ်းပါမယ်။
- ပြီးရင် `s2` ရဲ့ ပထမဆုံး `s1.length()` အလျားရှိတဲ့ ဝင်းဒိုးရဲ့ character frequency ကိုလည်း တွက်ထားပါမယ်။
- အဲ့ဒီ frequency counts နှစ်ခု တူညီနေရင် `true` ပြန်ပါမယ်။
- မတူသေးရင် ဝင်းဒိုးကို ညာဘက်သို့ တစ်လုံးချင်း တိုးရွှေ့ပါမယ်။
    - ညာဘက်က စာလုံးအသစ်ကို ပေါင်းထည့်ပြီး၊ ဘယ်ဘက်က ထွက်သွားတဲ့ စာလုံးကို နှုတ်ပစ်ပါမယ်။
    - ခြေလှမ်းတိုင်းမှာ frequencies တူမတူ စစ်ဆေးသွားပါမယ်။

> **Time Complexity:** $O(26 \cdot n) = O(n)$ - iteration တိုင်းမှာ character array ၂၆ လုံး တူမတူ တိုက်စစ်လို့ ဖြစ်ပါတယ်။
> **Space Complexity:** $O(26) = O(1)$

Java Solution

```java
import java.util.Arrays;

class Solution {
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length()) {
            return false;
        }

        int[] s1Count = new int[26];
```

```
        int[] s2Count = new int[26];

        // s1 ၏ အလျားအတိုင်း ပထမဆုံး window ၏ frequencies ကို တွက်သည်
        for (int i = 0; i < s1.length(); i++) {
            s1Count[s1.charAt(i) - 'a']++;
            s2Count[s2.charAt(i) - 'a']++;
        }

        // frequencies တူညီပါက permutation တွေ့ရှိပြီ ဖြစ်သည်
        if (Arrays.equals(s1Count, s2Count)) {
            return true;
        }

        // Window အား ညာဘက်သို့ တစ်လုံးချင်း တိုးရွှေ့သွားသည်
        for (int i = s1.length(); i < s2.length(); i++) {
            // ညာဘက်အသစ်ဝင်လာသော စာလုံးအား ပေါင်းသည်
            s2Count[s2.charAt(i) - 'a']++;
            // ဘယ်ဘက်မှ ထွက်သွားသော စာလုံးအား နုတ်သည်
            s2Count[s2.charAt(i - s1.length()) - 'a']--;

            // frequencies အား တိုက်စစ်သည်
            if (Arrays.equals(s1Count, s2Count)) {
                return true;
            }
        }

        return false;
    }
}
```

၆။ Minimum Window Substring

Given two strings `s` and `t` of lengths `m` and `n` respectively, return *the minimum window substring of* `s` *such that every character in* `t` *(including duplicates) is included in the window*. If there is no such substring, return *the empty string* `""` .

The testcases will be generated such that the answer is **unique**.

Example 1:

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
```

Example 2:

```
Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
```

Example 3:

```
Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
```

```
Since the window of s only has one 'a', return empty string.
```

ရှင်းလင်းချက်

ဒါကတော့ Sliding Window ရဲ့ အလွန် နာမည်ကြီးပြီး အမေးများတဲ့ **Hard** အဆင့်ရှိ မေးခွန်း ဖြစ်ပါတယ်။ String `s` ထဲမှာ String `t` ရဲ့ စာလုံးအားလုံး (ထပ်နေတာတွေ အပါအဝင်) ပါဝင်တဲ့ **အသေးဆုံး/အတိုဆုံး Substring** ကို ရှာခိုင်းတာ ဖြစ်ပါတယ်။

ဒါကို **Dynamic Size Sliding Window** နဲ့ စနစ်တကျ ဖြေရှင်းနိုင်ပါတယ်။

- ပထမဆုံး `t` ထဲက စာလုံးတွေရဲ့ Frequency Map ကို ဆောက်ပါမယ်။ (ဥပမာ- `tMap`)
- နောက်ထပ် map တစ်ခု `windowMap` ထားပြီး လက်ရှိ window ထဲရှိ စာလုံးတွေကို မှတ်ပါမယ်။
- `have` နဲ့ `need` ဆိုတဲ့ variable နှစ်ခု သုံးပြီး လိုအပ်တဲ့ စာလုံးအရေအတွက် ပြည့်စုံမှုရှိမရှိကို လျင်မြန်စွာ စစ်ပါမယ်။
    - `need` = `t` ထဲက ထူးခြားစာလုံး အရေအတွက် (Unique characters count in `t`)။
    - `have` = လက်ရှိ window ထဲမှာ `t` ရဲ့ character requirements ပြည့်စုံသွားတဲ့ ထူးခြား စာလုံး အရေအတွက်။
- `right` pointer ကို ညာဘက်ရွှေ့ပြီး window ကို ချဲ့ပါမယ်။ ဝင်လာတဲ့ စာလုံးရဲ့ Frequency က `tMap` ထဲက တန်ဖိုးနဲ့ ကိုက်ညီသွားရင် `have++` လုပ်ပါမယ်။
- `have == need` ဖြစ်သွားပြီ ဆိုရင် (လိုအပ်တဲ့ စာလုံးတွေ အကုန်ပါသွားပြီဆိုရင်)၊ ဒီ window က မှန်ကန်နေပါပြီ။
    - အဲ့ဒီအခါ အတိုဆုံး window ဖြစ်ဖို့အတွက် ဘယ်ဘက်ဆုံးက စာလုံးတွေကို `left` pointer တိုးပြီး တစ်လုံးချင်း ဖယ်ထုတ် (window ကို ကျဉ်း) ကြည့်ပါမယ်။
    - ဖယ်ထုတ်လိုက်လို့ စာလုံးလိုအပ်ချက် မပြည့်စုံတော့တဲ့ အခြေအနေ (`have < need`) ရောက်မှ `left` ရွှေ့တာကို ရပ်ပြီး `right` ကို ဆက်ချဲ့ပါမယ်။
    - ပတ်နေစဉ်တစ်လျှောက်လုံးမှာ အတိုဆုံး substring အလျားနဲ့ index ကို မှတ်သားထားပါမယ်။

> **Time Complexity:** $O(m+n)$ - string `s` ရော `t` ရောကို တစ်ပတ်စီပဲ ပတ်ရလို့ ဖြစ်ပါတယ်။
> **Space Complexity:** $O(m+n)$ - character counting dictionaries အတွက် ဖြစ်ပါတယ်။

Java Solution

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public String minWindow(String s, String t) {
        if (s == null || t == null || s.length() < t.length()) {
            return "";
        }

        // t ၏ စာလုံးလိုအပ်ချက်များကို မှတ်သားရန် Map
        Map<Character, Integer> tMap = new HashMap<>();
```

```
        for (char c : t.toCharArray()) {
            tMap.put(c, tMap.getOrDefault(c, 0) + 1);
        }

        // လက်ရှိ window အတွင်းရှိ စာလုံးများကို မှတ်သားရန် Map
        Map<Character, Integer> windowMap = new HashMap<>();

        int left = 0;
        int have = 0;
        int need = tMap.size(); // Unique character count in t

        int minLen = Integer.MAX_VALUE;
        int minLeft = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);

            // t ထဲတွင် လိုအပ်သော စာလုံးဖြစ်ပါက windowMap ထဲ ထည့်သည်
            if (tMap.containsKey(c)) {
                windowMap.put(c, windowMap.getOrDefault(c, 0) + 1);
                // လိုအပ်သော အရေအတွက်အတိအကျ ပြည့်သွားပါက 'have' ကို တိုးသည်
                if (windowMap.get(c).equals(tMap.get(c))) {
                    have++;
                }
            }

            // လိုအပ်ချက်များအားလုံး ပြည့်စုံနေပါက window အား တတ်နိုင်သမျှ ကျဉ်းစေသည်
            while (have == need) {
                // အနည်းဆုံး window အလျားကို update လုပ်သည်
                int windowSize = right - left + 1;
                if (windowSize < minLen) {
                    minLen = windowSize;
                    minLeft = left;
                }

                char leftChar = s.charAt(left);
                // ဘယ်ဘက်ဆုံးစာလုံးအား window ထဲမှ ဖယ်ထုတ်သည်
                if (tMap.containsKey(leftChar)) {
                    // ဖယ်ထုတ်လိုက်သဖြင့် လိုအပ်ချက်ထက် နည်းသွားပါက 'have' ကို လျှော့သည်
                    if (windowMap.get(leftChar).equals(tMap.get(leftChar))) {
                        have--;
                    }
                    windowMap.put(leftChar, windowMap.get(leftChar) - 1);
                }
                left++;
            }
        }

        return minLen == Integer.MAX_VALUE ? "" : s.substring(minLeft, minLeft + minLen);
    }
}
```

၇။ Sliding Window Maximum

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
```

Window တစ်ခုချင်းစီအတွက် အကြီးဆုံးတန်ဖိုးတွေက အောက်ပါအတိုင်း ဖြစ်ပါတယ်။

Window	Maximum
`[1, 3, -1]`	`3`
`[3, -1, -3]`	`3`
`[-1, -3, 5]`	`5`
`[-3, 5, 3]`	`5`
`[5, 3, 6]`	`6`
`[3, 6, 7]`	`7`

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

ရှင်းလင်းချက်

Array တစ်ခုနဲ့ ကိန်းသေ k (Window size) ပေးထားပါတယ်။ ဝင်းဒိုးကို ညာဘက်သို့ တစ်ဆင့်ချင်း ရွှေ့သွားတဲ့အခါ လက်ရှိ ဝင်းဒိုးတစ်ခုချင်းစီထဲမှာ ရှိတဲ့ **အကြီးဆုံး တန်ဖိုး (Maximum element)** ကို စုစည်းပြီး Array အဖြစ် ပြန်ထုတ်ပေးရမှာ ဖြစ်ပါတယ်။

ပုံမှန်အားဖြင့် ဝင်းဒိုးတစ်ခုချင်းစီထဲက အကြီးဆုံးကို လိုက်ရှာရင် $O(k)$ ကြာပြီး၊ တစ်ပတ်လုံးအတွက် $O(n \cdot k)$ ကြာပါမယ်။ k က ကြီးမားတဲ့အခါ Time Limit Exceeded (TLE) ဖြစ်သွားပါလိမ့်မယ်။

ဒါကို $O(n)$ နဲ့ ဖြေရှင်းဖို့ **Monotonic Decreasing Queue (or Deque - Double Ended Queue)** နည်းပညာကို သုံးရပါမယ်။ Deque ထဲမှာ Array ရဲ့ တန်ဖိုးတွေအစား သူတို့ရဲ့ `index` တွေကို သိမ်းဆည်းသွားပါမယ်။

- Deque ထဲက index တွေနဲ့ သက်ဆိုင်တဲ့ တန်ဖိုးတွေကို အမြဲတမ်း ကြီးစဉ်ငယ်လိုက် (Decreasing order) ဖြစ်နေအောင် ထိန်းသိမ်းပါမယ်။
- ဝင်လာတဲ့ `nums[right]` တန်ဖိုးထက် သေးငယ်တဲ့ Deque ရဲ့ အနောက်ဆုံး (tail) က index တွေကို အကုန် pop ထုတ်ပစ်ပါမယ်။ ဘာဖြစ်လို့လဲဆိုတော့ အခု အသစ်ဝင်လာတဲ့ကောင်က ပိုကြီးပြီး window ထဲမှာ ပိုကြာကြာကျန်မှာဖြစ်လို့၊ သူ့ထက်သေးတဲ့ အရင်ကောင်တွေဟာ ဘယ်တော့မှ အဖြေ (Maximum) ဖြစ်လာစရာ အကြောင်းမရှိတော့လို့ပါ။
- ပြီးရင် လက်ရှိ index `right` ကို Deque ရဲ့ အနောက်မှာ ထည့်ပါမယ်။

- Deque ရဲ့ အရှေ့ဆုံး (head) က index က လက်ရှိ window ရဲ့ ဘောင်အပြင်ဘက် ရောက်သွားရင် (`head_index < right - k + 1`)၊ ၎င်းကို အရှေ့ကနေ pop ထုတ်ပစ်ပါမယ်။
- Deque ရဲ့ အရှေ့ဆုံးမှာ ကျန်နေတဲ့ index နဲ့ သက်ဆိုင်တဲ့ တန်ဖိုး `nums[deque.peekFirst()]` ဟာ လက်ရှိ window ရဲ့ အကြီးဆုံးတန်ဖိုး အမြဲ ဖြစ်နေပါလိမ့်မယ်။

> **Time Complexity:** $O(n)$ - element တစ်ခုချင်းစီဟာ Deque ထဲကို အများဆုံး တစ်ကြိမ်ပဲ ဝင်ပြီး တစ်ကြိမ်ပဲ ထွက်ရလို့ ဖြစ်ပါတယ်။
> **Space Complexity:** $O(k)$ - Deque ရဲ့ အမြင့်ဆုံး size က k ဖြစ်လို့ဖြစ်ပါတယ်။

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Deque;

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || nums.length == 0 || k == 0 || k > nums.length) {
            return new int[0];
        }

        int n = nums.length;
        int[] result = new int[n - k + 1];
        int ri = 0; // result array index

        // Index များကို Decreasing order အတိုင်း သိမ်းမည့် Deque
        Deque<Integer> deque = new ArrayDeque<>();

        for (int i = 0; i < n; i++) {
            // Window ၏ အပြင်ဘက် ရောက်သွားသော index အဟောင်းများအား Deque အရှေ့မှ
ဖယ်ထုတ်သည်
            if (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
                deque.pollFirst();
            }

            // လက်ရှိတန်ဖိုးထက် ငယ်သော Deque အနောက်ဘက်ရှိ index များအားလုံးကို ဖယ်ထုတ်သည်
            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }

            // လက်ရှိ index အား Deque ထဲ ထည့်သည်
            deque.offerLast(i);

            // ပထမဆုံး window ပြည့်သွားသည်မှ စတင်၍ အဖြေစသိမ်းသည်
            if (i >= k - 1) {
                result[ri++] = nums[deque.peekFirst()];
            }
        }

        return result;
    }
}
```

အခန်း ၇ - Stack and Queue

Stack ဆိုတာ Data တွေကို အထပ်လိုက် စီထားတဲ့ ပုံစံမျိုး (ဥပမာ - စာအုပ်ပုံ) အလုပ်လုပ်တဲ့ Data Structure တစ်ခု ဖြစ်ပါတယ်။ သူ့မှာ အဓိက စည်းကမ်းတစ်ခု ရှိပါတယ်။ အဲ့ဒါကတော့ **LIFO (Last-In, First-Out)** ဖြစ်ပါတယ်။

LIFO (Last-In, First-Out)

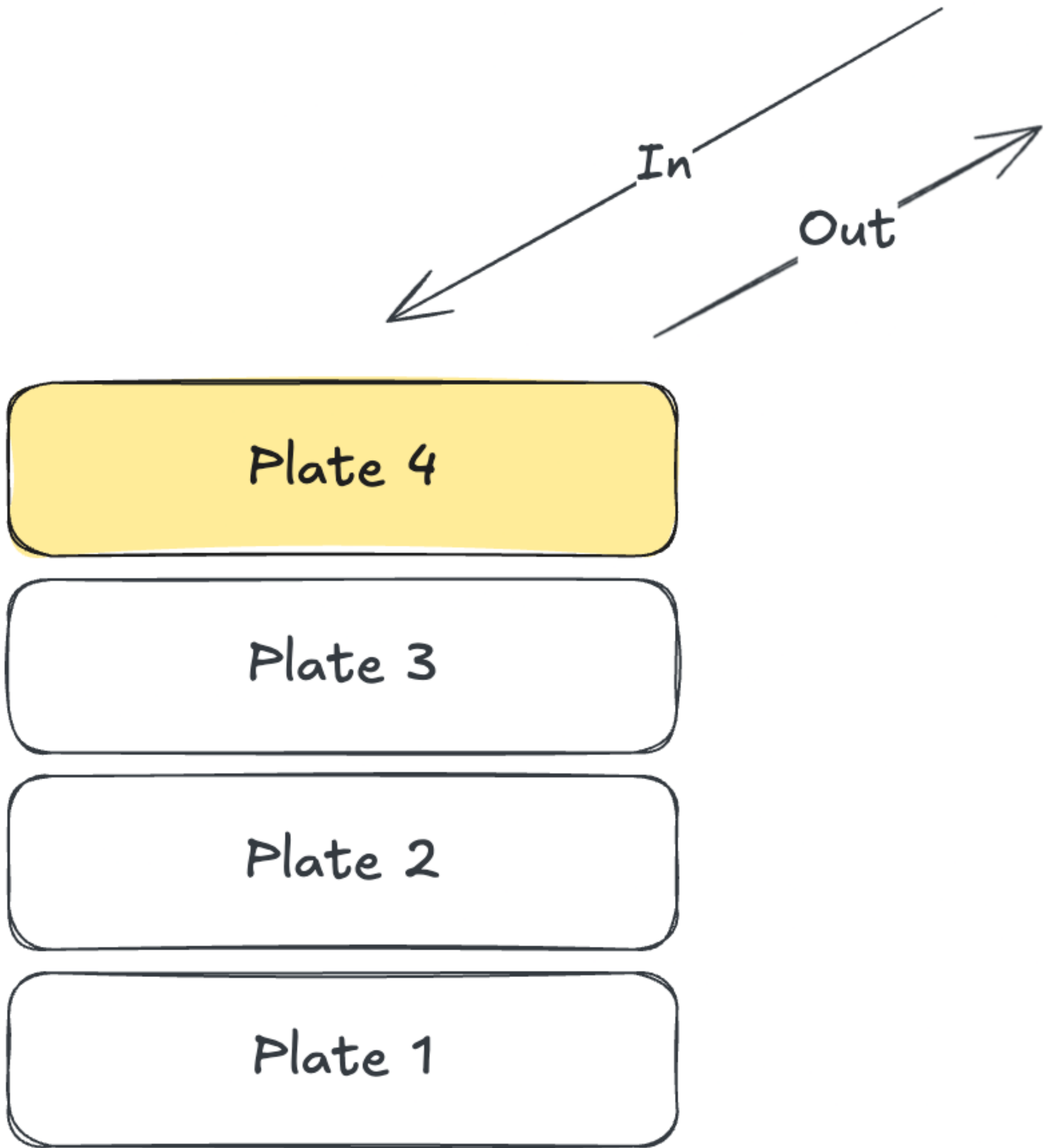
LIFO ဆိုတာ "နောက်ဆုံးမှ ဝင်လာတဲ့ကောင်က ပထမဆုံး ပြန်ထွက်ရမယ်" လို့ ဆိုလိုတာပါ။

ဥပမာ

စားသောက်ဆိုင် (Buffet) တစ်ခုမှာ ပန်းကန်ပြားတွေကို ပုံးထဲမှာ တစ်ချပ်ပေါ် တစ်ချပ် ထပ်ပြီး စီထားတာကို မြင်ဖူးမှာပါ။ စားပွဲထိုးက ဆေးပြီးသား ပန်းကန်အသစ်တွေကို ထည့်တဲ့အခါ အပေါ်ဆုံးကနေပဲ ထပ်တင်ရပါတယ်။ (Push)

စားမယ့်သူတွေ လာယူတဲ့အခါကျရင်လည်း အပေါ်ဆုံးက ပန်းကန် (နောက်ဆုံးမှ ဆေးပြီး တင်လိုက်တဲ့ ပန်းကန်) ကိုပဲ အရင်ဆုံး ယူသုံးကြပါတယ်။ (Pop)

အောက်ဆုံးမှာရှိတဲ့ ပန်းကန်ကို လိုချင်ရင် အပေါ်က ပန်းကန်တွေ အကုန်လုံးကို အရင် ဖယ်ထုတ်ပစ်မှသာ ရပါလိမ့်မယ်။ ဒါဟာ LIFO ရဲ့ အကောင်းဆုံး ဥပမာပါပဲ။

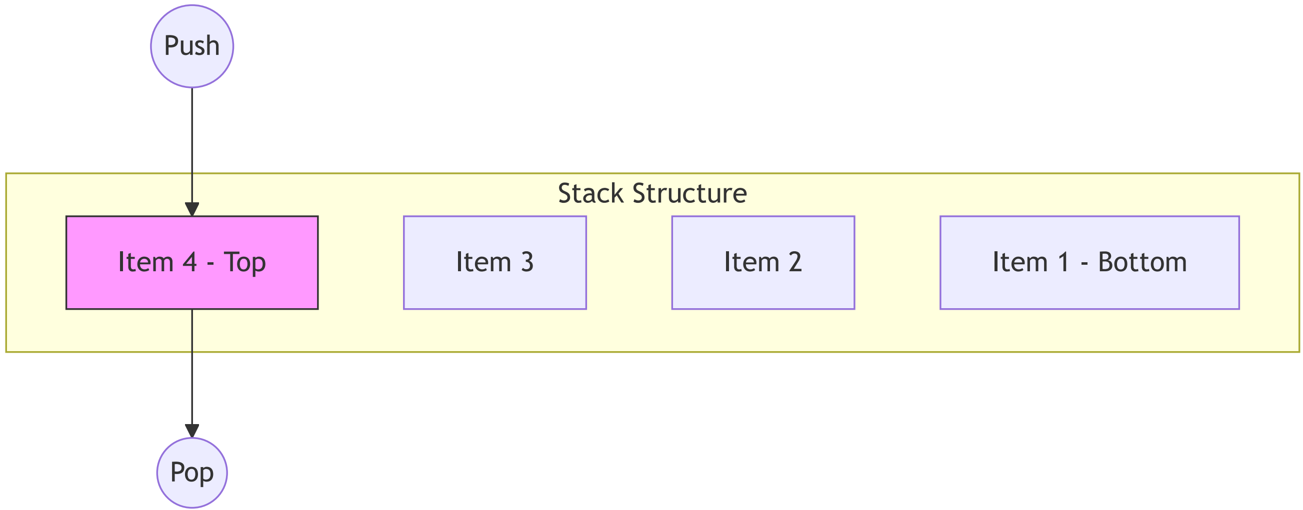


အဓိက လုပ်ဆောင်ချက်များ (Operations)

Stack တစ်ခုမှာ အဓိက လုပ်ဆောင်ချက် ၃ ခု ရှိပါတယ် —

1. **Push:** Data တစ်ခုကို Stack ရဲ့ အပေါ်ဆုံး (Top) မှာ ထည့်တာ။
2. **Pop:** Stack ရဲ့ အပေါ်ဆုံးက Data ကို ဖယ်ရှားပြီး ပြန်ထုတ်ပေးတာ။
3. **Peek / Top:** အပေါ်ဆုံးမှာ ဘာရှိလဲဆိုတာကို Data မထုတ်ဘဲ ကြည့်တာ။

ဒီအလုပ် ၃ ခုလုံးဟာ Stack ရဲ့ အပေါ်ဆုံး (Top) မှာပဲ တိုက်ရိုက် လုပ်တာ ဖြစ်တဲ့အတွက် Time Complexity ဟာ $O(1)$ ဖြစ်ပါတယ်။



Java ဖြင့် Stack အသုံးပြုခြင်း

Java မှာဆိုရင် `java.util.Stack` ကို သုံးပြီး အလွယ်တကူ ရေးနိုင်ပါတယ်။

နောက်ပိုင်း ခေတ်သစ် Java မှာ `Stack` အစား `Deque` (ဥပမာ `ArrayDeque`) ကို Stack အနေနဲ့ သုံးဖို့ ပိုမို အကြံပြုကြပါတယ်။ ဒါပေမယ့် Concept ကို နားလည်ဖို့အတွက် ရိုးရှင်းတဲ့ `Stack` ကို အရင် ကြည့်ရအောင်။

```java
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Stack တစ်ခု တည်ဆောက်ခြင်း
        Stack<String> history = new Stack<>();

        // Data ထည့်ခြင်း (Push) - O(1)
        history.push("google.com");
        history.push("facebook.com");
        history.push("youtube.com");

        // အပေါ်ဆုံးက ဘာလဲ ကြည့်ခြင်း (Peek) - O(1)
        System.out.println("Top Website: " + history.peek()); // youtube.com

        // Data ထုတ်ခြင်း (Pop) - O(1)
        String lastVisited = history.pop();
        System.out.println("Back from: " + lastVisited);

        // လက်ရှိ အပေါ်ဆုံးက ဘာလဲ
        System.out.println("Now Top is: " + history.peek()); // facebook.com
    }
}
```

Time Complexity

Operation	Average Case	Worst Case
push	$O(1)$	$O(1)$

pop	$O(1)$	$O(1)$
peek / top	$O(1)$	$O(1)$
search	$O(n)$	$O(n)$

> **မှတ်ချက်:** Stack မှာ Data ရှာဖွေတာ (Search) က $O(n)$ ကြာပါတယ်။ ဘာကြောင့်လဲဆိုတော့ အောက်ဆုံးက Data ကို လိုချင်ရင် အပေါ်က Data တွေကို တစ်ခုချင်းစီ ဖယ်ပြီးမှ ရှာလို့ ရလို့ပါ။ Data ကို လျင်မြန်စွာ ရှာဖွေချင်တယ် ဆိုရင် Hash Table လိုမျိုး အခြား Data Structure တွေကို သုံးရပါမယ်။

လက်တွေ့အသုံးချမှုများ

နေ့စဉ်သုံးနေတဲ့ Software တွေမှာ Stack ကို နေရာများစွာမှာ အသုံးပြုကြပါတယ် —

- **Undo / Redo Function:** စာရိုက်တဲ့အခါ မှားလို့ နောက်ပြန်ဆုတ် (Ctrl+Z) ချင်ရင် နောက်ဆုံးရိုက်ခဲ့တဲ့ အခြေအနေတွေကို Stack ထဲကနေ တစ်ခုချင်း ပြန်ထုတ်ပေးတာပါ။
- **Browser History (Back / Forward Button):** Back Button နှိပ်ရင် နောက်ဆုံးကြည့်ခဲ့တဲ့ Website (Top of Stack) ကို ပြန်ပြတာပါ။
- **Call Stack:** Programming Language အများစုမှာ Function (Method) တစ်ခုထဲကနေ နောက်တစ်ခုကို ထပ်ဆင့်ခေါ်တဲ့အခါ ဘယ်နေရာကနေ ပြန်စရမလဲဆိုတာ မှတ်သားဖို့ Stack Memory ကို သုံးပါတယ်။

> Design Pattern စာအုပ်တွင် Command Pattern တွင် Stack သုံးထားတာကို တွေ့နိုင်မှာပါ။

Questions

Stack ကို နားလည်သွားပြီ ဆိုရင် Stack ကို လက်တွေ့ အသုံးချရမယ့် LeetCode ပုစ္ဆာ တချို့ကို ကြည့်ရအောင်။

Valid Parentheses

Given a string `s` containing just the characters `'('` , `')'` , `'{'` , `'}'` , `'['` and `']'` , determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]{}"
Output: true
```

Example 3:

```
Input: s = "(]"
Output: false
```

ဒီပုစ္ဆာက Stack ကို လေ့လာတဲ့ သူတွေအတွက် အခြေခံ အကျဆုံးနဲ့ အရေးအကြီးဆုံး ပုစ္ဆာ ဖြစ်ပါတယ်။ ကျွန်တော်တို့ string ထဲမှာ အဖွင့် ကွင်း `(` , `{` , `[` တွေ့တိုင်း **သူနဲ့တွဲတဲ့ အပိတ်ကွင်း** ကို Stack ထဲ push ထည့်ပါမယ်။ အပိတ် ကွင်း တွေ့တဲ့ အခါမှာတော့ Stack ရဲ့ အပေါ်ဆုံးက ကွင်းကို pop လုပ်ပြီး လက်ရှိ အပိတ်ကွင်းနဲ့ ကိုက်ညီမှု ရှိမရှိ စစ်ဆေးပါမယ်။

> $O(n)$ time complexity, $O(n)$ space complexity ဖြစ်ရပါမယ်။

```java
import java.util.Stack;

class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            // အဖွင့်ကွင်းများ ဖြစ်လျှင် သက်ဆိုင်ရာ အပိတ်ကွင်းကို Stack ထဲ ထည့်သည်
            if (c == '(') {
                stack.push(')');
            } else if (c == '{') {
                stack.push('}');
            } else if (c == '[') {
                stack.push(']');
            } else {
                // အပိတ်ကွင်း ဖြစ်ပါက Stack အလွတ်ဖြစ်နေသလား သို့မဟုတ်
                // Stack အပေါ်ဆုံးက ကွင်းနှင့် မတူဘူးလား စစ်ဆေးသည်
                if (stack.isEmpty() || stack.pop() != c) {
                    return false;
                }
            }
        }

        // Stack လွတ်သွားမှသာ ကွင်းတွေ အစုံအလင် ပိတ်ပြီးသွားတာဖြစ်သည်
        return stack.isEmpty();
    }
}
```

Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Example 1:

```
Input:
["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]
[[], [1], [2], [0], [], [], [], []]

Output: [null,null,null,null,0,null,2,1]

Explanation:
MinStack minStack = new MinStack();
minStack.push(1);
minStack.push(2);
minStack.push(0);
minStack.getMin(); // return 0
minStack.pop();
minStack.top();    // return 2
minStack.getMin(); // return 1
```

ပုံမှန် Stack မှာ အသေးဆုံး တန်ဖိုး (Minimum element) ကို လိုချင်ရင် အပေါ်ဆုံးကနေ အောက်ခြေ အထိ လိုက်ရှာရတဲ့အတွက် $O(n)$ အချိန် ကြာပါတယ်။ အခုက Pushing နဲ့ Popping လုပ်နေချိန်မှာပါ $O(1)$ Time Complexity နဲ့ အသေးဆုံး တန်ဖိုးကို ချက်ချင်း သိချင်တာပါ။

ဒီ ပြဿနာကို ဖြေရှင်းဖို့အတွက် Stack နှစ်ခု (သို့မဟုတ် Stack တစ်ခုတည်းမှာ Minimum Value ကိုပါ တွဲသိမ်းတာ) သုံးလို့ ရပါတယ်။ `minStack` ဆိုတဲ့ နောက်ထပ် Stack တစ်ခု ထားပြီး၊ အသစ်ဝင်လာတဲ့ `val` က လက်ရှိ အသေးဆုံး တန်ဖိုးထက် ငယ်နေရင် (သို့မဟုတ် ညီနေရင်) `minStack` ထဲကိုပါ ထည့် သိမ်းပေးလိုက်ယုံပါပဲ။ ဒီနည်းနဲ့ operation တိုင်း $O(1)$ ဖြစ်ပြီး၊ နောက်ထပ် Stack တစ်ခု ပိုသုံးရတဲ့ အတွက် space ကတော့ $O(n)$ ဖြစ်ပါတယ်။

```
import java.util.Stack;

class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
```

```
            minStack = new Stack<>();
        }

        public void push(int val) {
            stack.push(val);
            // minStack လွတ်နေလျှင် (သို့) ထည့်မည့်တန်ဖိုးသည် လက်ရှိ အနည်းဆုံးထက် ငယ်/ညီ နေလျှင်
            if (minStack.isEmpty() || val <= minStack.peek()) {
                minStack.push(val);
            }
        }

        public void pop() {
            // ထုတ်လိုက်သော တန်ဖိုးသည် အနည်းဆုံး တန်ဖိုးဖြစ်နေပါက minStack မှပါ ထုတ်သည်
            if (stack.pop().equals(minStack.peek())) {
                minStack.pop();
            }
        }

        public int top() {
            return stack.peek();
        }

        public int getMin() {
            return minStack.peek();
        }
    }
```

Evaluate Reverse Polish Notation

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note that:

- The valid operators are `'+'` , `'-'` , `'*'` , and `'/'` .
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Reverse Polish Notation (RPN) ဟာ ဂဏန်း (Operands) တွေကို အရင်ရေးပြီးမှ ပေါင်းနှုတ်မြှောက်စား သင်္ကေတ (Operators) တွေကို နောက်ကနေ ကပ်ရေးတဲ့ နည်းစနစ်ပါ။ ဥပမာ `2 + 1` ကို RPN နဲ့ ရေးရင် `2 1 +` ဖြစ်ပါတယ်။ RPN ဟာ Stack နှင့် တွဲဖက် အသုံးပြုဖို့ အကောင်းဆုံး အရာ ဖြစ်ပါတယ်။ ကွန်ပျူတာတွေဟာ သင်္ချာညီမျှခြင်းတွေကို တွက်ချက်တဲ့အခါ Stack ကို သုံးပြီး အခုလိုပဲ တွက်ချက်ပါတယ်။

အလွယ်ဆုံး စဥ်းစားနည်းကတော့ —

- ဂဏန်း တွေ့ရင် Stack ထဲ ထည့် (Push)။
- သင်္ကေတ (`+` , `-` , `*` , `/`) တွေ့ရင် Stack ထဲကနေ ဂဏန်း ၂ ခု ထုတ် (Pop) ပြီး တွက်ချက်ကာ ရလာတဲ့ အဖြေကို Stack ထဲ ပြန်ထည့် (Push) ပြုလုပ်သွားယုံပါပဲ။
- နောက်ဆုံး Stack ထဲမှာ ကျန်နေတဲ့ ဂဏန်း ဟာ အဖြေ ဖြစ်ပါတယ်။

```
import java.util.Stack;

class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();

        for (String token : tokens) {
            // Operator တွေ့ပါက တွက်ချက်ခြင်းပြုလုပ်သည်
            if (token.equals("+")) {
                stack.push(stack.pop() + stack.pop());
            } else if (token.equals("*")) {
                stack.push(stack.pop() * stack.pop());
            } else if (token.equals("-")) {
                // အရင်ထွက်လာသည့်ဂဏန်းသည် အနောက်က ဂဏန်းဖြစ်သည်
                int a = stack.pop();
                int b = stack.pop();
                stack.push(b - a);
            } else if (token.equals("/")) {
                int a = stack.pop();
                int b = stack.pop();
                stack.push(b / a);
            } else {
                // ဂဏန်းတွေ့ပါက Stack ထဲသို့ Integer ပြောင်း၍ ထည့်သည်
                stack.push(Integer.parseInt(token));
            }
        }

        return stack.pop();
    }
}
```

Daily Temperatures

Given an array of integers `temperatures` represents the daily temperatures, return an array `answer` such that `answer[i]` is the number of days you have to wait after the ith day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

Example 1:

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

Example 2:

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

ဒီပုစ္ဆာမှာ နေ့စဉ် အပူချိန်တွေ ပေးထားပြီး နောက်ရက်တွေမှာ ကိုယ့်ထက် ပိုပူတဲ့နေ့ ရောက်ဖို့ ဘယ်နှရက် စောင့်ရမလဲ ဆိုတာကို ရှာရမှာပါ။

ရိုးရိုး Nested Loop နဲ့ ရှာရင် $O(n^2)$ ကြာပါမယ်။ Stack (Monotonic Decreasing Stack) ကို သုံးပြီး $O(n)$ နဲ့ ဖြေရှင်းလို့ ရပါတယ်။
Stack ထဲမှာ အပူချိန်ရဲ့ Index တွေကို သိမ်းပါမယ်။ လက်ရှိ ဝင်လာတဲ့ အပူချိန်က Stack ရဲ့ အပေါ်ဆုံးမှာ ရှိတဲ့ အပူချိန်ထက် ကြီးနေရင်၊ အဲ့ဒီ Stack ထဲက Index အတွက် ပိုပူတဲ့နေ့ကို တွေ့သွားပါပြီ။ ဒါကြောင့် Stack ကနေ Pop လုပ်ပြီး ရက်ကွာခြားချက် (Current Index - Popped Index) ကို အဖြေအဖြစ် မှတ်ယူပါမယ်။

```java
import java.util.Stack;

class Solution {
    public int[] dailyTemperatures(int[] temperatures) {
        int[] result = new int[temperatures.length];
        // Index များကို သိမ်းမည့် Stack
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < temperatures.length; i++) {
            // လက်ရှိ အပူချိန်သည် Stack အပေါ်ဆုံးမှ အပူချိန်ထက် ကြီးနေသရွေ့
            while (!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()]) {
                int prevIndex = stack.pop();
                // စောင့်ရမည့် ရက်အရေအတွက်ကို မှတ်သည်
                result[prevIndex] = i - prevIndex;
            }
            stack.push(i);
        }

        return result;
    }
}
```

Car Fleet

There are `n` cars at given miles away from the starting mile `0`, traveling to reach the mile `target`. You are given two integer arrays `position` and `speed`, both of length `n`, where `position[i]` is the starting mile of the ith car and `speed[i]` is the speed of the ith car in miles per hour.

A car cannot pass another car ahead of it. It can only catch up to another car and then drive at the same speed. A car fleet is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

Return the number of car fleets that will arrive at the destination.

Example 1:

```
Input: target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]
Output: 3
Explanation:
- The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at
12.
- The car starting at 0 does not catch up to any other car, so it is a fleet by itself.
- The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6.
  The fleet moves at speed 1 until it reaches target.
```

ဒီပုစ္ဆာမှာ ကားတွေက ပန်းတိုင် (target) ကို သွားကြမှာပါ။ အနောက်ကကားက အမြန်မောင်းလို့ အရှေ့က ကားကို မီသွားရင်တောင် ကျော်တက်လို့ မရဘဲ အရှေ့ကားနဲ့ တစ်တန်းတည်း (Fleet) အဖြစ် ပတ်သက် သွားရမှာပါ။ နောက်ဆုံး ပန်းတိုင်ရောက်ရင် Fleet ဘယ်နှစု ဖြစ်မလဲဆိုတာ မေးတာပါ။

ဒီပုစ္ဆာကို ဖြေရှင်းဖို့ ကားတွေကို သူတို့ရဲ့ စတင်တဲ့ နေရာ (Position) အလိုက် အစဉ်လိုက် (အကြီးမှ အငယ်၊ ပန်းတိုင်နဲ့ အနီးဆုံးကနေ အဝေးဆုံး) စီ (Sort) စဉ်းစားပါမယ်။
ပြီးရင် ကားတစ်စီးစီ ပန်းတိုင်ရောက်ဖို့ ကြာမယ့် အချိန် `(Distance / Speed)` ကို တွက်ပါမယ်။
အနောက်ကလာတဲ့ ကားရဲ့ အချိန်ဟာ အရှေ့ကကားရဲ့ အချိန်ထက် ငယ်နေရင် (သို့) ညီနေရင် (ဆိုလိုတာ က ပိုမြန်နေရင် သို့မဟုတ် တချိန်တည်းရောက်ရင်) အရှေ့ကားကို မီသွားပြီး Fleet တစ်ခုတည်း ဖြစ်သွား ပါမယ်။ အချိန် ပိုကြီးနေရင်တော့ (ဆိုလိုတာက ပိုနှေးနေရင်) မမီတဲ့အတွက် Fleet အသစ် တစ်ခု ဖြစ်ပါ မယ်။

```java
import java.util.Arrays;
import java.util.Stack;

class Solution {
    public int carFleet(int target, int[] position, int[] speed) {
        int n = position.length;
        // ကားများ၏ Position နှင့် ကြာမည့်အချိန်ကို တွဲသိမ်းရန် Array
        double[][] cars = new double[n][2];
        for (int i = 0; i < n; i++) {
            cars[i][0] = position[i];
            cars[i][1] = (double)(target - position[i]) / speed[i];
        }

        // Position အလိုက် အငယ်မှ အကြီးစီသည်
        Arrays.sort(cars, (a, b) -> Double.compare(a[0], b[0]));

        Stack<Double> stack = new Stack<>();

        // ပန်းတိုင်နှင့် အနီးဆုံး ကား (Array ရဲ့ အနောက်ဆုံး) မှစ၍ ပြောင်းပြန် စစ်ဆေးသည်
        for (int i = n - 1; i >= 0; i--) {
            double time = cars[i][1];
            // Stack လွတ်နေလျှင် သို့မဟုတ် လက်ရှိကားသည် Stack ထဲရှိ အရှေ့ကားထက် အချိန်ပိုကြာလျှင်
(နှေးပြီး မမီလျှင်)
            if (stack.isEmpty() || time > stack.peek()) {
```

```
                stack.push(time);
            }
            // အချိန်ပိုနည်းလျှင် (ပိုမြန်လျှင်) အရှေ့ကားကို မီသွား၍ Fleet တူသွားသဖြင့် Stack ထဲ ထပ်မထည့်
ပါ
        }

        // Stack ၏ အရွယ်အစားသည် Fleet အရေအတွက် ဖြစ်သည်
        return stack.size();
    }
}
```

Largest Rectangle In Histogram

Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1` , return the area of the largest rectangle in the histogram.

Example 1:

```
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The largest rectangle is shown in the shaded area, which has an area = 10
units.
```

ဒီပုစ္ဆာကတော့ Histogram (တိုင်လေးတွေ) ထဲမှာ အကျယ်ဆုံး စတုဂံ ဧရိယာ ကို ရှာရမယ့် မေးခွန်းပါ။ Stack ကို သုံးပြီး $O(n)$ အချိန်နဲ့ တွက်ထုတ်နိုင်တဲ့ အကောင်းဆုံး အသုံးချမှုတစ်ခု ဖြစ်ပါတယ်။

Monotonic Increasing Stack ကို သုံးပါမယ်။ Stack ထဲမှာ တိုင်တွေရဲ့ `index` တွေကို အစဉ်လိုက် မြင့်လာသရွေ့ သိမ်းသွားပါမယ်။
လက်ရှိ တိုင်ရဲ့ အမြင့်က Stack ရဲ့ အပေါ်ဆုံးက တိုင်အမြင့်ထက် ငယ်သွားပြီ ဆိုရင်၊ Stack ထဲက အရှည်ဆုံး တိုင်တွေအတွက် နောက်ထပ် ဆန့်လို့ရမယ့် ဘက် (ညာဘက် နယ်နိမိတ်) ကို တွေ့သွားပါပြီ။ ဒါကြောင့် Stack ကနေ Pop လုပ်ပြီး၊ အမြင့် `(height = heights[popped index])` နဲ့ အကျယ် `(width = current_index - new_top_index - 1)` ကို မြှောက်ကာ ဧရိယာ အကြီးဆုံးကို တွက်ချက် မှတ်သားသွားပါမယ်။

```java
import java.util.Stack;

class Solution {
    public int largestRectangleArea(int[] heights) {
        int maxArea = 0;
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i <= heights.length; i++) {
            // Array ပြီးဆုံးသွားလျှင် ကျန်နေသည့် Stack ကို ရှင်းရန် အမြင့် 0 ထည့်စဉ်းစားသည်
            int h = (i == heights.length) ? 0 : heights[i];

            // လက်ရှိအမြင့်သည် Stack အပေါ်ဆုံးမှ အမြင့်ထက် ငယ်နေသရွေ့ Pop လုပ်၍ ဧရိယာတွက်သည်
            while (!stack.isEmpty() && h < heights[stack.peek()]) {
                int height = heights[stack.pop()];
                // Stack လွတ်သွားလျှင် အစကနေ လက်ရှိအထိ ကျယ်သည်ဟု ယူဆသည်
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
                maxArea = Math.max(maxArea, height * width);
            }
            stack.push(i);
```

```
            }

            return maxArea;
        }
    }
```

Queue

Queue ဆိုတာ Data တွေကို **တန်းစီစောင့်** တဲ့ ပုံစံ (ဥပမာ - ဆိုင်ရှေ့မှာ လူတန်းစီတာ) အလုပ်လုပ်တဲ့ Data Structure ပါ။ Stack နဲ့ ပြောင်းပြန်ဖြစ်ပြီး အဓိက စည်းကမ်းက **FIFO (First-In, First-Out)** ဖြစ်ပါတယ်။

FIFO (First-In, First-Out)

FIFO ဆိုတာ "ပထမဆုံး ဝင်လာတဲ့ကောင်က ပထမဆုံး ပြန်ထွက်ရမယ်" လို့ ဆိုလိုတာပါ။

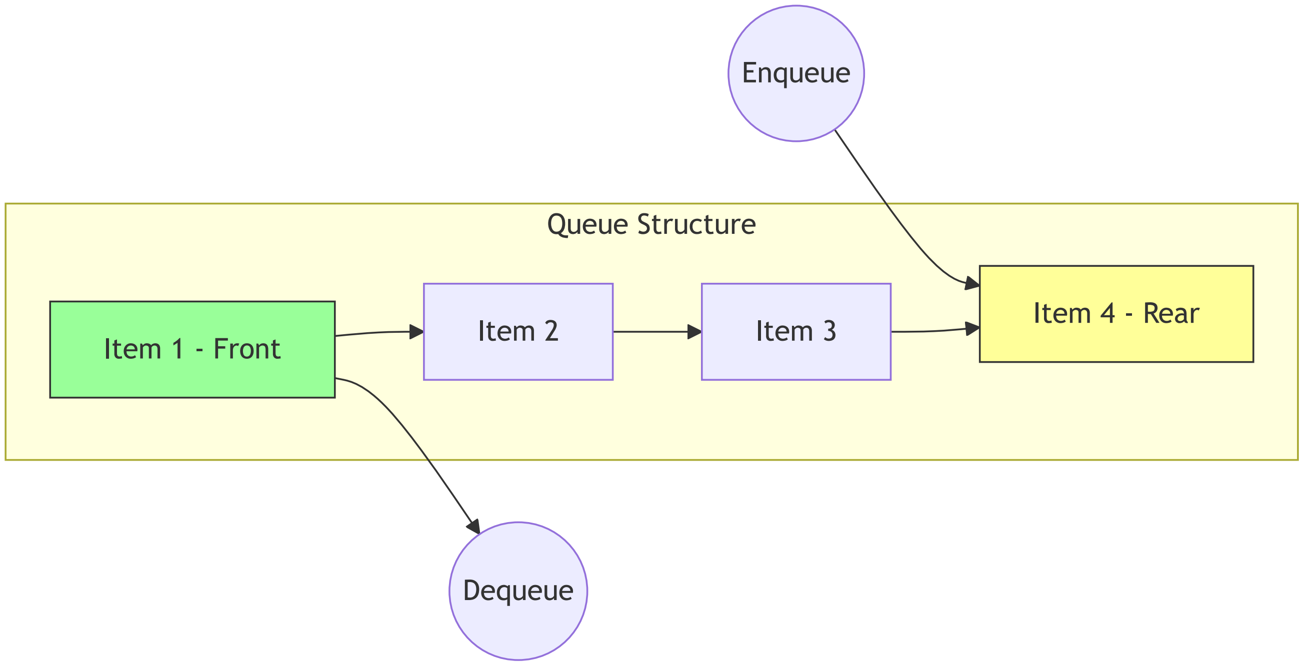
> **လွယ်ကူသော ဥပမာ (Ticket Counter Analogy):**
> ရုပ်ရှင်ရုံ လက်မှတ်ရောင်းကောင်တာရှေ့မှာ လူတွေ တန်းစီနေတာကို မြင်ဖူးမှာပါ။ အစောဆုံး လာတန်းစီတဲ့သူက အရှေ့ဆုံး (Front) မှာ ရှိပြီး၊ ပထမဆုံး လက်မှတ်ရတယ်။ နောက်မှ လာသူတွေက အနောက်ဆုံး (Rear / Back) ကနေ တန်းဆက်စီရတယ်။ ဘယ်သူမှ ဖြတ်ဝင်လို့ မရဘူး။ ဒါဟာ FIFO ရဲ့ အကောင်းဆုံး ဥပမာပါ။

အဓိက လုပ်ဆောင်ချက်များ (Operations)

Queue တစ်ခုမှာ အဓိက လုပ်ဆောင်ချက် ၃ ခု ရှိပါတယ် —

1. **Enqueue:** Data တစ်ခုကို Queue ရဲ့ အနောက်ဆုံး (Rear) မှာ ထည့်တာ။
2. **Dequeue:** Queue ရဲ့ အရှေ့ဆုံး (Front) က Data ကို ဖယ်ရှားပြီး ပြန်ထုတ်ပေးတာ။
3. **Peek / Front:** အရှေ့ဆုံးမှာ ဘာရှိလဲဆိုတာကို Data မထုတ်ဘဲ ကြည့်တာ။

ဒီအလုပ် ၃ ခုလုံးဟာ Queue ရဲ့ အစွန်း (Front/Rear) မှာပဲ လုပ်တာ ဖြစ်တဲ့အတွက် Time Complexity ဟာ $O(1)$ ဖြစ်ပါတယ်။



Java ဖြင့် Queue အသုံးပြုခြင်း

Java မှာ `Queue` က interface တစ်ခု ဖြစ်ပြီး၊ `LinkedList` ဒါမှမဟုတ် `ArrayDeque` နဲ့ တည်ဆောက်လေ့ ရှိပါတယ်။ (`ArrayDeque` က ပိုမြန်လို့ အကြံပြုပါတယ်။)

```
import java.util.ArrayDeque;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<>();

        // Data ထည့်ခြင်း (Enqueue) - O(1)
        queue.offer("Customer 1");
        queue.offer("Customer 2");
        queue.offer("Customer 3");

        // အရှေ့ဆုံးက ဘာလဲ ကြည့်ခြင်း (Peek) - O(1)
        System.out.println("Front: " + queue.peek()); // Customer 1

        // Data ထုတ်ခြင်း (Dequeue) - O(1)
        String served = queue.poll();
        System.out.println("Serving: " + served); // Customer 1

        // လက်ရှိ အရှေ့ဆုံးက ဘာလဲ
        System.out.println("Now Front: " + queue.peek()); // Customer 2
    }
}
```

> **မှတ်ချက်:** `offer` / `poll` / `peek` က Queue လွတ်/ပြည့် အခြေအနေမှာ exception မပစ်ဘဲ `false` / `null` ပြန်ပေးလို့ ပိုလုံခြုံပါတယ်။ (`add` / `remove` / `element` ကတော့ exception ပစ်ပါတယ်။)

Deque (Double-Ended Queue)

Deque ("deck" လို့ အသံထွက်) ဆိုတာ **အစွန်းနှစ်ဖက်စလုံး** (Front ရော Rear ရော) ကနေ ထည့်/ထုတ် လုပ်လို့ရတဲ့ Queue ပါ။ ဒါကြောင့် Deque တစ်ခုတည်းနဲ့ Stack (LIFO) အဖြစ်ရော Queue (FIFO) အဖြစ်ရော သုံးနိုင်ပါတယ်။ အရင်အခန်း (Sliding Window Maximum) မှာ သုံးခဲ့တဲ့ Monotonic Deque က ဒီ structure ပါပဲ။

	Stack (LIFO)	Queue (FIFO)	Deque
ထည့်တဲ့နေရာ	Top	Rear	နှစ်ဖက်လုံး
ထုတ်တဲ့နေရာ	Top	Front	နှစ်ဖက်လုံး

လက်တွေ့အသုံးချမှုများ

Queue ကို **"ရောက်လာတဲ့ အစဉ်အတိုင်း တစ်ခုပြီးတစ်ခု လုပ်ဆောင်"** ရမယ့် နေရာတိုင်းမှာ သုံးပါတယ် —

- **Background Job / Task Queue:** Server မှာ email ပို့ခြင်း၊ image ပြောင်းခြင်း စတဲ့ အလုပ်တွေကို ရောက်လာတဲ့ အစဉ်အတိုင်း တန်းစီ လုပ်ဆောင်တယ်။
- **Print Queue:** Printer ဆီ စာ ၃ စောင် တစ်ပြိုင်နက် ပို့ရင် ပထမ ပို့တဲ့ဟာ အရင် ထွက်တယ်။
- **Request / Message Queue:** Web server တွေ၊ Kafka/RabbitMQ လို message broker တွေက request တွေကို Queue နဲ့ စီမံတယ်။
- **BFS (Breadth-First Search):** Graph / Tree ကို အလွှာလိုက် ရှာဖွေတဲ့အခါ Queue သုံးတယ်။ (Chapter 18 မှာ အသေးစိတ် တွေ့ရပါမယ်။)

Questions

Queue ကို နားလည်သွားပြီ ဆိုရင် Stack နဲ့ Queue ၂ ခု ဆက်စပ်ပုံ မြင်သာအောင် classic ပုစ္ဆာ ၃ ပုဒ်ကို ကြည့်ရအောင်။

Implement Queue using Stacks

Stack ၂ ခုကိုသာ သုံးပြီး FIFO Queue တစ်ခု (`push` , `pop` , `peek` , `empty`) ကို တည်ဆောက်ပါ။

Stack က LIFO ဖြစ်ပေမယ့် Stack ၂ ခု တွဲသုံးရင် order ကို ပြောင်းပြန် လှန်ပြီး FIFO ဖြစ်အောင် လုပ်နိုင်ပါတယ်။

- `input` stack — အသစ်ဝင်လာတာတွေ push လုပ်ဖို့။
- `output` stack — ထုတ်ဖို့။ `output` လွတ်နေချိန် `input` ထဲက အကုန် ကူးထည့် လိုက်ရင် အစဉ်အတိုင်း ပြောင်းပြန် ဖြစ်သွားလို့ FIFO ရတယ်။

ဒီ "ကူးထည့်ခြင်း" က ရံဖန်ရံခါသာ ဖြစ်တဲ့အတွက် operation တစ်ခုချင်း **amortized** $O(1)$ ဖြစ်ပါတယ်။

```java
import java.util.Stack;

class MyQueue {
    private Stack<Integer> input;
    private Stack<Integer> output;

    public MyQueue() {
        input = new Stack<>();
        output = new Stack<>();
    }

    public void push(int x) {
        input.push(x);
    }

    public int pop() {
        peek(); // output ထဲ data ရှိအောင် အရင်ပြင်သည်
        return output.pop();
    }

    public int peek() {
        // output လွတ်နေမှသာ input ထဲက အကုန် ကူးထည့်သည် (order ပြောင်းပြန်ဖြစ်သွားသည်)
        if (output.isEmpty()) {
            while (!input.isEmpty()) {
                output.push(input.pop());
            }
        }
        return output.peek();
    }

    public boolean empty() {
        return input.isEmpty() && output.isEmpty();
    }
}
```

Implement Stack using Queues

ဒီတစ်ခါ ပြောင်းပြန်ပါ။ Queue သုံးပြီး LIFO Stack (`push` , `pop` , `top` , `empty`) တည်ဆောက်ပါ။

Queue တစ်ခုတည်းနဲ့ လုပ်လို့ ရပါတယ်။ နည်းလမ်းက — element အသစ် `push` လုပ်တိုင်း၊ အဲ့ဒီနောက် Queue ထဲမှာ **ရှေ့က ကျန်နေတဲ့ element တွေ အကုန်** ကို dequeue လုပ်ပြီး ပြန် enqueue လုပ်လိုက်တယ်။ ဒါဆို အသစ်ဝင်တဲ့ element က အရှေ့ဆုံးကို ရောက်သွားလို့ Stack လို နောက်ဆုံးဝင်တာ အရင်ထွက်တဲ့ ပုံစံ ဖြစ်သွားတယ်။

ဒီနည်းမှာ `push` က $O(n)$ ဖြစ်ပြီး၊ `pop` / `top` က $O(1)$ ဖြစ်ပါတယ်။

```java
import java.util.ArrayDeque;
import java.util.Queue;

class MyStack {
    private Queue<Integer> queue;
```

```
    public MyStack() {
        queue = new ArrayDeque<>();
    }

    public void push(int x) {
        queue.offer(x);
        // အသစ်ဝင်တာ အရှေ့ဆုံးရောက်အောင် ရှေ့က ကျန်သမျှကို နောက်ပြန်ပို့သည်
        for (int i = 1; i < queue.size(); i++) {
            queue.offer(queue.poll());
        }
    }

    public int pop() {
        return queue.poll();
    }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
}
```

Next Greater Element I

`nums1` သည် `nums2` ၏ subset ဖြစ်သည်။ `nums1` ထဲက ဂဏန်းတစ်ခုစီအတွက်၊ `nums2` ထဲမှာ ၎င်းရဲ့ **ညာဘက်က ပထမဆုံး ပိုကြီးတဲ့ ဂဏန်း** ကို ရှာပါ။ မရှိရင် `-1` ။

Example:

```
Input:  nums1 = [4,1,2], nums2 = [1,3,4,2]
Output: [-1,3,-1]
```

ဒါက Daily Temperatures နဲ့ တူတဲ့ **Monotonic Stack** pattern ပါ။ `nums2` ကို ဖြတ်ရင်း Stack (ကြီးစဉ်ငယ်လိုက်) သုံးပြီး ဂဏန်းတစ်ခုစီရဲ့ "next greater" ကို HashMap ထဲ မှတ်ထားပါမယ်။ ပြီးရင် `nums1` အတွက် Map ထဲက အဖြေကို ထုတ်ယူရုံပါပဲ။ Time $O(n+m)$။

```
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Map<Integer, Integer> nextGreater = new HashMap<>();
        Stack<Integer> stack = new Stack<>();

        for (int num : nums2) {
            // လက်ရှိ num သည် Stack အပေါ်ဆုံးထက် ကြီးနေသရွေ့ ၎င်းတို့၏ next greater ဖြစ်သည်
            while (!stack.isEmpty() && num > stack.peek()) {
                nextGreater.put(stack.pop(), num);
            }
            stack.push(num);
        }
```

```
        int[] result = new int[nums1.length];
        for (int i = 0; i < nums1.length; i++) {
            result[i] = nextGreater.getOrDefault(nums1[i], -1);
        }

        return result;
    }
}
```

အခန်း ၈ - Linked List

ယခင် အခန်းတွေမှာ Array အကြောင်းကို လေ့လာခဲ့ပါတယ်။ Array ဟာ Data တွေကို Memory ထဲမှာ တစ်ခုနဲ့တစ်ခု ကပ်လျက် စီထားတဲ့အတွက် Index နဲ့ ချက်ချင်း ရှာလို့ ရတယ် ($O(1)$)။ ဒါပေမယ့် အလယ်ထဲမှာ Data အသစ် ထည့်ချင်တာ၊ ဖျက်ချင်တာ ဆိုရင် နောက်က Data တွေ အကုန်လုံးကို တစ်နေရာစီ ရွှေ့ပေးရတဲ့အတွက် $O(n)$ ကုန်ကျပါတယ်။

Linked List ဆိုတာက Data တွေကို Memory ထဲမှာ ကပ်လျက် မစီတော့ဘဲ၊ နေရာ အနှံ့အပြားမှာ ထားပြီး တစ်ခုနဲ့တစ်ခုကို **Pointer** နဲ့ ချိတ်ဆက်ထားတဲ့ Data Structure တစ်မျိုး ဖြစ်ပါတယ်။

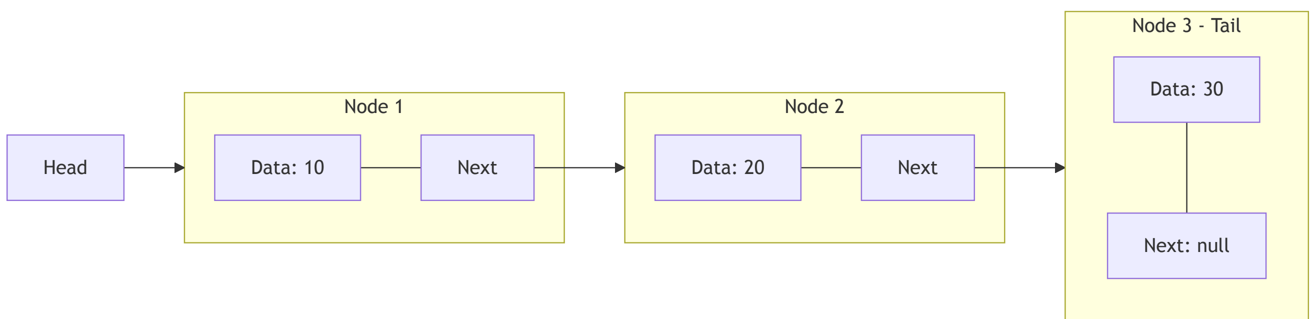
Treasure Hunt ဥပမာ
Treasure Hunt ကစားနည်း ကို စဉ်းစားကြည့်ပါ။ ပထမ စာရွက်မှာ "နောက်တစ်ခု အတွက် မီးဖိုချောင်ထဲ ကြည့်ပါ" လို့ ရေးထားတယ်။ မီးဖိုချောင်ရောက်တော့ နောက်စာရွက်မှာ "အိပ်ခန်းထဲ ကြည့်ပါ" လို့ ရေးထားတယ်။ စာရွက်တွေ (Data) က နေရာအနှံ့ ပြန့်ကျဲနေပေမယ့် တစ်ခုက နောက်တစ်ခုကို ညွှန်ပြ (Point) ပေးထားတဲ့အတွက် အစဉ်လိုက် လိုက်လို့ ရပါတယ်။ ဒါပေမယ့် တန်းပြီး "၅ ခုမြောက် စာရွက်" ဆို တန်းမသွားနိုင်ဘဲ ပထမ ကနေ တစ်ဆင့်ချင်း လိုက်ရှာရတာ Linked List နဲ့ တူပါတယ်။

Node Concept

Linked List ရဲ့ အခြေခံ အစိတ်အပိုင်းကို **Node** လို့ ခေါ်ပါတယ်။ Node တစ်ခုစီမှာ အပိုင်း ၂ ပိုင်း ပါပါတယ် —

1. **Data (Value):** သိမ်းချင်တဲ့ တန်ဖိုး။
2. **Next (Pointer):** နောက်လာမယ့် Node ရဲ့ Memory လိပ်စာ (Reference)။

အရှေ့ဆုံး Node ကို **Head** လို့ ခေါ်ပြီး၊ နောက်ဆုံး Node ကို**Tail** လို့ ခေါ်ပါတယ်။ Tail Node ရဲ့ Next က ဘာမှ မရှိတော့တဲ့အတွက် `null` ကို ညွှန်ထားပါတယ်။




```
// Node တစ်ခု၏ ဖွဲ့စည်းပုံ
class ListNode {
    int val;         // Data သိမ်းရန်
    ListNode next;   // နောက် Node ကို ညွှန်ပြရန်
```

```
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
```

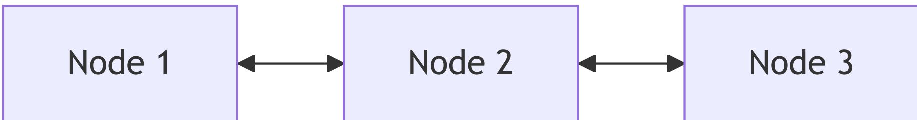
Singly Linked List

Node တစ်ခုက နောက်တစ်ခုကိုသာ (တစ်လမ်းသွား) ညွှန်ပြထားတဲ့ ပုံစံကို **Singly Linked List** လို့ ခေါ်ပါတယ်။ အရှေ့ကနေ အနောက်ကို သာ သွားလို့ ရပြီး၊ အနောက် ကို ပြန်သွား မရပါ။

```
public class LinkedListExample {
    public static void main(String[] args) {
        // Node တွေ တည်ဆောက်ခြင်း
        ListNode head = new ListNode(10);
        head.next = new ListNode(20);
        head.next.next = new ListNode(30);

        // အစကနေ အဆုံးထိ လှည့်ပတ်ခြင်း (Traversal) - O(n)
        ListNode current = head;
        while (current != null) {
            System.out.println(current.val);
            current = current.next; // နောက် Node သို့ ရွှေ့သည်
        }
    }
}
```

Doubly Linked List

Doubly Linked List မှာတော့ Node တစ်ခုစီက နောက် Node (`next`) ကိုရော၊ ရှေ့ Node (`prev`) ကို ပါ ညွှန်ပြထားပါတယ်။ ဒါကြောင့် အရှေ့ကို ရော အနောက်ကို ရော (နှစ်လမ်းသွား) သွားလို့ ရပါတယ်။ Memory ပိုကုန်ပေမယ့် အနောက်ကို ပြန်လှည့်ဖို့ လွယ်ကူပါတယ်။




```
class DoublyNode {
    int val;
    DoublyNode prev;   // ရှေ့ Node
    DoublyNode next;   // နောက် Node

    DoublyNode(int val) {
        this.val = val;
    }
}
```

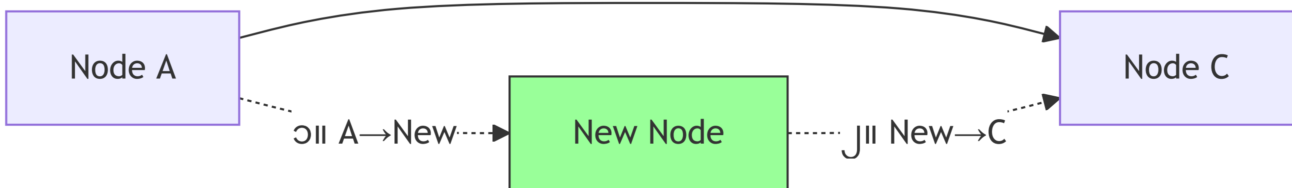
Doubly Linked List မှာ `prev` link ပါလာတဲ့အတွက် ထည့်/ဖျက်တဲ့အခါ pointer ၂ ဖက်စလုံး (ရှေ့ရော နောက်ရော) ကို ပြန်ချိတ်ပေးရပါတယ်။

```
// node အသစ်ကို existing node ရဲ့ နောက်မှာ ထည့်ခြင်း
newNode.prev = existing;
newNode.next = existing.next;
if (existing.next != null) {
    existing.next.prev = newNode; // နောက်က node ၏ prev ကို ပြန်ချိတ်
}
existing.next = newNode;

// node တစ်ခုကို ဖျက်ခြင်း (ရှေ့/နောက် ၂ ဖက်ကို တိုက်ရိုက် ဆက်ပေးနိုင်)
if (node.prev != null) node.prev.next = node.next;
if (node.next != null) node.next.prev = node.prev;
```

Singly Linked List နဲ့ မတူတာက — ဖျက်မယ့် node ရဲ့ ရှေ့က node ကို သီးခြား လိုက်ရှာစရာ မလိုဘဲ၊ `node.prev` ကနေ တန်းရတာပါ။ ဒါက LRU Cache လိုမျိုး node ကို မကြာခဏ ဖျက်/ရွှေ့ရတဲ့ နေရာ တွေမှာ အသုံးဝင်ပါတယ်။

Insert / Delete by Pointer

Linked List ရဲ့ အကောင်းဆုံး အချက်ကတော့ Pointer ကို ပြောင်းရုံနဲ့ Node တစ်ခုကို ထည့်တာ၊ ဖျက်တာ လုပ်နိုင်တာပါ။ Array လို နောက်က Data တွေ ရွှေ့ပေးစရာ မလိုတော့ပါ။ (သိမ်းမယ့်နေရာကို သိပြီး သား ဆိုရင် $O(1)$ ဖြစ်ပါတယ်)။




```
// Node A နှင့် Node C ကြားထဲ Node အသစ်ထည့်ခြင်း
ListNode newNode = new ListNode(15);
newNode.next = nodeA.next; // အသစ်က C ကို ညွှန်စေသည်
nodeA.next = newNode;      // A က အသစ်ကို ညွှန်စေသည်

// Node တစ်ခုကို ဖျက်ခြင်း (A ၏ နောက်က Node ကို ဖျက်ရန်)
nodeA.next = nodeA.next.next; // A ကို ကျော်၍ နောက်တစ်ခုသို့ ညွှန်စေသည်
```

Array vs Linked List နှိုင်းယှဉ်ချက်

လုပ်ဆောင်ချက်	Array	Linked List
Index ဖြင့်ရှာခြင်း (Access)	$O(1)$	$O(n)$
ရှေ့ဆုံးတွင် ထည့်/ဖျက်	$O(n)$	$O(1)$

နောက်ဆုံးတွင် ထည့်	$O(1)$	$O(n)$*
ရှာဖွေခြင်း (Search)	$O(n)$	$O(n)$
Memory	ကပ်လျက် (Contiguous)	ပြန့်ကျဲ (Pointer ပိုကုန်)

> * Dynamic Array မှာ နေရာ ပြည့်သွားရင် အသစ် ချဲ့ရတဲ့အတွက် တစ်ခါတစ်ရံ $O(n)$ ဖြစ်နိုင်ပါတယ်။
> \\ Tail Pointer သိမ်းထားရင် Linked List နောက်ဆုံးမှာ ထည့်တာ $O(1)$ ဖြစ်နိုင်ပါတယ်။

ဒီနှိုင်းယှဉ်ချက်ကို ကြည့်ရင် **Index နဲ့ ရှာတာ များမယ်ဆိုရင် Array** က ပိုသင့်တော်ပါတယ်။ **ထည့်/ဖျက်တာ များမယ်ဆိုရင် Linked List** က ပိုကောင်းပေမယ့်၊ ဒါက **ထည့်/ဖျက်ရမယ့် နေရာ (သို့) ရှေ့က node pointer ကို သိပြီးသား ဖြစ်မှသာ** မှန်ပါတယ်။ နေရာကို အရင် လိုက်ရှာရဦးမယ်ဆိုရင်တော့ ရှာဖွေတာက $O(n)$ ကုန်တဲ့အတွက် Linked List ရဲ့ အားသာချက် ပျောက်သွားပါတယ်။

လက်တွေ့အသုံးချမှုများ

- **Music Playlist:** သီချင်း တစ်ပုဒ်ပြီး တစ်ပုဒ် (Next) ဖွင့်တာ၊ ပြန်သွားတာ (Previous) တွေအတွက် Doubly Linked List က အသင့်တော်ဆုံးပါ။
- **Browser History:** ရှေ့သွား/နောက်ပြန် လုပ်တဲ့အတွက်လည်း Doubly Linked List ကို သုံးနိုင်ပါတယ်။
- **LRU Cache:** အကြာဆုံး မသုံးတော့တဲ့ (Least Recently Used) Data ကို ဖယ်ထုတ်တဲ့ Cache စနစ်မှာ Doubly Linked List နဲ့ Hash Table ကို ပေါင်းသုံးကြပါတယ်။ (Interview မှာ မကြာခဏ မေးတဲ့ "LRU Cache" ပုစ္ဆာက ဒီ DLL + HashMap ပေါင်းစပ်မှုနဲ့ `get` / `put` ၂ ခုလုံးကို $O(1)$ ရအောင် တည်ဆောက်ခိုင်းတာပါ။)
- **Undo/Redo:** လုပ်ဆောင်ချက်တွေကို အစဉ်လိုက် ချိတ်ဆက် မှတ်သားရာမှာ သုံးပါတယ်။

> **မှတ်ချက်:** နေ့စဉ် App ရေးတဲ့အခါ Linked List ကို တိုက်ရိုက် ကိုယ်တိုင် ရေးသုံးဖို့ နည်းပါတယ်။ ဒါပေမယ့် Pointer/Reference ဆိုတဲ့ စဉ်းစားနည်းဟာ Tree, Graph စတဲ့ နောက်ပိုင်း အခန်းတွေအတွက် အလွန် အရေးကြီးပါတယ်။

Questions

Linked List ကို နားလည်သွားပြီ ဆိုရင် လက်တွေ့ အသုံးချရမယ့် LeetCode ပုစ္ဆာ တချို့ကို ကြည့်ရအောင်။

Reverse Linked List

Given the `head` of a singly linked list, reverse the list, and return the reversed list.

Example 1:

```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:

```
Input: head = [1,2]
Output: [2,1]
```

ဒီပုစ္ဆာက Linked List အတွက် အခြေခံ အကျဆုံး ပုစ္ဆာ ဖြစ်ပါတယ်။ Node တွေရဲ့ `next` Pointer တွေကို ပြောင်းပြန် လှည့်ပေးရမှာပါ။

အဓိက စဉ်းစားနည်းကတော့ Pointer ၃ ခု (`prev` , `current` , `next`) ကို သုံးပြီး၊ Node တစ်ခုစီရဲ့ `next` ကို သူ့ရဲ့ ရှေ့က Node (`prev`) ဆီ ပြန်လှည့်ညွှန်ပေးသွားတာပါ။

> $O(n)$ time complexity, $O(1)$ space complexity ဖြစ်ရပါမယ်။

```java
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode next = current.next; // နောက် Node ကို ကြိုသိမ်းထားသည်
            current.next = prev;          // Pointer ကို ပြောင်းပြန် လှည့်သည်
            prev = current;               // prev ကို ရှေ့သို့ ရွှေ့သည်
            current = next;               // current ကို ရှေ့သို့ ရွှေ့သည်
        }

        return prev; // prev သည် နောက်ဆုံးတွင် Head အသစ် ဖြစ်သည်
    }
}
```

Merge Two Sorted Lists

You are given the heads of two sorted linked lists `list1` and `list2` . Merge the two lists into one **sorted** list. Return the head of the merged linked list.

Example 1:

```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

စီပြီးသား (Sorted) Linked List နှစ်ခုကို ပေါင်းပြီး၊ စီပြီးသား List တစ်ခုတည်း ဖြစ်အောင် လုပ်ရမှာပါ။

Dummy Node ဆိုတဲ့ ခေါင်းစီး Node အတုတစ်ခုကို သုံးပြီး၊ List နှစ်ခုကို တစ်ပြိုင်တည်း လိုက်နှိုင်းကာ၊ တန်ဖိုး ငယ်တဲ့ Node ကို ရွေးပြီး ဆက်ချိတ်သွားတဲ့ နည်းက အရိုးရှင်းဆုံးပါ။

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Dummy Node - ခေါင်းစီး အတု (ရှုပ်ထွေးမှု လျှော့ရန်)
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;

        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                tail.next = list1;
                list1 = list1.next;
            } else {
                tail.next = list2;
                list2 = list2.next;
            }
            tail = tail.next;
        }

        // ကျန်နေသေးသော List ကို ဆက်ချိတ်သည်
        tail.next = (list1 != null) ? list1 : list2;

        return dummy.next; // Dummy ၏ နောက်က Node သည် တကယ့် Head ဖြစ်သည်
    }
}
```

Linked List Cycle

Given `head`, the head of a linked list, determine if the linked list has a cycle in it. A cycle exists if some node in the list can be reached again by continuously following the `next` pointer.

Example 1:

```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle, the tail connects to the 1st node (0-indexed).
```

Linked List ထဲမှာ စက်ဝိုင်း (Cycle) ရှိမရှိ စစ်ရမှာပါ။ Cycle ဆိုတာ Node တစ်ခုရဲ့ `next` က နောက်ပြန် တစ်ခုခုကို ညွှန်ပြီး အဆုံးမသတ် (Infinite Loop) ဖြစ်နေတာ မျိုးပါ။

ဒီပုစ္ဆာကို **Floyd's Cycle Detection (Fast & Slow Pointers)** နည်းနဲ့ ဖြေရှင်းပါတယ်။ နှေး (Slow) Pointer က တစ်လှမ်းစီ၊ မြန် (Fast) Pointer က နှစ်လှမ်းစီ သွားပါမယ်။ Cycle ရှိနေရင် မြန်တဲ့ Pointer က နှေးတဲ့ Pointer ကို တစ်နေရာရာမှာ ပြန်မီ (ထပ်) သွားပါလိမ့်မယ်။

> ဒီနည်းကို "ပြေးခုန်ပစ်ကွင်း (Race Track)" မှာ မြန်တဲ့သူက နှေးတဲ့သူကို တစ်ပတ်ပြန်ကျော်ပြီး မီသွားသလို မြင်ယောင်နိုင်ပါတယ်။ $O(n)$ time, $O(1)$ space ဖြစ်ပါတယ်။

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;

        while (fast != null && fast.next != null) {
```

```
            slow = slow.next;         // တစ်လှမ်းစီ
            fast = fast.next.next;    // နှစ်လှမ်းစီ

            if (slow == fast) {       // ပြန်ဆုံသွားလျှင် Cycle ရှိသည်
                return true;
            }
        }

        return false; // fast သည် null သို့ ရောက်လျှင် Cycle မရှိ
    }
}
```

Find Middle Node

Given the `head` of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

Example 1:

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
```

Example 2:

```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
```

List ရဲ့ အလယ် Node ကို ရှာရမှာပါ။ ဒီနေရာမှာလည်း **Fast & Slow Pointers** နည်းကို သုံးနိုင်ပါတယ်။ Fast က နှစ်လှမ်းစီ၊ Slow က တစ်လှမ်းစီ သွားရင်၊ Fast က အဆုံးရောက်တဲ့အခါ Slow က အလယ်မှာ ရှိနေပါလိမ့်မယ်။

```
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;

        // Fast သည် အဆုံးရောက်သွားလျှင် Slow သည် အလယ်တွင် ရှိနေမည်
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        return slow;
    }
}
```

Remove Nth Node From End of List

Given the `head` of a linked list, remove the nth node from the end of the list and return its head.

Example 1:

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

အဆုံးကနေ ရေတွက်လို့ `n` ခုမြောက် Node ကို ဖျက်ရမှာပါ။ List ကို တစ်ခေါက်ပဲ ဖတ်ပြီး (One Pass) ဖြေရှင်းဖို့ Pointer နှစ်ခုကြားမှာ `n` အကွာအဝေး ထားတဲ့ နည်း (Two Pointers) ကို သုံးပါမယ်။

ဒီ Solution မှာ Head ကိုယ်တိုင် ဖျက်ရတဲ့ ကိစ္စကို လွယ်ကူစေဖို့ `dummy` Node ကို head ရှေ့မှာ ထည့်ထားပြီး `slow` နဲ့ `fast` နှစ်ခုစလုံးကို `dummy` ကနေ စပါတယ်။ `slow` ကို ဖျက်မယ့် Node ရဲ့ **ရှေ့** မှာ ရပ်စေချင်တဲ့အတွက်၊ `fast` ကို `dummy` ကနေ **`n + 1` လှမ်း** ရှေ့သွားခိုင်းရပါမယ်။ ဒါမှ `slow` နဲ့ `fast` ကြားက အကွာအဝေးက `n + 1` ဖြစ်ပြီး၊ `fast` က `null` ရောက်တဲ့အခါ `slow` က ဖျက်မယ့် Node ရဲ့ ရှေ့မှာ အတိအကျ ရှိနေမှာပါ။

> **မှတ်ချက်:** `dummy` မသုံးဘဲ ဖြေချင်ရင် `fast` ကို `head` ကနေ `n` လှမ်းသာ ရှေ့သွားစေပြီး၊ Head ကိုယ်တိုင် ဖျက်ရတဲ့ အခြေအနေ (`fast == null` ဖြစ်သွားတာ) ကို သီးခြား စစ်ပေးရပါမယ်။ Dummy Node နည်းက ဒီ edge case ကို သပ်သပ်ရပ်ရပ် ဖြေရှင်းပေးတာ ဖြစ်ပါတယ်။

```java
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // Head ကိုယ်တိုင် ဖျက်ရနိုင်သဖြင့် Dummy Node သုံးသည်
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode slow = dummy;
        ListNode fast = dummy;

        // fast ကို n+1 လှမ်း ရှေ့သွားစေသည်
        for (int i = 0; i <= n; i++) {
            fast = fast.next;
        }

        // နှစ်ခုစလုံး အတူတူ ရွှေ့သည်၊ fast အဆုံးရောက်လျှင် ရပ်သည်
        while (fast != null) {
            slow = slow.next;
            fast = fast.next;
        }

        // slow ၏ နောက်က Node ကို ကျော်၍ ဖျက်သည်
        slow.next = slow.next.next;

        return dummy.next;
    }
}
```

အခန်း ၉ - Binary Search

Binary Search (တစ်ဝက်စီ ခွဲခြမ်းရှာဖွေခြင်း) ဆိုတာ **Sorted (စီထားပြီးသား)** ဖြစ်နေတဲ့ linear data structure (ဥပမာ - Array) တစ်ခုထဲကနေ ကိုယ်လိုချင်တဲ့ တန်ဖိုး (Target) ကို အလွန် လျင်မြန်စွာ ရှာဖွေနိုင်တဲ့ **နည်းလမ်း** တစ်ခု ဖြစ်ပါတယ်။

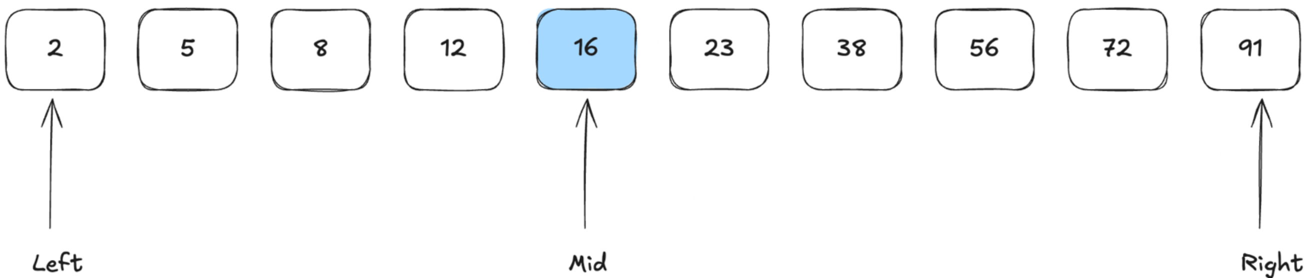
ပုစ္ဆာတွေမှာ အချိန်ကြာမြင့်မှု (Time Complexity) ကို $O(\log n)$ အထိ လျှော့ချချင်တဲ့အခါ Binary Search ကို မဖြစ်မနေ သုံးကြရပါတယ်။

Binary Search ၏ အဓိက အယူအဆ

Binary Search ရဲ့ အလုပ်လုပ်ပုံကို နားလည်ဖို့ **စီထားပြီးသား Array တစ်ခုထဲမှာ target ရှာတာ** ကို တိုက်ရိုက်ကြည့်ရအောင်။

ဥပမာ `nums = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]` ဖြစ်ပြီး `target = 23` ကို ရှာမယ်ဆိုပါစို့။

Binary Search မှာ `left` နဲ့ `right` က လက်ရှိရှာနေတဲ့ နယ်ပယ်ကို သတ်မှတ်ပါတယ်။ `mid` က အဲ့ဒီ နယ်ပယ်ရဲ့ အလယ်နေရာပါ။ `nums[mid]` ကို `target` နဲ့နှိုင်းယှဉ်ပြီး target မရှိနိုင်တဲ့ တစ်ဝက်ကို တစ်ခါတည်း ဖယ်ထုတ်သွားပါတယ်။



အဆင့်လိုက် ကြည့်ပါ။

အဆင့်	ရှာနေသည့်နယ်ပယ်	`mid` တန်ဖိုး	နှိုင်းယှဉ်ချက်	ဆက်လုပ်မည့်အရာ
၁	`[2, 5, 8, 12, 16, 23, 38, 56, 72, 91]`	`16`	`16 < 23`	`23` က ပိုကြီးလို့ ဘယ်ဘက်တစ်ဝက် `[2, 5, 8, 12, 16]` ကို ဖယ်
၂	`[23, 38, 56, 72, 91]`	`56`	`56 > 23`	`23` က ပိုငယ်လို့ ညာဘက်တစ်ဝက် `[56, 72, 91]` ကို ဖယ်
၃	`[23, 38]`	`23`	`23 == 23`	တွေ့ပြီ၊ index `5` ကို return ပြန်

ဒီ table မှာ အဓိက သတိထားရမယ့်အချက်က **တစ်ကြိမ်စစ်ပြီးတိုင်း ရှာစရာနယ်ပယ်က တစ်ဝက်လျော့သွားတာ** ပါ။

ပထမအဆင့်မှာ `16` ကို စစ်ပါတယ်။ `target = 23` က `16` ထက် ပိုကြီးတဲ့အတွက် `16` အပါအဝင် ဘယ်ဘက်ခြမ်းမှာ target မရှိနိုင်တော့ပါဘူး။ ဒါကြောင့် ဘယ်ဘက်တစ်ဝက်ကို တစ်ခါတည်း ဖယ်ပါတယ်။

ဒုတိယအဆင့်မှာ `56` ကို စစ်ပါတယ်။ `target = 23` က `56` ထက် ပိုငယ်တဲ့အတွက် `56` အပါအဝင် ညာဘက်ခြမ်းမှာ target မရှိနိုင်တော့ပါဘူး။ ဒါကြောင့် ညာဘက်တစ်ဝက်ကို ထပ်ဖယ်ပါတယ်။

နောက်ဆုံး ကျန်တဲ့နယ်ပယ် `[23, 38]` ထဲမှာ `23` ကို စစ်လိုက်တဲ့အခါ target ကို တွေ့သွားပါတယ်။

အခြေခံဆုံး စည်းမျဉ်းကို ဒီလို မှတ်နိုင်ပါတယ်။

အခြေအနေ	ဘာလုပ်မလဲ
`nums[mid] == target`	အဖြေတွေ့ပြီ၊ `mid` ကို return ပြန်
`nums[mid] < target`	target က ညာဘက်မှာပဲ ရှိနိုင်လို့ `left = mid + 1`
`nums[mid] > target`	target က ဘယ်ဘက်မှာပဲ ရှိနိုင်လို့ `right = mid - 1`

$O(\log n)$ ၏ အစွမ်းထက်ပုံ

အချက်အလက် အရေအတွက်ဟာ မည်မျှပင် များပြားပါစေ၊ Binary Search ဟာ တစ်ကြိမ်လျှင် ရှာဖွေရမည့် နယ်ပယ်ကို ထက်ဝက်တိတိ လျှော့ချသွားနိုင်တဲ့အတွက် အလွန် အစွမ်းထက်ပါတယ်။

- အချက်အလက် အရေအတွက် **၁,၀၀၀** (တစ်ထောင်) ရှိလျှင် အဆိုးဆုံး **၁၀ ကြိမ်**သာ ရှာရန် လိုအပ်သည်။
- အချက်အလက် အရေအတွက် **၁,၀၀၀,၀၀၀** (တစ်သန်း) ရှိလျှင် အဆိုးဆုံး **၂၀ ကြိမ်**သာ ရှာရန် လိုအပ်သည်။
- အချက်အလက် အရေအတွက် **၄,၀၀၀,၀၀၀,၀၀၀** (လေးဘီလီယံ) ရှိလျှင် အဆိုးဆုံး **၃၂ ကြိမ်**သာ ရှာရန် လိုအပ်သည်။

Binary Search တွင် သိထားရမည့် အခြေခံစည်းမျဉ်းများ

Binary Search ကို မှန်ကန်စွာ ရေးသားနိုင်ဖို့ အဓိက အချက် ၄ ချက်ကို နားလည်ထားရပါမယ်။

၁။ Pointers သုံးခု သတ်မှတ်ခြင်း

ရှာဖွေရမည့် ဘောင်ကို သတ်မှတ်ရန် `left` (အစ) နှင့် `right` (အဆုံး) pointer နှစ်ခု လိုအပ်ပြီး၊ နှိုင်းယှဉ်ရန် `mid` (အလယ်) pointer တစ်ခု လိုအပ်ပါသည်။

- `left` $= 0$
- `right` $= length - 1$

၂။ Integer Overflow ကို ရှောင်လွှဲခြင်း (Critical Tip)

အလယ်ကိန်း `mid` ကို ရှာတဲ့အခါ ပုံမှန်အားဖြင့် အောက်ပါအတိုင်း တွက်လေ့ရှိကြပါတယ် —

```
int mid = (left + right) / 2;
```

ဒီနည်းလမ်းက သာမန်အချိန်မှာ အလုပ်လုပ်ပေမယ့် `left` ရော `right` ပါ အလွန်ကြီးမားတဲ့ ကိန်းဂဏန်းတွေ ဖြစ်နေရင် (ဥပမာ - 2^{30} ဝန်းကျင်)၊ နှစ်ခုပေါင်းလိုက်တဲ့အခါ `int` ရဲ့ အမြင့်ဆုံး သတ်မှတ်ချက်ထက် ကျော်လွန်ပြီး **Integer Overflow** ဖြစ်သွားစေနိုင်ပါတယ်။ ဒါကြောင့် mid ကို ရှာသည့် အခါမှာ အောက်ပါအတိုင်း ရေးသားကြပါတယ်။

```
int mid = left + (right - left) / 2;
```

၃။ Pointer များကို ရွှေ့လျားပုံ (Boundary Update)

`mid` မှာ ရှိတဲ့ တန်ဖိုးနဲ့ `target` ကို နှိုင်းယှဉ်ပြီး ဘောင်တွေကို ကျဉ်းသွားစေရပါမယ်။

- `nums[mid] == target` ဖြစ်လျှင် အဖြေတွေ့ပြီဖြစ်၍ `mid` ကို return ပြန်သည်။
- `nums[mid] < target` ဖြစ်လျှင် target သည် ညာဘက်ခြမ်းတွင်သာ ရှိနိုင်၍ ဘယ်ဘက်ဘောင်ကို ရွှေ့သည်: `left = mid + 1`
- `nums[mid] > target` ဖြစ်လျှင် target သည် ဘယ်ဘက်ခြမ်းတွင်သာ ရှိနိုင်၍ ညာဘက်ဘောင်ကို ရွှေ့သည်: `right = mid - 1`

၄။ Infinite Loop မဖြစ်စေရန် (Common Pitfall)

Binary Search ရေးတဲ့အခါ အများဆုံး မှားတတ်တာက **loop က ဘယ်တော့မှ မရပ်ဘဲ infinite loop ဖြစ်သွားတာ** ပါ။ မှတ်ထားရမယ့် အချက် ၂ ချက် —

- **`left <= right` (target ရှာတဲ့ template):** boundary ကို `mid + 1` / `mid - 1` နဲ့ **အမြဲ ရွှေ့** ပေးရမယ်။ `left = mid` ဒါမှမဟုတ် `right = mid` လို့ ရေးမိရင် `mid` က နေရာ မပြောင်းတော့ဘဲ loop ထဲ ထိုင်ကျန်သွားပါတယ်။
- **`left < right` (boundary/lower bound ရှာတဲ့ template):** `right = mid` သုံးလို့ ရပေမယ့်၊ `left = mid` တော့ မသုံးရပါဘူး။ ဘာကြောင့်လဲဆိုတော့ element ၂ ခု ကျန်တဲ့အခါ `mid` က `left` ဖြစ်နေပြီး `left = mid` က နေရာ မပြောင်းလို့ infinite loop ဖြစ်စေပါတယ်။ ဒီအခါ `mid` ကို `left + (right - left + 1) / 2` (အပေါ်ကို round) လုပ်ပြီး ဖြေရှင်းရပါတယ်။

Lower Bound နှင့် Upper Bound

တန်ဖိုးတစ်ခုကို ရှာဖို့ထက် **"ဘယ်နေရာမှာ ထည့်သင့်သလဲ"** ဆိုတဲ့ boundary ကို ရှာရတာ ပိုအသုံးဝင်ပါတယ်။ ဒါက Binary Search ရဲ့ အစွမ်းအထက်ဆုံး အသုံးချမှုပါ။

- **Lower Bound** — target ထက် **ငယ်ခြင်းမရှိ (>= target)** တဲ့ ပထမဆုံး နေရာ (index)။
- **Upper Bound** — target ထက် **ကြီးတဲ့ (> target)** ပထမဆုံး နေရာ (index)။

ဥပမာ `nums = [1, 2, 2, 2, 5]` , `target = 2` ဆိုရင် —

- Lower bound = index `1` (ပထမဆုံး `2`)
- Upper bound = index `4` (ပထမဆုံး `2` ထက်ကြီးတဲ့ `5`)

`target` ရှာတဲ့ template နဲ့ မတူတာက — တွေ့တာနဲ့ ချက်ချင်း မရပ်ဘဲ၊ ဖြစ်နိုင်ချေ အဖြေကို မှတ်ထားပြီး ဆက် သွားတာပါ။

```
// Lower bound: nums[i] >= target ဖြစ်တဲ့ ပထမဆုံး index
int lowerBound(int[] nums, int target) {
    int left = 0, right = nums.length; // right က length (out of range) က စသည်
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;   // mid က သေးနေသေးလို့ ညာဘက်
        } else {
            right = mid;      // mid ကိုယ်တိုင် အဖြေ ဖြစ်နိုင်လို့ ဖယ်မပစ်
        }
    }
    return left;
}
```

Upper bound အတွက်ဆို `nums[mid] < target` ကို `nums[mid] <= target` လို့ ပြောင်းရုံပါပဲ။ ဒီ pattern နှစ်ခုက အောက်က **Search Insert Position** နဲ့ **First and Last Position** ပုစ္ဆာတွေရဲ့ အခြေခံပါ။

Real-world မှာ ဘယ်လိုသုံးလဲ

Binary Search က "**sorted ဖြစ်နေတဲ့ အရာ**" ဒါမှမဟုတ် "**monotonic (တစ်လမ်းသွား တိုး/လျော့) ဖြစ်တဲ့ အဖြေနယ်ပယ်**" ရှိတိုင်း သုံးနိုင်ပါတယ်။

- **Sorted records ရှာခြင်း** — Database index (B-Tree) တွေဟာ sorted key ပေါ်မှာ binary search သဘောတရားနဲ့ အလုပ်လုပ်တယ်။ `git bisect` က bug ဝင်တဲ့ commit ကို binary search နဲ့ ရှာတယ်။
- **First available version / capacity** — "condition မှန်တဲ့ အငယ်ဆုံးတန်ဖိုး" ရှာတာ — server အနည်းဆုံး ဘယ်နှလုံး လိုသလဲ၊ package version ဘယ်ကစပြီး bug ဝင်သလဲ (lower bound pattern)။
- **Price range / log timestamp** — sorted price list ဒါမှမဟုတ် time-series log ထဲက အပိုင်း (range) ကို lower/upper bound နဲ့ ဆွဲထုတ်တာ။
- **Minimum capacity / rate (Binary Search on Answer)** — အောက်က Koko Eating Bananas လို "အနည်းဆုံး k ဘယ်လောက် လိုသလဲ" ပုစ္ဆာမျိုး။

Questions

Binary Search ကို ကောင်းစွာ အသုံးချတတ်စေဖို့ အမေးအများဆုံး မေးခွန်းပုံစံတွေကို လွယ်ရာက ခက်ရာ တစ်ဆင့်ချင်း လေ့လာကြည့်ရအောင်။

၁။ Binary Search (Basic Template)

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

ရှင်းလင်းချက်

ဒါကတော့ Binary Search ရဲ့ အခြေခံအကျဆုံး Template ပုစ္ဆာ ဖြစ်ပါတယ်။ ပေးထားတဲ့ Array က အငယ်ကနေ အကြီး အစဉ်လိုက် စီပြီးသား ဖြစ်တဲ့အတွက် `left` နဲ့ `right` pointer တွေကို သုံးပြီး target ကို ရှာဖွေသွားရမှာ ဖြစ်ပါတယ်။ `left <= right` ဖြစ်နေသရွေ့ ပတ်ပါမယ်။

> **Time Complexity:** $O(\log n)$ - အဆင့်တိုင်းမှာ search space က ထက်ဝက် လျော့သွားလို့ ဖြစ်ပါတယ်။
> **Space Complexity:** $O(1)$ - memory ပို ထပ်မယူလို့ ဖြစ်ပါတယ်။

Java Solution

```java
class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            // Integer Overflow မဖြစ်စေရန် တွက်နည်း
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid; // အဖြေရှာတွေ့ပါက index ကို ပြန်သည်
            } else if (nums[mid] < target) {
```

```
                // target က ပိုကြီးနေလျှင် ညာဘက်ခြမ်းတွင် ဆက်ရှာရန်
                left = mid + 1;
            } else {
                // target က ပိုငယ်နေလျှင် ဘယ်ဘက်ခြမ်းတွင် ဆက်ရှာရန်
                right = mid - 1;
            }
        }

        return -1; // ရှာမတွေ့ပါက -1 ပြန်သည်
    }
}
```

၂။ Search Insert Position

Sorted array `nums` (distinct values) နဲ့ `target` ပေးထားတယ်။ target ရှိရင် သူ့ index ကို ပြန်ပါ။ မရှိရင် **target ကို စီထားတဲ့ အစဉ်အတိုင်း ထည့်သင့်တဲ့ index** ကို ပြန်ပါ။

Example:

```
Input: nums = [1,3,5,6], target = 5   → Output: 2
Input: nums = [1,3,5,6], target = 2   → Output: 1
Input: nums = [1,3,5,6], target = 7   → Output: 4
```

ရှင်းလင်းချက်

ဒါက အပေါ်က **Lower Bound** ကို တိုက်ရိုက် အသုံးချတာပါ။ `nums[i] >= target` ဖြစ်တဲ့ ပထမဆုံး index ဟာ — target ရှိရင် သူ့နေရာ၊ မရှိရင် ထည့်သင့်တဲ့နေရာ — အတိအကျ ဖြစ်နေပါတယ်။

> **Time Complexity:** $O(\log n)$ **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public int searchInsert(int[] nums, int target) {
        int left = 0;
        int right = nums.length; // out-of-range index ကစသည် (ထည့်စရာ နေရာ ဖြစ်နိုင်လို့)

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid; // mid ကိုယ်တိုင် အဖြေ ဖြစ်နိုင်လို့ ဖယ်မပစ်
            }
        }

        return left;
    }
}
```

၃။ Find First and Last Position of Element

Sorted array `nums` ထဲမှာ `target` ရဲ့ **ပထမဆုံး (first)** နဲ့ **နောက်ဆုံး (last)** index ကို ရှာပါ။ မရှိရင် `[-1, -1]` ။ $O(\log n)$ ဖြစ်ရမယ်။

Example:

```
Input: nums = [5,7,7,8,8,10], target = 8   →  Output: [3,4]
Input: nums = [5,7,7,8,8,10], target = 6   →  Output: [-1,-1]
```

ရှင်းလင်းချက်

target ထပ်နေနိုင်တဲ့အတွက် binary search ၂ ခါ လုပ်ပါမယ် —

- **First position** = `target` ရဲ့ **lower bound** (`>= target` ပထမဆုံး index)။
- **Last position** = `target` ရဲ့ **upper bound** ထက် တစ်လုံး နောက်ပြန် (`> target` ပထမဆုံး index ရဲ့ ရှေ့)။

> **Time Complexity:** $O(\log n)$ **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int first = lowerBound(nums, target);
        // target မရှိလျှင်
        if (first == nums.length || nums[first] != target) {
            return new int[] { -1, -1 };
        }
        int last = lowerBound(nums, target + 1) - 1; // target+1 ၏ lower bound ၏ ရှေ့
        return new int[] { first, last };
    }

    // nums[i] >= key ဖြစ်တဲ့ ပထမဆုံး index
    private int lowerBound(int[] nums, int key) {
        int left = 0, right = nums.length;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < key) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
}
```

၄။ Search a 2D Matrix

You are given an $m \times n$ integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.

- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target` , return `true` if `target` is in `matrix` or `false` otherwise.

You must write a solution in $O(\log(m \times n))$ time complexity.

Example 1:

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
```

ရှင်းလင်းချက်

ဒီပုစ္ဆာမှာ Matrix (2D Array) တစ်ခု ပေးထားပေမယ့် ထူးခြားချက်က Row တစ်ခုချင်းစီဟာ စီထားပြီးသား ဖြစ်ရုံသာမက၊ နောက် Row ရဲ့ ပထမဆုံးဂဏန်းက အရှေ့ Row ရဲ့ နောက်ဆုံးဂဏန်းထက် အမြဲ ပိုကြီးနေပါတယ်။ ဒါဟာ တကယ်တော့ 2D Matrix သာ ဖြစ်နေပေမယ့် ဂဏန်းအားလုံးကို တန်းစီချလိုက်ရင် **1D Sorted Array တစ်ခုတည်း** အတိုင်းပါပဲ။

ဒါကြောင့် 2D Matrix ကို 1D Array တစ်ခုလို စိတ်ကူးပုံဖော်ပြီး Binary Search သုံးနိုင်ပါတယ်။

- 1D Array ရဲ့ အလျား (Length) = $m \times n$ ဖြစ်ပါမယ်။
- Virtual 1D index `mid` ကို 2D Coordinates (Row, Col) အဖြစ် ပြန်ပြောင်းနည်း-
  - `row = mid / n` (ညာဘက် Column အရေအတွက်နဲ့ စားသည်)
  - `col = mid % n` (ညာဘက် Column အရေအတွက်နဲ့ စားကြွင်းရှာသည်)

> **Time Complexity:** $O(\log(m \times n))$ - 2D matrix တစ်ခုလုံးကို 1D array လို တစ်ကြိမ်တည်းနဲ့ ရှာဖွေနိုင်သောကြောင့် ဖြစ်သည်။
> **Space Complexity:** $O(1)$

Java Solution

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return false;
        }

        int rows = matrix.length;
        int cols = matrix[0].length;

        // 1D Array သဘောမျိုး အစ နှင့် အဆုံး သတ်မှတ်သည်
        int left = 0;
        int right = (rows * cols) - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Virtual 1D index 'mid' မှ 2D Row, Col သို့ ပြောင်းလဲခြင်း
            int midVal = matrix[mid / cols][mid % cols];
```

```
                if (midVal == target) {
                    return true;
                } else if (midVal < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            return false;
        }
    }
```

၅။ Koko Eating Bananas

Koko loves to eat bananas. There are `n` piles of bananas, the ith pile has `piles[i]` bananas. The guards have gone and will come back in `h` hours.

Koko can decide her bananas-per-hour eating speed of `k`. Each hour, she chooses some pile of bananas and eats `k` bananas from that pile. If the pile has less than `k` bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the minimum integer `k` such that she can eat all the bananas within `h` hours.

Example 1:

```
Input: piles = [3,6,7,11], h = 8
Output: 4
```

ရှင်းလင်းချက်

ဒီပုစ္ဆာက **"Binary Search on Answer / Solution Space"** လို့ ခေါ်တဲ့ အဆင့်မြင့် ပုံစံတစ်မျိုး ဖြစ်ပါတယ်။ ကျွန်တော်တို့က Array ထဲမှာ တန်ဖိုးတစ်ခုကို လိုက်ရှာတာ မဟုတ်ပါဘူး။ Koko ငှက်ပျောသီး စားရမယ့် **အနည်းဆုံးနှုန်း `k` (တစ်နာရီ စားမယ့် အရေအတွက်)** ကို စဉ်းစားရမှာ ဖြစ်ပါတယ်။

- တစ်နာရီ စားနိုင်မယ့် အနည်းဆုံးနှုန်း `k` က **၁** ဖြစ်ပါတယ်။
- တစ်နာရီ စားနိုင်မယ့် အများဆုံးနှုန်း `k` ကတော့ ပေးထားတဲ့ စုပုံတွေထဲက အများဆုံး အရေအတွက် `max(piles)` ဖြစ်ပါတယ် (ဘာဖြစ်လို့လဲဆိုတော့ တစ်နာရီကို စုပုံတစ်ပုံထက် ပိုမစားနိုင်လို့ အကြီးဆုံးပုံထက် ပိုမြန်အောင် စားဖို့ မလိုပါဘူး)။
- ဒါကြောင့် ကျွန်တော်တို့ရဲ့ ရှာဖွေရမယ့် ဘောင်က `[1, max(piles)]` ဖြစ်သွားပါပြီ။

- အလယ်ကိန်း `mid` ကို ယူပြီး တစ်နာရီ `mid` နှုန်းနဲ့ စားရင် စုစုပေါင်း ဘယ်နှနာရီ ကြာမလဲလို့ တွက်ကြည့်ပါမယ်။
    - ကြာမယ့်အချိန်က `h` နာရီထက် ငယ်ရင် သို့မဟုတ် ညီရင် (သတ်မှတ်ချိန်အတွင်း ပြီးရင်) - ဒီထက်ပိုနှေးပြီး စားလို့ ရမရ ဆက်စစ်ဖို့ ညာဘက်ဘောင်ကို ကျဉ်းပါမယ်: `right = mid - 1` (အဖြေကို `mid` အဖြစ် မှတ်ထားပါမယ်)။
    - ကြာမယ့်အချိန်က `h` ထက် ကျော်သွားရင် (သတ်မှတ်ချိန်အတွင်း မပြီးရင်) - ပိုမြန်မြန် စားဖို့ လိုတဲ့အတွက် ဘယ်ဘက်ဘောင်ကို တိုးပါမယ်: `left = mid + 1` ။

> တစ်ပုံချင်းစီ စားတဲ့အခါ ကြာမယ့်အချိန်ကို `Math.ceil((double)pile / k)` သို့မဟုတ် Integer arithmetic သုံးပြီး `(pile + k - 1) / k` ဖြင့် ရှာနိုင်ပါတယ်။

> **Time Complexity:** $O(n \log(\max(piles)))$ - binary search ဘောင် `[1, max(piles)]` ကို $O(\log(\max(piles)))$ အကြိမ် ပတ်ပြီး၊ အဆင့်တိုင်းမှာ $O(n)$ အချိန်ယူ၍ piles အားလုံး စားချိန်ကို တွက်သောကြောင့် ဖြစ်သည်။
> **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public int minEatingSpeed(int[] piles, int h) {
        int left = 1;
        int right = 0;

        // အများဆုံး စားနှုန်းကို ရှာရန် Piles ထဲမှ အကြီးဆုံးတန်ဖိုးကို ရှာသည်
        for (int pile : piles) {
            right = Math.max(right, pile);
        }

        int result = right; // အဆိုးဆုံးနှုန်းဖြင့် စတင်ထားမည်

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // တစ်နာရီ 'mid' နှုန်းဖြင့် စားပါက စုစုပေါင်း ကြာမည့်နာရီ
            long totalHours = 0;
            for (int pile : piles) {
                // (pile + mid - 1) / mid သည် Math.ceil((double)pile / mid) နှင့် အတူတူပင်
ဖြစ်သည်
                totalHours += (pile + mid - 1) / mid;
            }

            if (totalHours <= h) {
                // သတ်မှတ်ချိန်မီပါက ပိုနှေးသော စားနှုန်းကို ဆက်ရှာရန်
                result = mid;
                right = mid - 1;
            } else {
                // သတ်မှတ်ချိန်မမီပါက ပိုမြန်မြန် စားရန်
                left = mid + 1;
            }
        }
```

```
            return result;
        }
    }
```

၆။ Find Minimum in Rotated Sorted Array

Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated `4` times.
- `[0,1,2,4,5,6,7]` if it was rotated `7` times.

Notice that rotating an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]` .

Given the sorted rotated array `nums` of **unique** elements, return the minimum element of this array.

You must write an algorithm that runs in $O(\log n)$ time.

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

ရှင်းလင်းချက်

Sorted ဖြစ်နေတဲ့ Array တစ်ခုကို တစ်နေရာရာကနေ ဖြတ်ပြီး ခေါက်လိုက်တဲ့အခါ (Rotated sorted array)၊ Array ရဲ့ ပုံစံဟာ အပိုင်းနှစ်ပိုင်း ကွဲသွားပါတယ်။
အဓိက အယူအဆကတော့ — **အပိုင်းတစ်ခုခုသည် အမြဲတမ်း စနစ်တကျ စီထားလျက် ရှိနေမည်** ဖြစ်ပြီး၊ အခြားအပိုင်းတွင်မူ လှည့်ခေါက်ထားသော Pivot Point (အနည်းဆုံးတန်ဖိုး) ရှိနေပါလိမ့်မယ်။

- `nums[mid]` တန်ဖိုးကို ညာဘက်ဆုံးတန်ဖိုး `nums[right]` နဲ့ တိုက်စစ်ပါမယ်။
- `nums[mid] > nums[right]` ဖြစ်နေလျှင် - ဆိုလိုတာက ဘယ်ဘက်ခြမ်းကနေ အလယ်အထိဟာ ပုံမှန်အတိုင်း တိုးလာနေပေမယ့် ညာဘက်အဆုံးထက် ကြီးနေတာမို့လို့ အလှည့် Pivot Point (သို့) အငယ်ဆုံးကိန်းဟာ `mid` ရဲ့ ညာဘက်ခြမ်းမှာပဲ ရှိနိုင်ပါတော့တယ်။ ဒါကြောင့် `left = mid + 1` ။
- `nums[mid] <= nums[right]` ဖြစ်နေလျှင် - ဆိုလိုတာက `mid` ကနေ ညာဘက်ခြမ်းတစ်ခုလုံးက ပုံမှန်အတိုင်း စီပြီးသား ဖြစ်နေလို့ အငယ်ဆုံးကိန်းက `mid` ကိုယ်တိုင် သို့မဟုတ် `mid` ရဲ့ ဘယ်ဘက်ခြမ်းမှာပဲ ရှိပါလိမ့်မယ်။ ဒါကြောင့် `right = mid` ။

> ဤနေရာတွင် `left < right` ဟု သုံးရပါမည်။ ဘာကြောင့်လဲဆိုတော့ `right = mid` ဟု ပြောင်းလဲချိန်တွင် `mid` သည် အဖြေဖြစ်နိုင်သောကြောင့် loop မပတ်မိစေရန် ဖြစ်သည်။

> **Time Complexity:** $O(\log n)$
> **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while (left < right) {
            int mid = left + (right - left) / 2;

            // အလယ်ကိန်းသည် ညာဘက်အစွန်းကိန်းထက် ကြီးနေလျှင် Pivot သည် ညာဘက်တွင်ရှိသည်
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            } else {
                // အလယ်ကိန်းသည် ညာဘက်အစွန်းထက် ငယ်/ညီ နေပါက Pivot သည် ဘယ်ဘက်တွင် (သို့)
mid ကိုယ်တိုင်ဖြစ်သည်
                right = mid;
            }
        }

        // left နှင့် right ဆုံသည့် နေရာသည် အနည်းဆုံးတန်ဖိုးဖြစ်သည်
        return nums[left];
    }
}
```

၇။ Search in Rotated Sorted Array

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is possibly rotated at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]` .

Given the array `nums` **after** the possible rotation and an integer `target` , return the index of `target` if it is in `nums` , or `-1` if it is not in `nums` .

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

ရှင်းလင်းချက်

ဒီပုစ္ဆာက "Find Minimum in Rotated Sorted Array" အပေါ် ထပ်ဆင့်ပြီး target ကို ရှာခိုင်းတာ ဖြစ်ပါတယ်။ တစ်ဝက်ခွဲလိုက်တဲ့အခါ ဘယ်ဘက်ခြမ်း သို့မဟုတ် ညာဘက်ခြမ်း တစ်ဖက်ဖက်ဟာ **အမြဲတမ်း ပုံမှန် Sorted ဖြစ်နေဆဲ** ဆိုတဲ့ အချက်ကို အခြေခံပြီး ဖြေရှင်းရပါမယ်။

၁။ အရင်ဆုံး ဘယ်ဘက်ခြမ်း `[left, mid]` က ပုံမှန် Sorted ဖြစ်နေသလား သိဖို့ `nums[left] <= nums[mid]` ကို စစ်ဆေးပါတယ်။

- Sorted ဖြစ်နေလျှင် - target သည် အဲ့ဒီဘယ်ဘက်ခြမ်း ဘောင် `[nums[left], nums[mid])` ထဲမှာ ရှိမရှိ စစ်သည်။ ရှိလျှင် `right = mid - 1` ၊ မရှိလျှင် ညာဘက်ခြမ်းမှာ ဆက်ရှာရန် `left = mid + 1` ။

၂။ အကယ်၍ ဘယ်ဘက်ခြမ်းက Sorted မဟုတ်ပါက၊ ညာဘက်ခြမ်း `(mid, right]` က မလွဲမသွေ ပုံမှန် Sorted ဖြစ်နေပါလိမ့်မယ်။

- target သည် ညာဘက်ခြမ်း ဘောင် `(nums[mid], nums[right]]` ထဲမှာ ရှိမရှိ စစ်သည်။ ရှိလျှင် `left = mid + 1` ၊ မရှိလျှင် ဘယ်ဘက်ခြမ်းမှာ ဆက်ရှာရန် `right = mid - 1` ။

> **Time Complexity:** $O(\log n)$
> **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            }

            // ဘယ်ဘက်ခြမ်းသည် ပုံမှန် Sorted ဖြစ်နေသလား စစ်ဆေးသည်
            if (nums[left] <= nums[mid]) {
                // target သည် ဘယ်ဘက်ခြမ်း၏ ဘောင်ထဲတွင် ရှိနေသလား
                if (target >= nums[left] && target < nums[mid]) {
                    right = mid - 1; // ဘယ်ဘက်သို့ ရွှေ့သည်
                } else {
                    left = mid + 1;  // ညာဘက်သို့ ရွှေ့သည်
                }
            }
            // မဟုတ်ပါက ညာဘက်ခြမ်းသည် ပုံမှန် Sorted ဖြစ်နေရမည်
            else {
                // target သည် ညာဘက်ခြမ်း၏ ဘောင်ထဲတွင် ရှိနေသလား
                if (target > nums[mid] && target <= nums[right]) {
                    left = mid + 1;  // ညာဘက်သို့ ရွှေ့သည်
                } else {
                    right = mid - 1; // ဘယ်ဘက်သို့ ရွှေ့သည်
                }
            }
        }
```

```
            return -1;
        }
    }
```

၈။ Time Based Key-Value Store

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the `TimeMap` class:

- `TimeMap()` Initializes the object of the data structure.
- `void set(String key, String value, int timestamp)` Stores the key `key` with the value `value` at the given time `timestamp` .
- `String get(String key, int timestamp)` Returns a value such that `set` was called previously, with `timestamp_prev <= timestamp` . If there are multiple such values, it returns the value associated with the largest `timestamp_prev` . If there are no values, it returns `""` .

Example 1:

```
Input:
["TimeMap", "set", "get", "get", "set", "get", "get"]
[[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]
Output:
[null, null, "bar", "bar", null, "bar2", "bar2"]
```

ရှင်းလင်းချက်

ဒီပုစ္ဆာကတော့ Database တွေမှာ Version ထိန်းချုပ်မှု (Timestamp tracking) အတွက် သုံးတဲ့ ပုံစံမျိုး ဖြစ်ပါတယ်။

- `set` လုပ်တဲ့အခါ `key` တစ်ခုတည်းအတွက် `value` တွေနဲ့ ဝင်လာတဲ့ `timestamp` တွေကို တွဲပြီး သိမ်းသွားရပါမယ်။ (ဥပမာ- `Map<String, List<Pair>>`)။
- `get` လုပ်တဲ့အခါ `timestamp` ပေးထားပြီး ထို timestamp ထက် ငယ်သော သို့မဟုတ် ညီသော **အကြီးဆုံး timestamp_prev (Floor value)** ရဲ့ တန်ဖိုးကို ရှာပေးရပါမယ်။
- ပေးထားချက်အရ `set` ရဲ့ timestamp တွေဟာ အမြဲတမ်း တိုးလာနေတဲ့အတွက် (Strictly increasing)၊ သက်ဆိုင်ရာ `key` ရဲ့ List ထဲမှာ Binary Search သုံးပြီး အလွယ်တကူ ရှာဖွေနိုင်ပါတယ်။

> **Time Complexity:**
> - `set` : $O(1)$
> - `get` : $O(\log n)$ (ပေးထားသော Key အတွက် list အရွယ်အစား n ပေါ်မူတည်၍)
>
> **Space Complexity:** $O(total_entries)$ - entries အားလုံးကို map ထဲတွင် သိမ်းဆည်းသောကြောင့် ဖြစ်သည်။

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class TimeMap {
    // Value နှင့် Timestamp ကို တွဲသိမ်းရန် Class တည်ဆောက်သည်
    private static class Node {
        String value;
        int timestamp;

        Node(String value, int timestamp) {
            this.value = value;
            this.timestamp = timestamp;
        }
    }

    private Map<String, List<Node>> map;

    public TimeMap() {
        map = new HashMap<>();
    }

    public void set(String key, String value, int timestamp) {
        map.putIfAbsent(key, new ArrayList<>());
        map.get(key).add(new Node(value, timestamp));
    }

    public String get(String key, int timestamp) {
        if (!map.containsKey(key)) {
            return "";
        }

        List<Node> list = map.get(key);
        int left = 0;
        int right = list.size() - 1;
        String result = "";

        // Floor value ရှာရန် Binary Search
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (list.get(mid).timestamp <= timestamp) {
                // လက်ရှိ timestamp ထက် ငယ်/ညီပါက ဖြစ်နိုင်ချေရှိသောအဖြေအဖြစ် မှတ်ပြီး
                // ပိုကြီးသည့်အဖြေရှိနိုင်သေးသဖြင့် ညာဘက်သို့ ဆက်ရှာသည်
                result = list.get(mid).value;
                left = mid + 1;
            } else {
```

```
                // timestamp ကြီးနေပါက ဘယ်ဘက်သို့ ရွှေ့သည်
                right = mid - 1;
            }
        }

        return result;
    }
}
```

၉။ Median of Two Sorted Arrays

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be $O(\log(m+n))$.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

ရှင်းလင်းချက်

ဒါကတော့ Binary Search အခန်းရဲ့ အခက်ခဲဆုံးနဲ့ အဆင့်အမြင့်ဆုံး **Hard** အဆင့်ရှိ မေးခွန်း ဖြစ်ပါတယ်။ ပုံမှန်အားဖြင့် Arrays နှစ်ခုကို ပေါင်းပြီး Sort လုပ်ရင် $O(m+n)$ ကြာပေမယ့်၊ ဒီပုစ္ဆာက $O(\log(m+n))$ သာ ကြာအောင် စွမ်းဆောင်ရမှာ ဖြစ်ပါတယ်။

ဒါကို ဖြေရှင်းဖို့အတွက် **"Partition (ပိုင်းခြားခြင်း) နည်းလမ်း"** ကို သုံးရပါမယ်။
ကျွန်တော်တို့ လိုချင်တဲ့ အလယ်တန်ဖိုး (Median) ဆိုတာ စုစုပေါင်း Array ၂ ခုကို ဘယ်ခြမ်း (Left Partition) နဲ့ ညာခြမ်း (Right Partition) လို့ ညီတူညီမျှ ခွဲလိုက်တဲ့အခါ -

- Left Partition မှာ ရှိသမျှ ဂဏန်းတွေအားလုံးက Right Partition က ဂဏန်းတွေအားလုံးထက် ငယ်သို့မဟုတ် ညီ နေရပါမယ်။
- Left partition ၏ အရွယ်အစားသည် `(m + n + 1) / 2` ဖြစ်ရပါမယ်။

```
Array A (nums1):  [ ... A[i-1] | A[i] ... ]
Array B (nums2):  [ ... B[j-1] | B[j] ... ]
                  <-- Left --> | <-- Right -->
```

ကျွန်တော်တို့က အရွယ်အစား ပိုငယ်တဲ့ Array (ဥပမာ A) ပေါ်မှာ Binary Search စလုပ်ပါမယ်။

- အကယ်၍ A ထဲက Partition index `i` ကို ရွေးလိုက်ရင်၊ B ထဲက Partition index `j` က အလိုအလျောက် သတ်မှတ်ပြီးသား ဖြစ်သွားပါတယ်: `j = (m + n + 1) / 2 - i` ။
- ပြီးရင် ဖြတ်တောက်လိုက်တဲ့ နယ်နိမိတ် ၄ ခုဖြစ်တဲ့ `A[i-1]` , `A[i]` , `B[j-1]` , `B[j]` တို့ကို စစ်ဆေးပါမယ်။
    - **မှန်ကန်သော အခြေအနေ:** `A[i-1] <= B[j]` နှင့် `B[j-1] <= A[i]` ဖြစ်ပါက ခွဲခြားမှု မှန်ကန်သွားပါပြီ။
        - စုစုပေါင်းအရေအတွက် မကိန်း ဖြစ်ပါက `max(A[i-1], B[j-1])` သည် Median ဖြစ်သည်။
        - စုံကိန်း ဖြစ်ပါက `(max(A[i-1], B[j-1]) + min(A[i], B[j])) / 2.0` သည် Median ဖြစ်သည်။
    - `A[i-1] > B[j]` ဖြစ်နေပါက - A ဘက်မှ partition က ကြီးလွန်းနေ၍ ဘယ်ဘက်သို့ ရွှေ့ရန် `right = i - 1` လုပ်သည်။
    - `B[j-1] > A[i]` ဖြစ်နေပါက - A ဘက်မှ partition က သေးလွန်းနေ၍ ညာဘက်သို့ ရွှေ့ရန် `left = i + 1` လုပ်သည်။

> **Time Complexity:** $O(\log(\min(m,n)))$ - ပိုမိုတိုတောင်းသော array ပေါ်တွင်သာ binary search ပြုလုပ်သောကြောင့် အလွန်လျင်မြန်သည်။
> **Space Complexity:** $O(1)$

Java Solution

```java
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        // nums1 က အမြဲတမ်း အရှယ်အစား ပိုငယ်စေရန် ပြောင်းလဲသည် (Binary Search ပိုမြန်စေရန်)
        if (nums1.length > nums2.length) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int m = nums1.length;
        int n = nums2.length;

        int left = 0;
        int right = m;
        int halfLen = (m + n + 1) / 2;

        while (left <= right) {
            int i = left + (right - left) / 2; // nums1 ၏ partition
            int j = halfLen - i;               // nums2 ၏ partition

            // နယ်နိမိတ်တန်ဖိုးများ သတ်မှတ်ခြင်း (Out of bounds ဖြစ်ပါက Infinity ယူသည်)
            int maxLeft1 = (i == 0) ? Integer.MIN_VALUE : nums1[i - 1];
            int minRight1 = (i == m) ? Integer.MAX_VALUE : nums1[i];

            int maxLeft2 = (j == 0) ? Integer.MIN_VALUE : nums2[j - 1];
            int minRight2 = (j == n) ? Integer.MAX_VALUE : nums2[j];

            // ခွဲခြားမှု မှန်ကန်မှု ရှိမရှိ စစ်ဆေးသည်
            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
                // စုစုပေါင်း အရေအတွက် မကိန်းဖြစ်လျှင်
```

```
                if ((m + n) % 2 != 0) {
                    return Math.max(maxLeft1, maxLeft2);
                }
                // စုစုပေါင်း အရေအတွက် စုံကိန်းဖြစ်လျှင်
                return ((long)Math.max(maxLeft1, maxLeft2) + Math.min(minRight1,
minRight2)) / 2.0;
            }
            // nums1 ၏ ဘယ်ဘက်ခြမ်းသည် ကြီးလွန်းနေပါက ၎င်းအား လျှော့ရန်
            else if (maxLeft1 > minRight2) {
                right = i - 1;
            }
            // nums1 ၏ ဘယ်ဘက်ခြမ်းသည် ငယ်လွန်းနေပါက တိုးရန်
            else {
                left = i + 1;
            }
        }

        return 0.0;
    }
}
```

အခန်း ၁၀ - Recursion

Recursion (ပြန်လည်ခေါ်ဆိုခြင်း) ဆိုတာ **Function တစ်ခုက သူ့ကိုယ်သူ ပြန်ခေါ်ပြီး**အလုပ်လုပ်တဲ့ နည်းလမ်းတစ်ခု ဖြစ်ပါတယ်။ ပြဿနာ (Problem) ကြီးတစ်ခုကို **တူညီတဲ့ပုံစံရှိတဲ့ ပိုသေးငယ်တဲ့ ပြဿနာ (Smaller Subproblem)** တွေအဖြစ် ခွဲခြမ်းပြီး ဖြေရှင်းသွားတဲ့ အတွေးအခေါ်ပါ။

Recursion ဟာ magic မဟုတ်ပါဘူး။ နောက်ကွယ်မှာ **Function Call Stack** ဆိုတဲ့ စနစ်နဲ့ အလုပ်လုပ်တာ ဖြစ်ပါတယ်။ ဒီအခန်းကို ကောင်းစွာ နားလည်ထားရင် နောက်ပိုင်း Trees, Graphs, Backtracking, Dynamic Programming အခန်းတွေအတွက် အခြေခံ ခိုင်မာသွားပါလိမ့်မယ်။

Recursion ၏ အဓိက အယူအဆ (Core Intuition)

Recursion ကို နားလည်ဖို့ အရိုးရှင်းဆုံး ဥပမာဖြစ်တဲ့ **Factorial** ($n!$) ကို ကြည့်ရအောင်။

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

ဒါကို ဂရုတစိုက် ကြည့်ရင် –

$5! = 5 \times 4!$

$4! = 4 \times 3!$

ဆိုတဲ့ ပုံစံကို တွေ့ရပါမယ်။ ဆိုလိုတာက `factorial(5)` ကို ဖြေဖို့ `factorial(4)` ကို သိရင်ရပြီ၊ `factorial(4)` ကို ဖြေဖို့ `factorial(3)` ကို သိရင်ရပြီ ဆိုတဲ့ သဘောပါ။ ဒါဟာ ပြဿနာကြီးတစ်ခုကို တူညီတဲ့ ပိုသေးတဲ့ ပြဿနာအဖြစ် ခွဲလိုက်တာ ဖြစ်ပါတယ်။

ဒါပေမယ့် ဒီလို ဆက်ခွဲနေရင် ဘယ်တော့ ရပ်မလဲ။ `factorial(1)` (သို့) `factorial(0)` ရောက်ရင် `1` ပြန်ပြီး ရပ်ရပါမယ် (`0! = 1` ဖြစ်သည်)။ ဒါကို **Base Case** လို့ ခေါ်ပါတယ်။

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))    // factorial(1) = 1 (Base Case)
= 5 * 4 * 3 * 2 * 1
= 120
```

ဒီနေရာမှာ Recursion ရဲ့ မရှိမဖြစ် အစိတ်အပိုင်း ၂ ခုကို တွေ့ရပါတယ်။

အစိတ်အပိုင်း	အဓိပ္ပါယ်	Factorial ဥပမာ
Base Case	ပြန်မခေါ်တော့ဘဲ ရပ်တန့်သွားမယ့် အခြေအနေ	`n == 0` (သို့) `n == 1` ဆိုလျှင် `1` ပြန်သည်

Recursive Case	သူ့ကိုယ်သူ ပိုသေးတဲ့ problem နဲ့ ပြန်ခေါ်တဲ့ အပိုင်း	`n * factorial(n - 1)`

Base Case မရှိရင် function က ဘယ်တော့မှ မရပ်ဘဲ အဆုံးမဲ့ ခေါ်နေပြီး Stack Overflow ဖြစ်သွားပါလိမ့်မယ်။

Function Call Stack

Recursion ကို တကယ် နားလည်ဖို့ **Call Stack** ကို မြင်တတ်ရပါမယ်။ Function တစ်ခုကို ခေါ်လိုက်တိုင်း computer က အဲ့ဒီ function ရဲ့ အချက်အလက် (parameter, local variable, ပြန်ဆက်လုပ်ရမယ့်နေရာ) ကို **Stack** ထဲ ထည့်ထားပါတယ်။ Function ပြီးသွားမှ Stack ထဲကနေ ပြန်ထုတ် (pop) ပါတယ်။

`factorial(3)` ကို ခေါ်လိုက်တဲ့အခါ Call Stack က ဒီလို အလုပ်လုပ်ပါတယ်။

အဆင့်	လုပ်ဆောင်ချက်	Call Stack (အပေါ်ဆုံး = နောက်ဆုံးခေါ်သည်)
၁	`factorial(3)` ခေါ်သည်	`factorial(3)`
၂	`factorial(2)` ထပ်ခေါ်သည်	`factorial(3)` → `factorial(2)`
၃	`factorial(1)` ထပ်ခေါ်သည်	`factorial(3)` → `factorial(2)` → `factorial(1)`
၄	`factorial(1)` က `1` ပြန်သည် (Base Case)	`factorial(3)` → `factorial(2)`
၅	`factorial(2)` က `2 * 1 = 2` ပြန်သည်	`factorial(3)`
၆	`factorial(3)` က `3 * 2 = 6` ပြန်သည်	(ဗလာ)

အောက်ဘက်ကို ဆင်းသွားတာ (function တွေ ထပ်ခေါ်တာ) ကို **Winding** လို့ ခေါ်ပြီး၊ Base Case ရောက်လို့ အပေါ်ပြန်တက်လာတာ (တန်ဖိုးတွေ ပြန်ပေးတာ) ကို**Unwinding** လို့ ခေါ်ပါတယ်။

Stack Overflow

Stack ရဲ့ နေရာက အကန့်အသတ်ရှိပါတယ်။ Recursion က အလွန်များတဲ့အကြိမ် (ဥပမာ - သိန်းနဲ့ချီ) ထပ်ခေါ်နေရင် Stack ပြည့်သွားပြီး **Stack Overflow** error ဖြစ်ပါတယ်။ ဒါကြောင့် —

- Base Case ကို မှန်ကန်စွာ သတ်မှတ်ထားရပါမယ်။
- Recursion depth အလွန်နက်နိုင်တဲ့ ပြဿနာတွေအတွက် iteration (loop) သို့မဟုတ် stack data structure ကို သုံးတာက ပိုသင့်တော်ပါတယ်။

Recursion vs Iteration

ပြဿနာတိုင်းကို Recursion နဲ့ရော Loop (Iteration) နဲ့ရော ဖြေလို့ ရပါတယ်။ ဘယ်ဟာ ရွေးမလဲ ဆိုတာ ပြဿနာရဲ့ သဘောပေါ်မူတည်ပါတယ်။

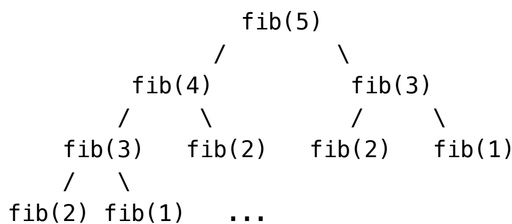
နှိုင်းယှဉ်ချက်	Recursion	Iteration (Loop)
ဖတ်ရလွယ်ကူမှု	Tree, Graph စတဲ့ ပြဿနာတွေအတွက် ပိုရှင်းလင်း	ရိုးရှင်းတဲ့ ထပ်ခါလုပ်ငန်းအတွက် ပိုကောင်း
Memory	Call Stack သုံးလို့ $O(depth)$ space ပို ယူသည်	Stack မယူလို့ $O(1)$ ဖြစ်နိုင်သည်
အန္တရာယ်	Stack Overflow ဖြစ်နိုင်သည်	Stack Overflow မဖြစ်
အသုံးဝင်မှု	ပြဿနာက သူ့အလိုလို subproblem အဖြစ် ကွဲတာမျိုး	တန်းတန်း ထပ်ခါလုပ်ရတာမျိုး

> **မှတ်ချက်:** Factorial လို ရိုးရှင်းတဲ့ ပြဿနာတွေကို loop နဲ့ ရေးတာ ပိုသင့်တော်ပေမယ့်၊ Tree traversal, folder ထဲက folder ထဲက file ရှာတာ စတဲ့ **ကိုယ်တိုင် ထပ်ဆင့်ခွဲထွက်တဲ့ (nested)** ပြဿနာတွေအတွက်တော့ Recursion က အများကြီး ပိုရှင်းပါတယ်။

Tree Recursion

တစ်ခါတစ်ရံ function တစ်ခုက သူ့ကိုယ်သူ **တစ်ကြိမ်ထက်ပိုပြီး** ပြန်ခေါ်တတ်ပါတယ်။ ဒါကို **Tree Recursion** လို့ ခေါ်ပါတယ်။ အကောင်းဆုံး ဥပမာက Fibonacci ဖြစ်ပါတယ်။

$$fib(n) = fib(n-1) + fib(n-2)$$



```
                 fib(5)
               /        \
          fib(4)          fib(3)
         /     \          /     \
    fib(3)    fib(2)   fib(2)    fib(1)
     /   \
fib(2) fib(1)    ...
```

ဒီပုံမှာ `fib(2)` , `fib(3)` တွေကို **အကြိမ်ကြိမ် ထပ်ခါတွက်နေတာ** ကို သတိထားမိပါလိမ့်မယ်။ ဒီလို ထပ်နေတဲ့ အလုပ်တွေ (Overlapping Subproblems) ကို သိမ်းထားပြီး ပြန်သုံးတဲ့ နည်းလမ်းကို **Dynamic Programming** လို့ ခေါ်ပြီး အခန်း ၂၂ မှာ ဆက်လေ့လာရပါမယ်။

Real-world Examples

Recursion ဟာ ပညာရပ်သက်သက်ဆန်တဲ့ concept မဟုတ်ဘဲ၊ programming မှာ **Nested ဖွဲ့စည်းထားတဲ့ data** တွေကို ကိုင်တွယ်ရာမှာ အမြဲ ကြုံရပါတယ်။

- **Folder Traversal** — Folder တစ်ခုထဲမှာ folder တွေ ထပ်ရှိ၊ အဲ့ folder တွေထဲမှာ folder တွေ ထပ်ရှိ။ File တွေ အကုန် ရှာဖို့ recursion သုံးရသည်။

- **Comment Replies** — Comment တစ်ခုအောက်မှာ reply, reply အောက်မှာ reply ထပ်ရှိနိုင်သည် (nested thread)။
- **Category Tree / Menu Tree** — Category ကြီးအောက်မှာ subcategory, အဲ့အောက်မှာ ထပ်မံ subcategory။
- **JSON / DOM Tree** — Web page ရဲ့ HTML element တွေဟာ element ထဲ element ထပ်ရှိတဲ့ tree structure ဖြစ်သည်။

ဒီ data တွေအားလုံးရဲ့ တူညီချက်က **ကိုယ်တိုင်ကိုယ်ကျ ထပ်ဆင့်ဖွဲ့စည်းထား (self-similar / nested)** ခြင်း ဖြစ်ပြီး၊ ဒါက Recursion နဲ့ အကိုက်ညီဆုံး ဖြစ်ပါတယ်။

Questions

Recursion ကို ကောင်းစွာ နားလည်စေဖို့ အခြေခံကျတဲ့ မေးခွန်း ၆ မျိုးကို တစ်ဆင့်ချင်း လေ့လာကြည့်ရအောင်။

၁။ Factorial

အပြုသဘော ကိန်းပြည့် `n` တစ်ခုအတွက် `n!` (factorial) ကို recursion သုံးပြီး တွက်ပါ။

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$ ဖြစ်ပြီး `0! = 1` ဖြစ်သည်။

Example 1:

```
Input: n = 5
Output: 120
Explanation: 5 * 4 * 3 * 2 * 1 = 120
```

ရှင်းလင်းချက်

ဒါက Recursion ရဲ့ အခြေခံအကျဆုံး ပုစ္ဆာ ဖြစ်ပါတယ်။

- **Base Case:** `n` က `0` သို့မဟုတ် `1` ဖြစ်ရင် `1` ကို တန်းပြန်သည်။
- **Recursive Case:** `n * factorial(n - 1)` ကို ပြန်သည်။

> **Time Complexity:** $O(n)$ - function ကို `n` ကြိမ် ခေါ်သောကြောင့်။
> **Space Complexity:** $O(n)$ - Call Stack တွင် တစ်ပြိုင်နက် `n` ကြိမ်စာ နေရာ ယူသောကြောင့်။

Java Solution

```java
class Solution {
    public long factorial(int n) {
        // Base Case: n က 0 သို့ 1 ဆိုလျှင် ရပ်သည်
        if (n <= 1) {
            return 1;
```

```
        }
        // Recursive Case: သူ့ကိုယ်သူ ပိုသေးသော problem ဖြင့် ပြန်ခေါ်သည်
        return n * factorial(n - 1);
    }
}
```

၂။ Fibonacci Number

Fibonacci အစဉ်ဟာ `0` , `1` နဲ့စပြီး၊ နောက်ဂဏန်းတိုင်းသည် ရှေ့ဂဏန်း ၂ ခု ပေါင်းခြင်း ဖြစ်သည်။ `n` ပေးထားသည့်အခါ n^{th} Fibonacci number ကို ပြန်ပါ။

$$fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$$

Example 1:

```
Input: n = 6
Output: 8
Explanation: 0, 1, 1, 2, 3, 5, 8 - index 6 သည် 8 ဖြစ်သည်။
```

ရှင်းလင်းချက်

ဒါက **Tree Recursion** ရဲ့ ဥပမာ ဖြစ်ပါတယ်။ function တစ်ခုက သူ့ကိုယ်သူ **၂ ကြိမ်** ပြန်ခေါ်ပါတယ်။

- **Base Case:** `n < 2` ဆိုလျှင် `n` ကို တန်းပြန်သည် (`fib(0)=0` , `fib(1)=1`)။
- **Recursive Case:** `fib(n-1) + fib(n-2)` ကို ပြန်သည်။

> **သတိ:** ဒီ pure recursion နည်းက subproblem တွေကို အကြိမ်ကြိမ် ထပ်တွက်နေလို့ $O(2^n)$ ကြာပါတယ်။ ဒါကို Memoization နဲ့ $O(n)$ အထိ လျှော့ချနိုင်ပုံကို အခန်း ၂၂ (Dynamic Programming) မှာ ဆက်လေ့လာပါမယ်။

> **Time Complexity:** $O(2^n)$ - recursion tree က အပိုင်းနှစ်ခုစီ ထပ်ဖွံ့သွားသောကြောင့်။
> **Space Complexity:** $O(n)$ - Call Stack ရဲ့ အနက်ဆုံး အလွှာသည် `n` ဖြစ်သောကြောင့်။

Java Solution

```java
class Solution {
    public int fib(int n) {
        // Base Case
        if (n < 2) {
            return n;
        }
        // Recursive Case: သူ့ကိုယ်သူ ၂ ကြိမ် ပြန်ခေါ်သည် (Tree Recursion)
        return fib(n - 1) + fib(n - 2);
    }
}
```

၃။ Reverse a String

Character array `s` တစ်ခုကို recursion သုံးပြီး နေရာချင်း ပြောင်းပြန် (reverse) လုပ်ပါ။ နေရာအပို မယူဘဲ (in-place) ပြုလုပ်ပါ။

Example 1:

```
Input: s = ['h','e','l','l','o']
Output: ['o','l','l','e','h']
```

ရှင်းလင်းချက်

Two Pointers သဘောကို recursion နဲ့ ပေါင်းစပ်ထားတာ ဖြစ်ပါတယ်။ `left` နဲ့ `right` အစွန်းနှစ်ဖက်က character တွေကို လဲ၊ ပြီးရင် အတွင်းဘက်ကို ဆက်ဝင်သည်။

- **Base Case:** `left >= right` ဆိုလျှင် အလယ်ရောက်ပြီ ဖြစ်လို့ ရပ်သည်။
- **Recursive Case:** `s[left]` နဲ့ `s[right]` ကို လဲပြီး `reverse(s, left + 1, right - 1)` ကို ဆက်ခေါ်သည်။

> **Time Complexity:** $O(n)$ - character `n` ခုကို တစ်ကြိမ်စီ ထိသောကြောင့်။
> **Space Complexity:** $O(n)$ - Call Stack အတွက် ဖြစ်သည်။

Java Solution

```java
class Solution {
    public void reverseString(char[] s) {
        reverse(s, 0, s.length - 1);
    }

    private void reverse(char[] s, int left, int right) {
        // Base Case: pointer နှစ်ခု ဆုံသွားလျှင် ရပ်သည်
        if (left >= right) {
            return;
        }
        // အစွန်းနှစ်ဖက် character ကို လဲသည်
        char temp = s[left];
        s[left] = s[right];
        s[right] = temp;
        // အတွင်းဘက်သို့ ဆက်ဝင်သည်
        reverse(s, left + 1, right - 1);
    }
}
```

၄။ Sum of Nested List

ကိန်းဂဏန်းတွေ ရော list တွေ ရော ရောထားတဲ့ nested list တစ်ခု ပေးထားသည်။ ထဲက ကိန်းဂဏန်း အားလုံးရဲ့ စုစုပေါင်းကို ရှာပါ။ list ထဲမှာ list ထပ်ပါနိုင်သည် (အလွှာ မည်မျှ နက်နက်)။

Example 1:

```
Input: [1, [2, 3], [4, [5, 6]]]
Output: 21
Explanation: 1 + 2 + 3 + 4 + 5 + 6 = 21
```

ရှင်းလင်းချက်

ဒါက Recursion ရဲ့ အစစ်အမှန် အသုံးဝင်ပုံ ဖြစ်ပါတယ်။ Nested structure ကို ကိုင်တွယ်ဖို့ recursion က အကိုက်ညီဆုံးပါ။ Element တစ်ခုစီကို ကြည့်ပြီး —

- **Base Case:** element က ကိန်းဂဏန်း ဖြစ်ရင် အဲ့တန်ဖိုးကို ပေါင်းသည်။
- **Recursive Case:** element က list ဖြစ်ရင် အဲ့ list ထဲကို recursion နဲ့ ပြန်ဝင်ပြီး စုစုပေါင်းကို ယူသည်။

> **Time Complexity:** $O(n)$ - nested element အားလုံး (စုစုပေါင်း `n` ခု) ကို တစ်ကြိမ်စီ ကြည့်သောကြောင့်။
> **Space Complexity:** $O(d)$ - `d` သည် list ၏ အနက်ဆုံး အလွှာ (depth) ဖြစ်သည်။

Java Solution

```java
import java.util.List;

class Solution {
    // Object ဖြစ်နိုင်သည် — Integer သို့မဟုတ် List
    public int sumNested(List<Object> list) {
        int total = 0;
        for (Object item : list) {
            if (item instanceof Integer) {
                // Base Case: ကိန်းဂဏန်းဖြစ်လျှင် တန်းပေါင်းသည်
                total += (Integer) item;
            } else {
                // Recursive Case: list ဖြစ်လျှင် ထဲကို ပြန်ဝင်သည်
                @SuppressWarnings("unchecked")
                List<Object> subList = (List<Object>) item;
                total += sumNested(subList);
            }
        }
        return total;
    }
}
```

၅။ Binary Tree Preorder Traversal

Binary tree တစ်ခု၏ root ကို ပေးထားသည်။ Node တန်ဖိုးများကို **Preorder** (root → left → right) အစီအစဉ်ဖြင့် ပြန်ပါ။

Example 1:

```
Input:
      1
       \
```

```
        2
       /
      3
  Output: [1, 2, 3]
```

ရှင်းလင်းချက်

Tree ဆိုတာ သူ့အလိုလို **recursive structure** ဖြစ်ပါတယ် — node တိုင်းသည် သူ့အောက်မှာ subtree (ပိုသေးတဲ့ tree) တွေ ဆွဲထားပါတယ်။ ဒါကြောင့် tree traversal အတွက် recursion က သဘာဝအကျဆုံး နည်းလမ်း ဖြစ်ပါတယ်။

- **Base Case:** node က `null` ဖြစ်လျှင် ဘာမှ မလုပ်ဘဲ ပြန်သည်။
- **Recursive Case:** node ၏ တန်ဖိုးကို မှတ်၊ ပြီးရင် left subtree ကို recursion ဆက်ဝင်၊ ပြီးရင် right subtree ကို ဆက်ဝင်သည်။

> Visit လုပ်တဲ့ အစီအစဉ်ကို ပြောင်းရုံနဲ့ **Inorder** (left → root → right) နဲ့ **Postorder** (left → right → root) ကိုလည်း တွက်နိုင်ပါတယ်။ Tree အကြောင်းကို အခန်း ၁၃ မှာ အသေးစိတ် ဆက်လေ့လာပါမယ်။

> **Time Complexity:** $O(n)$ - node `n` ခုစလုံးကို တစ်ကြိမ်စီ ကြည့်သောကြောင့်။
> **Space Complexity:** $O(h)$ - `h` သည် tree ၏ အမြင့် (Call Stack depth) ဖြစ်သည်။

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) { this.val = val; }
}

class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        traverse(root, result);
        return result;
    }

    private void traverse(TreeNode node, List<Integer> result) {
        // Base Case: node မရှိလျှင် ရပ်သည်
        if (node == null) {
            return;
        }
        result.add(node.val);         // root
        traverse(node.left, result);  // left subtree
        traverse(node.right, result); // right subtree
    }
}
```

၆။ Generate All Subsets (Combinations Introduction)

Distinct ကိန်းဂဏန်းတွေပါတဲ့ array `nums` တစ်ခု ပေးထားသည်။ ဖြစ်နိုင်တဲ့ subset (power set) အားလုံးကို ထုတ်ပေးပါ။

Example 1:

```
Input: nums = [1, 2, 3]
Output: [[], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3]]
Note: subset များ၏ အစီအစဉ်ကို မည်သို့ဖြစ်စေ (any order) ပြန်နိုင်သည်။
```

ရှင်းလင်းချက်

ဒါက Recursion ကနေ **Backtracking** (အခန်း ၁၆) ဆီ ကူးပြောင်းပေးတဲ့ မေးခွန်း ဖြစ်ပါတယ်။ Element တစ်ခုစီအတွက် **"ထည့်မလား / မထည့်ဘူးလား"** ဆိုတဲ့ ဆုံးဖြတ်ချက် ၂ ခု ရှိပါတယ်။ ဒီ ဆုံးဖြတ်ချက်တွေကို recursion tree အဖြစ် ဖွဲ့ပြီး အကုန်လိုက်စမ်းသွားရပါမယ်။

- **Choose:** လက်ရှိ element ကို subset ထဲ ထည့်ပြီး ရှေ့ဆက်သည်။
- **Explore:** နောက် element ဆီ recursion နဲ့ ဆက်ဝင်သည်။
- **Undo (Backtrack):** ထည့်ထားတဲ့ element ကို ပြန်ထုတ်ပြီး "မထည့်တဲ့" လမ်းကြောင်းကို စမ်းသည်။

> **Time Complexity:** $O(n \times 2^n)$ - subset 2^n ခု ရှိပြီး တစ်ခုစီ ကူးယူရန် $O(n)$ ကြာသောကြောင့်။
> **Space Complexity:** $O(n)$ - recursion depth နှင့် လက်ရှိ subset အတွက် (output array မပါ)။ Output အားလုံးကို တွက်ရင်တော့ subset 2^n ခု၊ တစ်ခုစီ အရှည် $O(n)$ ဖြစ်လို့ $O(n \cdot 2^n)$ ဖြစ်သည်။

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] nums, int start,
                           List<Integer> current, List<List<Integer>> result) {
        // လက်ရှိ subset ကို အဖြေထဲ ထည့်သည် (လမ်းကြောင်းတိုင်းသည် subset တစ်ခု)
        result.add(new ArrayList<>(current));

        for (int i = start; i < nums.length; i++) {
            // Choose: element ကို ထည့်သည်
            current.add(nums[i]);
```

```
            // Explore: နောက် element ဆီ ဆက်ဝင်သည်
            backtrack(nums, i + 1, current, result);
            // Undo: ပြန်ထုတ်ပြီး အခြားလမ်းကြောင်း စမ်းသည်
            current.remove(current.size() - 1);
        }
    }
}
```

အခန်း ၁၁ - Sorting

Sorting (စီခြင်း) ဆိုတာ data တွေကို **သတ်မှတ်ထားတဲ့ အစီအစဉ်တစ်ခု** (ဥပမာ - အငယ်ဆုံးကနေ အကြီးဆုံး၊ A ကနေ Z) အတိုင်း ပြန်စီပေးတဲ့ လုပ်ငန်းစဉ် ဖြစ်ပါတယ်။ Developer တစ်ယောက်အနေနဲ့ data ကို စီရတာ နေ့စဉ်လိုလို ကြုံရပါတယ် — user တွေကို registration date အလိုက်၊ product တွေကို ဈေးနှုန်းအလိုက်၊ leaderboard ကို score အလိုက် စသဖြင့်ပါ။

ဒါပေမယ့် Sorting ဟာ "data စီတယ်" ဆိုတဲ့ အလုပ်တစ်ခုတည်းအတွက်ပဲ အရေးကြီးတာ မဟုတ်ပါဘူး။ **စီပြီးသား (sorted) data ဟာ အလုပ်အများကြီးကို ပိုလွယ်စေပါတယ်။** ဥပမာ —

- Binary Search (အခန်း ၉) လုပ်ဖို့ data က စီပြီးသား ဖြစ်ဖို့ လိုသည်။
- ထပ်နေတဲ့ (duplicate) element တွေ ရှာဖို့ စီလိုက်ရင် ဘေးချင်းကပ် ရှာခြင်း။
- အကြီးဆုံး/အငယ်ဆုံး `k` ခု ရှာဖို့ စီပြီး ထိပ်ကနေ ယူနိုင်သည်။
- Interval, schedule ပြဿနာတွေဟာ စီပြီးမှ ဖြေရှင်းလို့ ရသည်။

ဒါကြောင့် Sorting ကို algorithm အများကြီးရဲ့ **အခြေခံ စတင်အဆင့် (preprocessing step)** အဖြစ် မြင်တတ်ဖို့ အရေးကြီးပါတယ်။

Sorting method တွေ အများကြီးရှိပါတယ်။ ဒါပေမယ့် အကုန်လုံးကို **တန်းတူ အရေးထား** လေ့လာစရာ မလိုပါဘူး။ ဒီအခန်းမှာ —

- **Basic Sorts** (Bubble, Selection, Insertion) — concept နားလည်ဖို့အတွက်သာ။
- **Efficient Sorts** (Merge, Quick, Heap) — real-world မှာ အသုံးတည့်တဲ့ $O(n \log n)$ sorts။
- **Non-comparison Sorts** (Counting, Radix, Bucket) — special case တွေအတွက်။

ဆိုပြီး အပိုင်း ၃ ပိုင်း ခွဲ လေ့လာသွားပါမယ်။ လက်တွေ့မှာ programming language အများစုမှာ built-in sort function (`Arrays.sort()` , `.sorted()`) တွေ ပါပြီးသား ဖြစ်ပေမယ့်၊ နောက်ကွယ်က အလုပ်လုပ်ပုံ နားလည်ထားမှ **ဘယ်အချိန် ဘာရွေးသုံးရမလဲ** ဆုံးဖြတ်တတ်ပါမယ်။

Sorting ၏ အခြေခံ Concepts နှစ်ခု

Algorithm တွေ မလေ့လာခင်၊ sorting algorithm တိုင်းကို နှိုင်းယှဉ်တဲ့အခါ မေးရမယ့် မေးခွန်း ၂ ခု ရှိပါတယ် —

- **"Stable ဖြစ်လား?"**
- **"In-place ဖြစ်လား?"**

ဒီ concept ၂ ခုက algorithm ရွေးချယ်မှုကို တိုက်ရိုက် ဆုံးဖြတ်ပါတယ်။

Stable vs Unstable

Stable sort ဆိုတာ တန်ဖိုးတူတဲ့ element တွေရဲ့ မူလ အစီအစဉ် (relative order) ကို မပျက်စေဘဲ ထိန်းထားပေးတာ ဖြစ်ပါတယ်။

ဘာကြောင့် အရေးကြီးလဲ ဆိုတာ ဥပမာနဲ့ ကြည့်ရအောင်။ User list ကို အရင်က **name** အလိုက် စီထားပြီးသား ဖြစ်တယ် ဆိုပါစို့။ အခု **age** အလိုက် ထပ်စီချင်တယ်။

```
မူလ (name အလိုက် စီထားပြီး):
  Alice (25),  Bob (30),  Carol (25),  David (30)

age အလိုက် Stable sort လုပ်ရင်:
  Alice (25),  Carol (25),  Bob (30),  David (30)
  → age တူတဲ့ Alice, Carol တို့ဟာ မူလ name အစီအစဉ်အတိုင်း ကျန်သည်။

age အလိုက် Unstable sort လုပ်ရင်:
  Carol (25),  Alice (25),  David (30),  Bob (30)
  → age တူတဲ့ element တွေရဲ့ အစီအစဉ် ပျက်သွားနိုင်သည်။
```

Stable sort ဖြစ်မှ **"age အလိုက် စီ၊ age တူရင် name အလိုက် စီ"** ဆိုတဲ့ **multi-field sorting** ကို အဆင့်လိုက် (field တစ်ခုပြီးတစ်ခု) လုပ်လို့ ရပါတယ်။ ဒါကြောင့် real-world object sorting မှာ stability က အလွန် အရေးကြီးပါတယ်။

In-place vs Extra Memory

In-place sort ဆိုတာ **memory အပို မယူဘဲ** ($O(1)$ extra space) မူလ array ထဲမှာတင် နေရာချင်းလဲပြီး စီတာ ဖြစ်ပါတယ်။ **Extra memory** လိုတဲ့ sort (ဥပမာ - Merge Sort) ကတော့ array အသစ်တွေ ဆောက်ပြီး စီတာ ဖြစ်လို့ data အရွယ်နဲ့ အချိုးကျ memory ပိုသုံးပါတယ်။

Data က သေးရင် ဒီ memory က ပြဿနာ မဟုတ်ပါ။ ဒါပေမယ့် memory အကန့်အသတ်ရှိတဲ့ စက် (embedded device) တွေ၊ ဒါမှမဟုတ် data အလွန်ကြီးတဲ့အခါ in-place ဖြစ်တာ အရေးကြီးလာပါတယ်။

အောက်က ဇယားက sorting algorithm အားလုံးရဲ့ ခြုံငုံ နှိုင်းယှဉ်ချက် ဖြစ်ပါတယ်။

Algorithm	Time (Average)	Time (Worst)	Space	Stable	အသုံးပြုရန်
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$	✅	လေ့လာရန် အတွက်သာ
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	❌	swap နည်းချင်ရင်
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$	✅	data သေး/စီပြီးသား
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	✅	stable လိုရင်၊ linked list
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$ avg / $O(n)$ worst	❌	general-purpose (မြန်)

Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	❌	memory အကန့်အသတ်ရှိရင်
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	✅*	integer, range သေး
Radix Sort	$O(d(n+k))$	$O(d(n+k))$	$O(n+k)$	✅	ကိန်းကြီး/string

> Counting Sort က **prefix sum နဲ့ output array သုံး ရေးမှသာ** stable ဖြစ်ပါတယ်။
> **မှတ်ချက်:** ဇယားထဲက "Stable" / "Space" တန်ဖိုးတွေဟာ **ပုံမှန် (typical) implementation** ကို ရည်ညွှန်းတာ ဖြစ်ပါတယ်။ ဥပမာ - Quick Sort ရဲ့ space က average $O(\log n)$ ဖြစ်ပေမယ့် pivot မကောင်းရင် worst case $O(n)$ ဖြစ်နိုင်ပါတယ်။

အပိုင်း ၁ — Basic Sorting Methods

ဒီ sort ၃ မျိုးက အကုန်လုံး $O(n^2)$ ဖြစ်ပြီး data ကြီးလာရင် နှေးပါတယ်။ ဥပမာ - element ၁ သန်း ($n = 10^6$) ဆိုရင် $n^2 = 10^{12}$ ကြိမ် တွက်ရမှာ ဖြစ်လို့ နာရီနဲ့ချီ ကြာနိုင်ပါတယ်။ ဒါကြောင့် real-world မှာ အဓိက မသုံးပါဘူး။ ဒါပေမယ့် sorting ရဲ့ concept ("နှိုင်းယှဉ်ပြီး လဲ" ဆိုတဲ့ အခြေခံ) ကို နားလည်ဖို့ အကောင်းဆုံး စမှတ် ဖြစ်ပါတယ်။

Bubble Sort

Bubble Sort ဟာ ဘေးချင်းကပ် element ၂ ခုကို နှိုင်းယှဉ်ပြီး အစီအစဉ် မှားနေရင် **လဲ (swap)** လုပ်တဲ့ နည်းလမ်း ဖြစ်ပါတယ်။ Loop တစ်ပတ်ပြီးတိုင်း အကြီးဆုံး element က ရေပေါ်က **ပူဖောင်း (bubble)** လို နောက်ဆုံးနေရာကို တက်သွားတဲ့အတွက် ဒီနာမည် ရတာ ဖြစ်ပါတယ်။

`[5, 1, 4, 2]` ကို စီတဲ့ အဆင့်ဆင့်ကို အသေးစိတ် ကြည့်ရအောင်။ **Pass ၁** (ဘေးချင်းကပ် နှိုင်းယှဉ်ခြင်း) —

နှိုင်းယှဉ်	array အခြေအနေ	လုပ်ဆောင်ချက်
`5` vs `1`	`[5, 1, 4, 2]`	`5 > 1` $\rightarrow$ လဲ $\rightarrow$ `[1, 5, 4, 2]`
`5` vs `4`	`[1, 5, 4, 2]`	`5 > 4` $\rightarrow$ လဲ $\rightarrow$ `[1, 4, 5, 2]`
`5` vs `2`	`[1, 4, 5, 2]`	`5 > 2` $\rightarrow$ လဲ $\rightarrow$ `[1, 4, 2, 5]`

Pass ၁ ပြီးတော့ အကြီးဆုံး `5` က နောက်ဆုံးနေရာ ရောက်သွားပါတယ်။ Pass ၂ မှာ `[1, 4, 2]` ကို ဆက်လုပ်ပြီး `4` က နေရာရ၊ Pass ၃ မှာ `[1, 2]` ကို လုပ်ပြီး စီပြီးသား ဖြစ်သွားပါတယ်။

```
void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        boolean swapped = false;
```

```
            // တစ်ပတ်တိုင်း အကြီးဆုံးက အနောက်ဆုံးရောက်လို့ (n-1-i) အထိသာ ကြည့်ရန် လိုလောက်
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // တစ်ပတ်လုံး swap မဖြစ်ရင် စီပြီးသား ဖြစ်လို့ ရပ်သည် (optimization)
            if (!swapped) break;
        }
    }
```

`swapped` flag က optimization တစ်ခု ဖြစ်ပါတယ်။ Data က စီပြီးသား ဖြစ်နေရင် ပထမ pass မှာ ဘယ်တုန်းမှ swap မဖြစ်လို့ ချက်ချင်း ရပ်ပြီး $O(n)$ ဖြစ်သွားပါတယ်။

> **မှတ်ချက်:** Bubble Sort ကို **သင်ယူဖို့သက်သက်** အတွက်သာ ဖြစ်ပြီး production code မှာ **မသုံးသင့်ပါ**။ ရိုးရှင်းပြီး နားလည်လွယ်ပေမယ့်၊ swap အကြိမ်ရေ အလွန်များတဲ့အတွက် basic sort ၃ ခုထဲမှာတောင် အနှေးဆုံး ဖြစ်ပါတယ်။ ကျောင်းသုံးစာအုပ်တွေမှာ နာမည်ကြီးပေမယ့်၊ လက်တွေ့မှာ Bubble Sort ကို interview နဲ့ classroom အပြင် ရှားရှားပါးပါး တွေ့ရတာက — တူညီအောင် ရိုးရှင်းတဲ့ Insertion Sort က ပိုကောင်းတဲ့ performance ပေးနိုင်လို့ ဖြစ်ပါတယ်။

Selection Sort

Selection Sort ဟာ စီမထားသေးတဲ့ အပိုင်းထဲက **အငယ်ဆုံး element ကို ရွေး (select)** ပြီး၊ စီပြီးသား အပိုင်းရဲ့ နောက်ဆုံးနေရာနဲ့ လဲတဲ့ နည်းလမ်း ဖြစ်ပါတယ်။ "အငယ်ဆုံးကို ရွေး၊ ရွှေ့ထား" ဆိုတဲ့ အလုပ်ကို ထပ်ခါ လုပ်သွားတာ ဖြစ်ပါတယ်။

`[5, 2, 4, 1]` ကို စီကြည့်ရအောင် —

အဆင့်	စီမထားသေးသော အပိုင်း	အငယ်ဆုံး	လဲပြီးနောက်
၁	`[5, 2, 4, 1]`	`1`	`[1, 2, 4, 5]` (`5` နဲ့ `1` လဲ)
၂	`[2, 4, 5]`	`2`	`[1, 2, 4, 5]` (နေရာမှန်ပြီး)
၃	`[4, 5]`	`4`	`[1, 2, 4, 5]` (နေရာမှန်ပြီး)

```
void selectionSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        // i နောက်ပိုင်းမှ အငယ်ဆုံးကို ရှာသည်
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // အငယ်ဆုံးကို လက်ရှိနေရာ (i) နဲ့ လဲသည်
```

```
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
```

> Selection Sort ရဲ့ ထူးခြားချက်က **swap (write) အကြိမ်ရေ နည်းတာ** (အများဆုံး n ကြိမ်) ဖြစ်ပါတယ်။ memory ထဲ ရေး (write) တာ ဈေးကြီးတဲ့ environment တွေ (ဥပမာ - flash memory၊ write count ကန့်သတ်ထားတဲ့ storage) မှာ Bubble Sort ထက် swap နည်းတဲ့အတွက် အသုံးဝင်နိုင်ပါတယ်။ (Java လို language မှာ object ကြီးတွေ swap လုပ်တာဟာ object တစ်ခုလုံး မဟုတ်ဘဲ **reference** ကိုသာ လဲတာ ဖြစ်လို့ "object ကြီးလို့ swap ဈေးကြီး" ဆိုတာ မမှန်ပါ။) ဒါပေမယ့် comparison ကတော့ data စီပြီးသား ဖြစ်နေရင်တောင် **အမြဲ** $O(n^2)$ ဖြစ်နေပါတယ်။ ဒါ့အပြင် Selection Sort က **unstable** ဖြစ်ပါတယ် — ဝေးတဲ့ element တွေကို လဲတဲ့အတွက် တန်ဖိုးတူ element တွေရဲ့ အစီအစဉ် ပျက်နိုင်ပါတယ်။

Insertion Sort

Insertion Sort ဟာ **ဖဲချပ်တွေကို လက်ထဲမှာ စီတဲ့ ပုံစံ** ဖြစ်ပါတယ်။ ဖဲတစ်ချပ် ထပ်ယူတိုင်း၊ လက်ထဲက စီပြီးသား ဖဲတွေကြားမှာ မှန်ကန်တဲ့နေရာကို ထိုးထည့်တာ မျိုးပါ။ Element တစ်ခုစီကို ယူပြီး၊ သူ့ရှေ့က **စီပြီးသား အပိုင်းထဲမှာ မှန်ကန်တဲ့နေရာကို ထိုးထည့် (insert)** ပါတယ်။

`[5, 2, 4, 1]` ကို စီကြည့်ရအောင်။ `|` က "ဒီ ဘယ်ဘက်ပိုင်းသည် စီပြီး" ဆိုတာ ပြပါတယ် —

ယူသော element	ဖြစ်စဉ်	array အခြေအနေ
(စတင်)	ပထမ element ကို စီပြီးသား ယူဆ	`[5 | 2, 4, 1]`
`2`	`2 < 5` → `5` ကို ညာတွန်း၊ `2` ကို ရှေ့ထား	`[2, 5 | 4, 1]`
`4`	`4 < 5` → `5` တွန်း၊ `4 > 2` → `4` ကို `2`, `5` ကြား	`[2, 4, 5 | 1]`
`1`	`1` က အကုန်ထက်ငယ် → အရှေ့ဆုံး ထိုးထည့်	`[1, 2, 4, 5 |]`

```
void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // ထိုးထည့်မယ့် element
        int j = i - 1;
        // key ထက် ကြီးတဲ့ element တွေကို တစ်နေရာစီ ညာဘက်တွန်းသည်
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // ဖြစ်ပေါ်လာတဲ့ နေရာလွတ်မှာ key ထည့်သည်
    }
}
```

> **အသုံးဝင်မှု:** Insertion Sort ဟာ basic sort ၃ ခုထဲမှာ **လက်တွေ့အသုံးဝင်ဆုံး** ဖြစ်ပါတယ်။
>
> - **Small Data** ဖြစ်ရင် $O(n^2)$ ဖြစ်ပေမယ့် overhead နည်းလို့ Quick/Merge Sort ထက်တောင် ပိုမြန်နိုင်သည်။
> - **Nearly sorted** ဖြစ်ရင် inner `while` loop က ဘာမှ မလုပ်ရဘဲ ကျော်သွားလို့ $O(n)$ နီးပါး ဖြစ်သည်။
>
> ဒါကြောင့် Java, Python စတဲ့ language အများစုရဲ့ built-in sort ဟာ data အပိုင်းသေးတွေ ရောက်တဲ့အခါ Insertion Sort ကို အလိုအလျောက် တွဲသုံးကြပါတယ်။

အပိုင်း ၂ — Efficient Comparison Sorting

Basic sort တွေက ဘာကြောင့် နှေးသလဲ ဆိုရင်၊ element တစ်ခုကို သူနဲ့ ဝေးတဲ့ နေရာ ရောက်ဖို့ တစ်ဆင့်ချင်းသာ ရွှေ့လို့ ဖြစ်ပါတယ်။ Efficient sort တွေက **Divide and Conquer** (ခွဲပြီး အနိုင်ယူ) သဘောတရားနဲ့ ပြဿနာကို တစ်ဝက်စီ ခွဲဖြေတဲ့အတွက် $O(n \log n)$ ရအောင် လုပ်နိုင်ပါတယ်။

> $O(n \log n)$ **ဆိုတာ ဘာကြောင့် ကောင်းသလဲ?** Element ၁ သန်းအတွက် $n^2 = 10^{12}$ ဖြစ်ပေမယ့် $n \log n \approx 2 \times 10^7$ သာ ဖြစ်ပါတယ်။ ၅ သိန်းဆ ပိုမြန်တာ ဖြစ်ပါတယ်။

Divide and Conquer ဆိုတာ ဘာလဲ

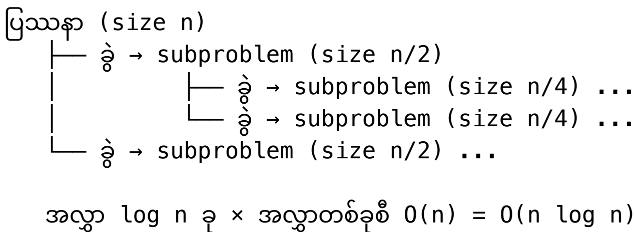
Merge Sort နဲ့ Quick Sort မလေ့လာခင်၊ သူတို့ ၂ ခုလုံးရဲ့ နောက်ကွယ်က **Divide and Conquer** (ခွဲပြီး အနိုင်ယူ) သဘောတရားကို အရင် နားလည်ထားရင် ပိုရှင်းပါတယ်။ ဒါက sorting တင် မဟုတ်ဘဲ algorithm design ရဲ့ အရေးကြီးဆုံး pattern တစ်ခု ဖြစ်ပါတယ်။

အယူအဆက ရိုးရှင်းပါတယ် — **ပြဿနာကြီးတစ်ခုကို တိုက်ရိုက် မဖြေဘဲ၊ ပိုသေးတဲ့ ပြဿနာ ၂ ခု (သို့) အများ အဖြစ် ခွဲ၊ တစ်ခုစီ သီးသန့်ဖြေ၊ ပြီးမှ ပြန်ပေါင်း** လုပ်တာ ဖြစ်ပါတယ်။ အဆင့် ၃ ဆင့် ရှိပါတယ် —

အဆင့်	အလုပ်	Merge Sort ဥပမာ
၁. Divide	ပြဿနာကို ပိုသေးတဲ့ subproblem အဖြစ် ခွဲသည်	array ကို တစ်ဝက်စီ ခွဲသည်
၂. Conquer	subproblem တစ်ခုစီကို (များသောအားဖြင့် recursion ဖြင့်) ဖြေသည်	တစ်ခုစီကို သီးသန့် စီသည်
၃. Combine	subproblem ၏ အဖြေများကို ပြန်ပေါင်းပြီး အဖြေအပြည့် ရယူသည်	စီပြီးသား ၂ ခြမ်းကို merge လုပ်သည်

ဒီ pattern ဟာ **Recursion** (အခန်း ၁၀) နဲ့ တိုက်ရိုက် ဆက်စပ်ပါတယ် — "ပြဿနာကြီးကို တူညီတဲ့ ပို သေးတဲ့ ပြဿနာ အဖြစ် ခွဲ" ဆိုတဲ့ idea အတိုင်းပါပဲ။ ကွာတာက — recursion သက်သက်မှာ subproblem **တစ်ခု** ဆီ ဆင်းတတ်ပြီး၊ divide and conquer မှာ subproblem **အများ** (ပုံမှန် ၂ ခု) ဆီ ခွဲ ဆင်းပြီး အဖြေတွေ ပြန်ပေါင်းတာ ဖြစ်ပါတယ်။

ဘာကြောင့် မြန်သလဲ? ပြဿနာကို တစ်ဝက်စီ ခွဲတဲ့အတွက် အလွှာ (level) အရေအတွက်က $\log n$ ပဲ ရှိပါတယ်။ အလွှာတစ်ခုစီမှာ စုစုပေါင်း အလုပ်က $O(n)$ ဆိုရင် စုစုပေါင်း $O(n \log n)$ ဖြစ်ပါတယ်။ ဒါက basic sort တွေရဲ့ $O(n^2)$ ထက် များစွာ မြန်တဲ့ အကြောင်းရင်း ဖြစ်ပါတယ်။



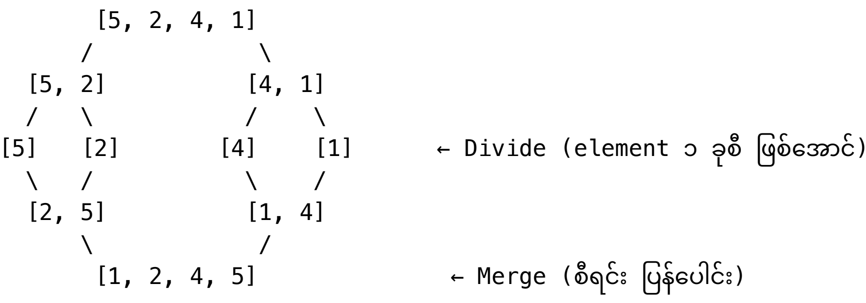
> **ဘယ်နေရာတွေမှာ တွေ့ရသလဲ?** ဒီစာအုပ်မှာ divide and conquer ကို တွေ့ဖူးပြီးသား နေရာတွေ ရှိပါတယ် —
> - **Binary Search** (အခန်း ၉) — search space ကို တစ်ဝက်စီ ခွဲ (subproblem ၁ ခုကိုသာ ဆက်ဖြေ)။
> - **Merge Sort / Quick Sort** (ဒီအခန်း) — array ကို ခွဲ၊ စီ၊ ပြန်ပေါင်း။
>
> ဒါကြောင့် divide and conquer ကို **သီးခြား chapter** အဖြစ် မလေ့လာဘဲ၊ သူ့ကို သုံးတဲ့ algorithm တွေကို လေ့လာခြင်းဖြင့် နားလည်သွားမှာ ဖြစ်ပါတယ်။

အခု Merge Sort နဲ့ Quick Sort တို့ဟာ ဒီ ၃ အဆင့်ကို ဘယ်လို အသုံးချလဲ ကြည့်ရအောင်။

Merge Sort

Merge Sort ဟာ idea ၂ ဆင့်နဲ့ အလုပ်လုပ်ပါတယ် —

1. **Divide:** array ကို element တစ်ခုစီ ဖြစ်သွားအောင် တစ်ဝက်စီ ဆက်တိုက် ခွဲသည်။
2. **Merge:** ခွဲထားတဲ့ ပိုင်းသေးတွေကို **စီရင်း ပြန်ပေါင်း** ဖို့လိုပါတယ်။ အပိုင်း ၂ ခုစလုံး စီပြီးသား ဖြစ်နေတဲ့အတွက် ပေါင်းတာ မြန်တယ်။



Merge လုပ်ပုံ အသေးစိတ်: ဘာကြောင့် မြန်သလဲ ဆိုတာ နားလည်ဖို့ `[2, 5]` နဲ့ `[1, 4]` ကို ပေါင်းတာ ကြည့်ရအောင်။ ၂ ခုလုံးက စီပြီးသား ဖြစ်လို့၊ ရှေ့ဆုံး ၂ ခုကို နှိုင်းယှဉ်ပြီး အငယ်ကို အရင်ထုတ်ရုံ ဖြစ်ပါတယ် —

နှိုင်းယှဉ်	အငယ်ကို ထုတ်	ရလဒ်
`2` vs `1`	`1`	`[1]`
`2` vs `4`	`2`	`[1, 2]`
`5` vs `4`	`4`	`[1, 2, 4]`
`5` (ကျန်)	`5`	`[1, 2, 4, 5]`

```
void mergeSort(int[] arr, int left, int right) {
    if (left >= right) return; // Base Case: element ၁ ခု (သို့) ၀ ခုသာ ကျန် → စီပြီးသား

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);       // ဘယ်ခြမ်း စီသည်
    mergeSort(arr, mid + 1, right);  // ညာခြမ်း စီသည်
    merge(arr, left, mid, right);    // စီပြီးသား ၂ ခြမ်းကို ပြန်ပေါင်းသည်
}

void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;
    // ၂ ခြမ်းကို နှိုင်းယှဉ်ပြီး အငယ်ကို အရင်ထည့်သည်
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) temp[k++] = arr[i++]; // <= က stability ထိန်းသည်
        else temp[k++] = arr[j++];
    }
    // ကျန်တဲ့ element တွေ ထည့်သည်
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];
    // ပေါင်းပြီးသား result ကို မူလ array ထဲ ပြန်ကူးသည်
    System.arraycopy(temp, 0, arr, left, temp.length);
}
```

> **ဘာကြောင့် $O(n \log n)$ လဲ?** array ကို တစ်ဝက်စီ ခွဲတဲ့အတွက် **အလွှာ (level)** $\log n$ **ခု** ရှိပါတယ်။ အလွှာတစ်ခုစီမှာ element အားလုံး (n ခု) ကို merge လုပ်ရတဲ့အတွက် $O(n)$ ကြာသည်။ စုစုပေါင်း $O(n \log n)$ ဖြစ်ပါတယ်။

> **အားသာချက်:** Merge Sort ဟာ **Stable** ဖြစ်ပြီး worst case မှာတောင် $O(n \log n)$ အာမခံပါတယ် (Quick Sort လို worst case မရှိ)။ Linked List စီတာ၊ ဒါမှမဟုတ် memory ထဲ မဆံ့တဲ့ data ကြီးကို file နဲ့ ပိုင်းခွဲ စီတဲ့ **External Sorting** အတွက် အကောင်းဆုံး ဖြစ်ပါတယ်။
> **အားနည်းချက်:** Merge လုပ်ဖို့ array အပို ($O(n)$ extra space) လိုပါတယ်။

Quick Sort

Quick Sort ဟာ Merge Sort နဲ့ ဆန့်ကျင်ဘက် — **ခွဲတဲ့အပိုင်းမှာ အလုပ်လုပ်ပြီး၊ ပေါင်းတဲ့အပိုင်းမှာ ဘာမှ မလုပ်ရ** သည့် နည်းလမ်း ဖြစ်ပါတယ်။

Element တစ်ခုကို **Pivot** အဖြစ် ရွေးပြီး၊ pivot ထက် **ငယ်တာတွေကို ဘယ်ဘက်၊ ကြီးတာတွေကို ညာဘက်** ခွဲ (partition) ထားပါတယ်။

ဒီ partition ပြီးတဲ့အခါ pivot က **သူ့ နေရာအမှန် ရောက်သွား** ပါပြီ။ ပြီးတော့ ဘယ်ခြမ်းနဲ့ ညာခြမ်းကို သီးခြားစီ ဒီအတိုင်း ထပ်လုပ်သွားပါတယ်။

`[5, 2, 8, 1, 9, 3]` မှာ `pivot = 3` (နောက်ဆုံး element) ရွေးပြီး partition လုပ်တာ —

`i` ရဲ့ role: `i` က "pivot ထက်ငယ်တဲ့ element တွေ စုထားတဲ့ zone ရဲ့ နောက်ဆုံး index" ဖြစ်ပါတယ်။ စတင်တဲ့အချိန်မှာ zone က အလွတ် ဖြစ်တဲ့အတွက် `i = -1` (array ထဲ မရောက်သေးတဲ့ မြောက်ဘက် edge) မှ စပါတယ်။ pivot ထက်ငယ်တဲ့ element တစ်ခု တွေ့တိုင်း `i` တိုးပြီး ထို element ကို zone ထဲ ဆွဲယူပါတယ်။ loop ပြီးသောအခါ index `0` ကနေ `i` ကြားက element အားလုံး pivot ထက် ငယ်ပါတယ်။ `i = -1` မှ စပါတယ်

Step	j	arr[j]	arr[j] < pivot (3)?	လုပ်ဆောင်ချက်	Array အခြေအနေ	i
စမှတ်	—	—	—	—	`[5, 2, 8, 1, 9, 3]`	-1
1	0	5	No	ဘာမှ မလုပ်	`[5, 2, 8, 1, 9, 3]`	-1
2	1	2	Yes	i++→0, swap(0,1)	`[2, 5, 8, 1, 9, 3]`	0
3	2	8	No	ဘာမှ မလုပ်	`[2, 5, 8, 1, 9, 3]`	0
4	3	1	Yes	i++→1, swap(1,3)	`[2, 1, 8, 5, 9, 3]`	1
5	4	9	No	ဘာမှ မလုပ်	`[2, 1, 8, 5, 9, 3]`	1
ပြီး	—	—	—	swap(i+1=2, high=5)	`[2, 1, **3**, 5, 9, 8]`	1

- Pivot `3` က index 2 တွင် သူ့နေရာအမှန် ရောက်သွားပြီ
- ဘယ်ခြမ်း `[2, 1]` နဲ့ ညာခြမ်း `[5, 9, 8]` ကို သီးခြားစီ ထပ်လုပ်သည်

```
void quickSort(int[] arr, int low, int high) {
    if (low >= high) return; // Base Case: element ၁ ခု (သို့) ၀ ခု

    int pivotIndex = partition(arr, low, high); // pivot ကို correct position တွင် ထားသည်
    quickSort(arr, low, pivotIndex - 1);  // ဘယ်ခြမ်း (ပိုသေး)
```

```
        quickSort(arr, pivotIndex + 1, high); // ညာခြမ်း (ပိုကြီး)
    }

    int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // နောက်ဆုံး element ကို pivot အဖြစ်ရွေးသည်
        int i = low - 1;       // "ငယ်တာတွေရဲ့ boundary (နယ်နိမိတ်)"

        for (int j = low; j < high; j++) {
            // pivot ထက် ငယ်ရင် ဘယ်ဘက်ပိုင်းသို့ ရွှေ့သည်
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        // pivot ကို "ငယ်တာ" နဲ့ "ကြီးတာ" အကြား မှန်ကန်တဲ့နေရာ ထားသည်
        int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
        return i + 1;
    }
```

> **Pivot ရွေးချယ်ခြင်း ဘာကြောင့် အရေးကြီးသလဲ?** Pivot က partition ၂ ခြမ်းကို **ညီညီ ခွဲပေး**ရင် အကောင်းဆုံး ($O(n \log n)$) ဖြစ်ပါတယ်။ ဒါပေမယ့် မကောင်းရွေးမိရင် — ဥပမာ **စီပြီးသား array** `[1,2,3,4,5]` မှာ အမြဲ နောက်ဆုံးကို pivot ရွေးရင် — ခြမ်းတစ်ခုက အလွတ်၊ တစ်ခြမ်းက ကျန်အကုန် ဖြစ်ပြီး $O(n^2)$ ဆုတ်ယုတ်သွားပါတယ်။ ဒါကြောင့် လက်တွေ့မှာ —
> - Pivot ကို **random** ရွေးခြင်း၊
> - **"median-of-three"** (ပထမ၊ အလယ်၊ နောက်ဆုံး ၃ ခုရဲ့ အလယ်ကို ရွေး) နည်း — တို့ကို သုံးကြပါတယ်။

> **အားသာချက်:** In-place ($O(\log n)$ recursion stack သာ) ဖြစ်ပြီး cache-friendly (ဘေးချင်းကပ် memory ကို သုံး) သလို memory လည်း သက်သာတဲ့အတွက် လက်တွေ့ workload အများစုမှာ Merge Sort ထက် **မကြာခဏ ပိုမြန်** လေ့ရှိပါတယ်။ Java ရဲ့ primitive array sort (`Arrays.sort(int[])`) ဟာ **Dual-Pivot Quicksort** ကို သုံးပြီး၊ object array sort (`Arrays.sort(Object[])`) ကတော့ stable ဖြစ်တဲ့ **Timsort** ကို သုံးပါတယ်။
> **အားနည်းချက်:** Worst case $O(n^2)$ ဖြစ်နိုင်ပြီး၊ Stable မဟုတ်ပါ။

Heap Sort

Heap Sort ဟာ data ကို **Heap** ဆိုတဲ့ tree structure အဖြစ် အသုံးချတဲ့ နည်းလမ်း ဖြစ်ပါတယ်။ **Max-heap** ဆိုတာ "မိဘ node က သားသမီး node ထက် အမြဲ ကြီးတယ်" ဆိုတဲ့ စည်းမျဉ်းရှိတဲ့ binary tree ဖြစ်လို့၊ **အကြီးဆုံး element က အမြဲ ထိပ် (root) မှာ ရှိ**ပါတယ်။

Heap Sort က ဒီ feature ကို အသုံးချပါတယ် —

1. Array ကို max-heap အဖြစ် တည်ဆောက်သည်။
2. ထိပ်က အကြီးဆုံးကို နောက်ဆုံးနေရာနဲ့ လဲ၊ heap အရွယ် ၁ ခု လျှော့။
3. ထိပ်မှာ heap rule ပျက်သွားလို့ ပြန်ချိန်ညှိ (heapify) သည်။

(Heap structure ရဲ့ အသေးစိတ်ကို အခန်း ၁၂ မှာ ဆက်လေ့လာပါမယ်။ ဒီနေရာမှာ idea ကိုသိအောင် ပြောပြထားခြင်း ဖြစ်ပါသည်)

```
void heapSort(int[] arr) {
    int n = arr.length;
    // Step 1: array ကို max-heap အဖြစ် တည်ဆောက်သည် (နောက်ဆုံး မိဘ node ကနေ စ)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    // Step 2: အကြီးဆုံး (root) ကို နောက်ဆုံးနဲ့ လဲ၊ heap အရွယ် လျှော့ပြီး ပြန်ချိန်ညှိ
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;
        heapify(arr, i, 0);
    }
}

// node i ကို သူ့ subtree အတွင်း မှန်ကန်တဲ့နေရာ ဆင်းသွားအောင် ချိန်ညှိသည်
void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1, right = 2 * i + 2; // array ထဲ child တွေရဲ့ index
    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest != i) {
        int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;
        heapify(arr, n, largest); // အောက်ကို ဆက်ချိန်ညှိသည်
    }
}
```

> **အားသာချက်:** In-place ($O(1)$ extra space) ဖြစ်ပြီး worst case မှာတောင် $O(n \log n)$ အာမခံပါတယ်။ Quick Sort လို worst case ပြဿနာ မရှိဘဲ memory လည်း သက်သာတဲ့အတွက် memory အကန့်အသတ်ရှိတဲ့ system တွေမှာ ကောင်းပါတယ်။
> **အားနည်းချက်:** Stable မဟုတ်သလို၊ heap က memory ထဲ ခုန်ကျော် ဖတ်ရတဲ့အတွက် cache-friendly မဖြစ်ပါ။ ဒါကြောင့် လက်တွေ့မှာ Quick Sort လောက် မမြန်ပါ။

အပိုင်း ၃ — Non-comparison Sorting

အပေါ်က sort တွေအားလုံးဟာ element တွေကို **နှိုင်းယှဉ်** ပြီး စီတာ ဖြစ်ပါတယ်။ သင်္ချာအရ သက်သေပြထားတာက — **နှိုင်းယှဉ်ပြီး စီတဲ့ algorithm ဘယ်ဟာမှ $O(n \log n)$ ထက် မမြန်နိုင်ပါ။** ဒါကို **comparison sort lower bound** လို့ ခေါ်ပါတယ်။

> **ဘာကြောင့်လဲ?** Element `n` ခုကို စီနိုင်တဲ့ ဖြစ်နိုင်ခြေ အစီအစဉ် (permutation) က $n!$ မျိုး ရှိပါတယ်။ နှိုင်းယှဉ်မှု တစ်ခုစီက "ကြီးလား/ငယ်လား" အဖြေ ၂ မျိုးပဲ ထွက်လို့၊ ဖြစ်နိုင်ခြေ $n!$ ခုကို ခွဲခြားဖို့ အနည်းဆုံး $\log_2(n!) \approx n \log n$ ကြိမ် နှိုင်းယှဉ်ဖို့ လိုပါတယ်။

ဒါဆို $O(n \log n)$ ထက် ဘယ်လို ပိုမြန်အောင် လုပ်မလဲ? **နှိုင်းယှဉ်တာ မလုပ်တော့ဘဲ** data ရဲ့ သဘော (ဥပမာ - "integer ဖြစ်တယ်၊ ၀ ကနေ ၁၀၀ ကြားပဲ ရှိတယ်") ကို တိုက်ရိုက် အသုံးချရင် ပိုမြန်အောင် လုပ်နိုင်ပါတယ်။

Counting Sort

Counting Sort ဟာ element တစ်ခုစီ **ဘယ်နှစ်ကြိမ် ပါသလဲ ရေတွက် (count)** ပြီး၊ အဲ့ count ကို သုံးကာ နေရာချ စီတဲ့ နည်းလမ်း ဖြစ်ပါတယ်။

`[2, 5, 3, 0, 2, 3, 0]` (range ၀-၅) ကို စီကြည့်ရအောင် —

```
Step 1 — ရေတွက်:
  value:  0  1  2  3  4  5
  count:  2  0  2  2  0  1     (0 က ၂ ကြိမ်၊ 2 က ၂ ကြိမ်၊ ...)

Step 2 — count အလိုက် ပြန်ဖြန့်:
  0 ကို ၂ ခု → 0, 0
  2 ကို ၂ ခု → 2, 2
  3 ကို ၂ ခု → 3, 3
  5 ကို ၁ ခု → 5
  ရလဒ်: [0, 0, 2, 2, 3, 3, 5]
```

```
int[] countingSort(int[] arr, int k) { // k = အကြီးဆုံး တန်ဖိုး
    int[] count = new int[k + 1];
    // Step 1: element တစ်ခုစီ ဘယ်နှစ်ကြိမ် ပါသလဲ ရေတွက်သည်
    for (int num : arr) count[num]++;

    int[] result = new int[arr.length];
    int idx = 0;
    // Step 2: count အလိုက် ငယ်ရာကနေ ကြီးရာ ပြန်ဖြန့်ချသည်
    for (int num = 0; num <= k; num++) {
        while (count[num] > 0) {
            result[idx++] = num;
            count[num]--;
        }
    }
    return result;
}
```

ဒီနည်းက နှိုင်းယှဉ်တာ လုံးဝ မလုပ်ဘဲ၊ element `n` ခုနဲ့ range `k` ကိုသာ ဖြတ်သန်းရလို့ $O(n+k)$ ဖြစ်ပါတယ်။ Range `k` က `n` နဲ့ နီးစပ်ရင် (k မကြီးရင်) ဒါက $O(n)$ နီးပါး ဖြစ်ပါတယ်။

> **Stability:** အပေါ်က ရိုးရှင်းတဲ့ "count လုပ် ပြီး ပြန်ဖြန့်" version က ကိန်းသက်သက်အတွက် အဆင်ပြေပါတယ်။ ဒါပေမယ့် Counting Sort ကို **stable** ဖြစ်စေဖို့ (object တွေကို field အလိုက် စီတဲ့အခါ၊ ဒါမှမဟုတ် Radix Sort ထဲ သုံးတဲ့အခါ) **prefix sum (cumulative count)** နဲ့ **output array သီးခြား** သုံးပြီး ရေးရပါတယ်။

Stable Counting Sort (prefix sum နည်း):

`[2, 5, 3, 0, 2, 3, 0]` ကို ဥပမာ ယူပြည့်ရအောင် —

```
Step 1 — count:       index: 0  1  2  3  4  5
                      count: 2  0  2  2  0  1

Step 2 — prefix sum (count[i] += count[i-1]):
                      count: 2  2  4  6  6  7
                      (meaning: 0 ထက် ငယ်/ညီ k ခုက index 0..1 မှာ ရှိမယ်)

Step 3 — နောက်ဆုံးကနေ ရှေ့ဆီ ဖြတ် (stable ဖြစ်ဖို့):
  arr[6]=0 → count[0]=2 → result[1]=0, count[0]--→1
  arr[5]=3 → count[3]=6 → result[5]=3, count[3]--→5
  arr[4]=2 → count[2]=4 → result[3]=2, count[2]--→3
  arr[3]=0 → count[0]=1 → result[0]=0, count[0]--→0
  arr[2]=3 → count[3]=5 → result[4]=3, count[3]--→4
  arr[1]=5 → count[5]=7 → result[6]=5, count[5]--→6
  arr[0]=2 → count[2]=3 → result[2]=2, count[2]--→2

Result: [0, 0, 2, 2, 3, 3, 5]
```

"နောက်ဆုံးကနေ ရှေ့ဆီ" ဖြတ်ရတဲ့ အကြောင်းရင်း — တန်ဖိုးတူ element တွေ (ဥပမာ `2` ၂ ခု) ကို မူလ array ထဲ နောက်မှာ ရှိတဲ့ဟာကို output ထဲ နောက်မှာ ထားဖို့ ဖြစ်ပါတယ်။ ဒါမှ မူလ relative order မပျက်ပါ။

```
int[] stableCountingSort(int[] arr, int k) {
    int[] count = new int[k + 1];
    for (int num : arr) count[num]++;

    // prefix sum: count[i] = arr ထဲ i နဲ့ ညီ/ငယ်တဲ့ element ဘယ်နှစ်ခု ရှိသလဲ
    for (int i = 1; i <= k; i++) count[i] += count[i - 1];

    int[] result = new int[arr.length];
    // နောက်ဆုံးကနေ ရှေ့ဆီ ဖြတ်မှ stable ဖြစ်သည်
    for (int i = arr.length - 1; i >= 0; i--) {
        result[--count[arr[i]]] = arr[i];
    }
    return result;
}
```

> **သတိ:** တန်ဖိုး range `k` က အလွန်ကြီးရင် (ဥပမာ - ၁ ကနေ သန်းပေါင်းများစွာ၊ ဒါမှမဟုတ် negative/decimal တွေ ပါ) count array က memory အများကြီး ယူသွားလို့ မသင့်တော်တော့ပါ။ "Integer ဖြစ်ပြီး range သေး" တဲ့အခါမှ သုံးပါ။

Radix Sort

Counting Sort က range ကြီးရင် မသုံးနိုင်တဲ့ ပြဿနာကို Radix Sort က ဖြေရှင်းပါတယ်။ ကိန်းတွေကို **ဂဏန်းတစ်လုံးချင်း (digit by digit)** —နောက်ဆုံး ကနေ ကြီးတဲ့နေရာဆီ — Counting Sort နဲ့ ထပ်ခါ စီသွားတဲ့ နည်းလမ်း ဖြစ်ပါတယ်။ ဂဏန်းတစ်လုံးက ၀-၉ ပဲ ရှိလို့ range သေးပြီး Counting Sort လုပ်လို့ ရပါတယ်။

```
[170, 45, 75, 90, 2, 802]
ones (နောက်ဆုံးဂဏန်း) အလိုက်  → [170, 90, 2, 802, 45, 75]
tens (ဆယ်လီ) အလိုက်           → [2, 802, 45, 170, 75, 90]
```

```
hundreds (ရာလီ) အလိုက်      → [2, 45, 75, 90, 170, 802]  ← စီပြီး
```

Ones-digit pass အသေးစိတ်: `[170, 45, 75, 90, 2, 802]` ကို ones digit (နောက်ဆုံးဂဏန်း) အလိုက် stable sort လုပ်ပုံ —

element	ones digit
170	0
45	5
75	5
90	0
2	2
802	2

digit 0 bucket → `170, 90` (မူလ အစီအစဉ်အတိုင်း stable)
digit 2 bucket → `2, 802`
digit 5 bucket → `45, 75`
ပြန်ဆက် → `[170, 90, 2, 802, 45, 75]`

ဒီ ones-digit pass မှာ stable ဖြစ်မှ digit 0 bucket ထဲ `170` က `90` ရှေ့မှာ ကျန်ပါတယ်။ မူလ order မပျက်ဘဲ ထိန်းထားတဲ့အတွက် tens, hundreds pass တွေ ဆက်လုပ်ရင် မှန်ကန်သော ရလဒ် ရပါတယ်။

ဂဏန်း `d` လုံးရှိရင် Counting Sort ကို `d` ကြိမ် လုပ်ရလို့ $O(d \times (n + k))$ ဖြစ်ပါတယ်။ Radix Sort အလုပ်လုပ်ဖို့ digit-level Counting Sort က **stable** ဖြစ်ဖို့ မဖြစ်မနေ လိုအပ်ပါတယ် — မဟုတ်ရင် ရှေ့ digit တွေ စီထားတာ ပျက်သွားမှာ ဖြစ်ပါတယ်။

Bucket Sort

Bucket Sort ဟာ တန်ဖိုးတွေကို **bucket (အကန့်) အများ** ထဲ ခွဲဝေထည့်ပြီး၊ bucket တစ်ခုစီကို သီးခြားစီ (များသောအားဖြင့် Insertion Sort နဲ့) စီကာ နောက်ဆုံး bucket တွေကို အစဉ်လိုက် ပြန်ဆက်တဲ့ နည်းလမ်း ဖြစ်ပါတယ်။

```
[0.78, 0.17, 0.39, 0.26, 0.72, 0.94]   (၀ ကနေ ၁ ကြား decimal)
bucket ၁၀ ခု ခွဲ (တန်ဖိုး × 10):
  bucket[1]: 0.17
  bucket[2]: 0.26
  bucket[3]: 0.39
  bucket[7]: 0.78, 0.72  → သီးခြားစီ → 0.72, 0.78
  bucket[9]: 0.94
ပြန်ဆက်: [0.17, 0.26, 0.39, 0.72, 0.78, 0.94]
```

Data က အပိုင်းအခြားတစ်ခုအတွင်း **ညီညာ ဖြန့်ကျက် (uniformly distributed)** နေတဲ့အခါ bucket တစ်ခုစီမှာ element နည်းနည်းပဲ ရှိလို့ $O(n)$ နီးပါး ဖြစ်ပါတယ်။ ဒါပေမယ့် တန်ဖိုးတွေ bucket တစ်ခုထဲ စုနေရင်တော့ $O(n^2)$ ဆုတ်နိုင်ပါတယ်။

> **အကျဉ်းချုပ်:** Non-comparison sort တွေဟာ "data က integer လား၊ range သေးလား၊ ညီညာဖြန့်နေလား" ဆိုတဲ့ **အခြေအနေ ပြည့်မှ** သုံးနိုင်ပါတယ်။ အထွေထွေ data (object, string နဲ့ custom rule, range မသိ) အတွက်တော့ Quick/Merge/Heap Sort ပဲ သုံးရပါမယ်။

Object များကို Field အလိုက် စီခြင်း (Custom Comparator)

လက်တွေ့ development မှာ ကိန်းသက်သက် မဟုတ်ဘဲ **object** (user, product, order) တွေကို field တစ်ခုခုအလိုက် စီရတာ အများဆုံးပါ။ ဒီအတွက် sort algorithm ကိုယ်တိုင် မရေးဘဲ၊ built-in sort ကို **Custom Comparator** နဲ့ သုံးပါတယ်။ Comparator ဆိုတာ "element ၂ ခု ဘယ်ဟာ အရင်လာရမလဲ" ဆုံးဖြတ်ပေးတဲ့ rule (function) ဖြစ်ပါတယ်။

Comparator တစ်ခုက `a` , `b` ၂ ခုကို ယူပြီး —

- **negative** ပြန်ရင် → `a` က ရှေ့လာရမည်၊
- **positive** ပြန်ရင် → `b` က ရှေ့လာရမည်၊
- **0** ပြန်ရင် → အတူတူ (အစီအစဉ် မပြောင်း) — ဆိုပြီး ဆုံးဖြတ်ပါတယ်။

```java
import java.util.Arrays;
import java.util.Comparator;

class User {
    String name;
    int age;
    User(String name, int age) { this.name = name; this.age = age; }
}

class Solution {
    void sortUsers(User[] users) {
        // age အငယ်ကနေ အကြီး (ascending) စီသည်
        Arrays.sort(users, Comparator.comparingInt(u -> u.age));

        // Multi-field: age အလိုက် အရင်စီ၊ age တူရင်မှ name အလိုက် စီသည်
        Arrays.sort(users, Comparator
            .comparingInt((User u) -> u.age)
            .thenComparing(u -> u.name));

        // age အကြီးကနေ အငယ် (descending) စီသည်
        Arrays.sort(users, Comparator.comparingInt((User u) -> u.age).reversed());
    }
}
```

> **Stability ဘာကြောင့် ဒီနေရာ အရေးကြီးသလဲ?** Java ရဲ့ object sort (`Arrays.sort(Object[])`) ဟာ **Timsort** (Merge Sort + Insertion Sort ပေါင်းစပ်ထား၊ stable) ကို သုံးပါတယ်။ Stable ဖြစ်လို့ "name အလိုက် အရင်စီ၊ ပြီးမှ age အလိုက် ထပ်စီ" ဆိုတဲ့ **chained sort** ကိုလည်း မှန်ကန်စွာ လုပ်နိုင်ပါတယ်။ (Primitive `int[]` တွေအတွက်တော့ Dual-Pivot Quick Sort ကို သုံးပါတယ် — primitive မှာ တန်ဖိုးတူ element တွေ ခွဲမရလို့ stability က အရေးမကြီးပါ။)

Real-world Examples

Sorting ဟာ backend, frontend, data processing အားလုံးမှာ နေ့စဉ် ကြုံရတဲ့ လုပ်ငန်း ဖြစ်ပါတယ်။

- **Sort users by created date** — user list ကို အသစ်ဆုံး/အဟောင်းဆုံး အလိုက် ပြသခြင်း။
- **Sort products by price** — e-commerce မှာ "Price: Low to High" filter။
- **Sort transactions by amount** — ငွေပမာဏ အကြီးဆုံး transaction တွေ ရှာခြင်း။
- **Sort leaderboard by score** — game ranking ကို score အလိုက် စီခြင်း (score တူရင် အချိန် အလိုက် ဆက်စီ)။
- **Sort logs by timestamp** — log entry တွေကို အချိန်အလိုက် စီပြီး debug လုပ်ခြင်း။

ဒီ use case တွေအတွက် language ရဲ့ built-in sort ($O(n \log n)$) ကို custom comparator နဲ့ သုံးတာ အကောင်းဆုံး ဖြစ်ပါတယ်။ ကိုယ်တိုင် sort algorithm ရေးစရာ မလိုပါ။ ဒါပေမယ့် မေးရမယ့် မေးခွန်း က — **"ဒီ sort က stable ဖြစ်လား?"** ဆိုတာပါ။ Programming language အများစုမှာ optimize လုပ် ထားတဲ့ built-in sort ပါပေမယ့်၊ **stable ဖြစ်မဖြစ်က language နဲ့ data type ပေါ်မူတည်** ပါတယ် (ဥပမာ - Java မှာ object sort က stable ဖြစ်ပေမယ့် primitive `int[]` sort က မဟုတ်ပါ)။ Multi-field sorting လုပ်မယ်ဆိုရင် stable ဖြစ်တာ အရေးကြီးလို့ အရင် စစ်ဆေးသင့်ပါတယ်။

> **Timsort:** Python နဲ့ Java (object sort) တို့ဟာ **Timsort** (သို့) Timsort-inspired algorithm ကို သုံးကြပါတယ်။ Timsort ဟာ **Merge Sort နဲ့ Insertion Sort ကို ပေါင်းစပ်ထား**ပြီး stable ဖြစ် ပါတယ်။ Data ဟာ အပိုင်းအားဖြင့် စီပြီးသား (partially sorted) ဖြစ်တဲ့အခါ — real-world data အများစုက ဒီလို ဖြစ်တတ် — အလွန်မြန်ပါတယ်။ ဒါက အပေါ်မှာ ပြောခဲ့တဲ့ Insertion Sort ရဲ့ "nearly sorted data မှာ မြန်တယ်" ဆိုတဲ့ အချက်ကို real-world system တွေမှာ အသုံးချထား တာ ဖြစ်ပါတယ်။

Questions

Sorting concept တွေကို လက်တွေ့ ပြဿနာတွေနဲ့ ချိတ်ဆက်ပြီး အသုံးချနည်း ၅ မျိုးကို တစ်ဆင့်ချင်း လေ့လာကြည့်ရအောင်။ မေးခွန်းတွေဟာ "sort ကို ဘယ်လို အသုံးချမလဲ" ဆိုတဲ့ pattern ကို ဖော်ပြ ထားပါတယ်။

၁။ Merge Sorted Array

စီပြီးသား (ascending) array ၂ ခု `nums1` နဲ့ `nums2` ပေးထားသည်။ `nums1` မှာ `nums2` ကို ဆံ့အောင် နေရာ (`m + n` အရွယ်) ထားပြီးသား ဖြစ်သည် (နောက်ပိုင်းမှာ `0` တွေ ဖြည့်ထား)။ နှစ်ခုကို **တစ်ခုတည်း စီပြီးသား array** အဖြစ် ပေါင်းပါ။ `nums1` ထဲမှာတင် (in-place) ပြုလုပ်ပါ။

Example 1:

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
```

ရှင်းလင်းချက်

နှစ်ခုစလုံး စီပြီးသား ဖြစ်လို့ **Merge Sort ရဲ့ merge အဆင့်** ကို တိုက်ရိုက် သုံးနိုင်ပါတယ်။ ဒါပေမယ့် **အရှေ့ကနေ** စီရင် `nums1` ရဲ့ မူလ တန်ဖိုးတွေ ဖျက်မိနိုင်ပါတယ် (ဥပမာ - position 0 ကို ရေးလိုက်ရင် `nums1[0]=1` ပျောက်သွားမယ်)။

ဒါကို **အနောက်ဆုံး (အကြီးဆုံး) ကနေ** စပြီး ဖြည့်လိုက်ရင် ဖြေရှင်းနိုင်ပါတယ်။ `nums1` ရဲ့ အနောက်ပိုင်းက `0` တွေ (နေရာလွတ်) ဖြစ်လို့ ရေးချလို့ ရပါတယ်။

- Pointer `i` က `nums1` ရဲ့ နောက်ဆုံး တကယ့်တန်ဖိုး (`m-1`)၊ `j` က `nums2` ရဲ့ နောက်ဆုံး (`n-1`)။
- `k` က ဖြည့်မယ့်နေရာ (အနောက်ဆုံး)။
- `i` , `j` ၂ ခုထဲက **အကြီးကို** `k` နေရာမှာ ထား၊ ပြီးရင် pointer တွေ နောက်ဆုတ်။

> **Time Complexity:** $O(m+n)$ - element အားလုံးကို တစ်ကြိမ်စီ ကြည့်သောကြောင့်။
> **Space Complexity:** $O(1)$ - `nums1` ထဲမှာတင် ပြုလုပ်လို့ memory အပို မလို။

Java Solution

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1;      // nums1 ၏ နောက်ဆုံး တကယ့်တန်ဖိုး
        int j = n - 1;      // nums2 ၏ နောက်ဆုံး
        int k = m + n - 1;  // ဖြည့်မယ့် နေရာ (အနောက်ဆုံး)

        while (j >= 0) {
            // ၂ ခုထဲက အကြီးကို အနောက်ဆုံးနေရာ ထားသည်
            if (i >= 0 && nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        // nums1 ဘက် ကျန်ရင် နေရာမှန်ပြီးသား ဖြစ်လို့ ဆက်မလုပ်တော့
    }
}
```

၂။ Sort Colors

`0` (အနီ), `1` (အဖြူ), `2` (အပြာ) တွေပဲ ပါတဲ့ array `nums` ပေးထားသည်။ အရောင်တူတွေ စုနေအောင်၊ `0` → `1` → `2` အစီအစဉ်အတိုင်း **in-place** စီပါ။ Library sort မသုံးဘဲ ပြုလုပ်ပါ။

Example 1:

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

ရှင်းလင်းချက်

တန်ဖိုး ၃ မျိုးပဲ ရှိလို့ Counting Sort လုပ်လို့ ရပါတယ် (count ပြီး ပြန်ဖြန့်)။ ဒါပေမယ့် အဲ့ဒါက array ကို ၂ ပတ် ဖြတ်ရပါတယ်။ ဒီ ပြဿနာက **တစ်ပတ်တည်း (one pass)** နဲ့ ဖြေနိုင်ပါတယ်။

ဒါက **Dutch National Flag** ပြဿနာ (Edsger Dijkstra တင်ပြခဲ့) ဖြစ်ပြီး pointer ၃ ခု သုံးပါတယ် —

- `low` — `0` တွေ စုထားမယ့် boundary (နယ်နိမိတ်) (low ဘယ်ဘက်မှာ `0` တွေ ရှိ)။
- `high` — `2` တွေ စုထားမယ့် boundary (နယ်နိမိတ်) (high ညာဘက်မှာ `2` တွေ ရှိ)။
- `mid` — လက်ရှိ ကြည့်နေတဲ့ နေရာ။

`mid` က element ကို ကြည့်ပြီး —

- `0` ဆို → `low` နဲ့ လဲ၊ `low` နဲ့ `mid` ၂ ခုလုံး တိုး။
- `1` ဆို → နေရာမှန်ပြီး → `mid` သာ တိုး။
- `2` ဆို → `high` နဲ့ လဲ၊ `high` လျှော့ (လဲထားတဲ့ element ကို ပြန်စစ်ဖို့ `mid` မတိုး)။

`[2, 0, 2, 1, 1, 0]` ကို တစ်ဆင့်ချင်း ကြည့်ရအောင် —

Step	nums[mid]	လုပ်ဆောင်ချက်	Array	low	mid	high
စမှတ်	—	—	`[2, 0, 2, 1, 1, 0]`	0	0	5
1	2	swap(mid=0, high=5), high--	`[0, 0, 2, 1, 1, 2]`	0	0	4
2	0	swap(low=0, mid=0), low++, mid++	`[0, 0, 2, 1, 1, 2]`	1	1	4
3	0	swap(low=1, mid=1), low++, mid++	`[0, 0, 2, 1, 1, 2]`	2	2	4
4	2	swap(mid=2, high=4), high--	`[0, 0, 1, 1, 2, 2]`	2	2	3
5	1	mid++	`[0, 0, 1, 1, 2, 2]`	2	3	3
6	1	mid++	`[0, 0, 1, 1, 2, 2]`	2	4	3

ပြီး	—	mid > high → ရပ်	[0, 0, 1, 1, 2, 2]	—	—	—

Step 1 နဲ့ Step 4 မှာ `mid` မတိုးတဲ့ အကြောင်းရင်း — `high` ဘက်ကနေ လဲလာတဲ့ element ဘာ တန်ဖိုးရှိမှန်း မသိသေးတဲ့အတွက် ထဲ element ကို ထပ်စစ်ဖို့ `mid` ကို တိုး မချနေဘဲ ထားတာ ဖြစ်ပါတယ်။

> **Time Complexity:** $O(n)$ - တစ်ပတ်တည်း။
> **Space Complexity:** $O(1)$ - in-place။

Java Solution

```
class Solution {
    public void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;
        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums, low, mid);
                low++; mid++;
            } else if (nums[mid] == 1) {
                mid++; // 1 က အလယ်မှာ ရှိပြီးသား
            } else {
                swap(nums, mid, high);
                high--; // လဲထားတဲ့ တန်ဖိုးကို ပြန်စစ်ရန် mid မတိုး
            }
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i]; nums[i] = nums[j]; nums[j] = temp;
    }
}
```

၃။ Top K Frequent Elements

ကိန်းပြည့် array `nums` နဲ့ ကိန်း `k` ပေးထားသည်။ **Frequency အများဆုံး (most frequent)** ဖြစ်တဲ့ element `k` ခုကို ပြန်ပါ။

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1, 2]
Explanation: 1 က ၃ ကြိမ်၊ 2 က ၂ ကြိမ်၊ 3 က ၁ ကြိမ် ပါသည်။
```

ရှင်းလင်းချက်

ဖြေနည်း ၃ ဆင့် ရှိပါတယ် —

1. **Count:** Hash Map (အခန်း ၄) နဲ့ element တစ်ခုစီရဲ့ frequency ကို ရေတွက်သည် — `{1:3, 2:2, 3:1}` ။

2. **Sort by frequency:** frequency အလိုက် စီရမယ်။ Frequency ၏ တန်ဖိုးက အများဆုံး `n` (array length) ပဲ ဖြစ်နိုင်လို့၊ **frequency ကို index အဖြစ်** သုံးတဲ့ **Bucket Sort** idea နဲ့ sort လုပ်စရာ မလိုဘဲ $O(n)$ နဲ့ ရပါတယ်။
3. **Collect:** frequency အများဆုံး (bucket နောက်ဆုံး) ကနေ စပြီး `k` ခု ကောက်ယူသည်။

`bucket[f]` = frequency `f` အတိအကျ ရှိတဲ့ element တွေ စာရင်း ဖြစ်ပါတယ်။

> **ဘာကြောင့် Bucket Sort idea သုံးသလဲ?** ပုံမှန် sort လုပ်ရင် $O(n \log n)$ ဖြစ်ပေမယ့်၊ frequency က `1` ကနေ `n` ကြားသာ ဖြစ်လို့ index အဖြစ် တိုက်ရိုက် သုံးနိုင်ပြီး $O(n)$ ရအောင် လုပ်နိုင်ပါတယ်။

> **Time Complexity:** $O(n)$ - count + bucket fill + collect။
> **Space Complexity:** $O(n)$ - map နှင့် bucket အတွက်။

Java Solution

```java
import java.util.*;

class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // Step 1: frequency ရေတွက်သည်
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) {
            count.put(num, count.getOrDefault(num, 0) + 1);
        }

        // Step 2: frequency ကို index အဖြစ် bucket ထဲ ထည့်သည်
        List<Integer>[] bucket = new List[nums.length + 1];
        for (int num : count.keySet()) {
            int freq = count.get(num);
            if (bucket[freq] == null) bucket[freq] = new ArrayList<>();
            bucket[freq].add(num);
        }

        // Step 3: frequency အများဆုံး (နောက်ဆုံး bucket) ကနေ k ခု ကောက်သည်
        int[] result = new int[k];
        int idx = 0;
        for (int f = bucket.length - 1; f >= 0 && idx < k; f--) {
            if (bucket[f] != null) {
                for (int num : bucket[f]) {
                    result[idx++] = num;
                    if (idx == k) break;
                }
            }
        }
        return result;
    }
}
```

> **မှတ်ချက်:** ဒီပြဿနာကို **Heap (Priority Queue)** နဲ့လည်း $O(n \log k)$ နဲ့ ဖြေနိုင်ပါတယ်။ Heap က "အကြီးဆုံး/အငယ်ဆုံး `k` ခု" ပြဿနာတွေအတွက် အသုံးဝင်ပြီး၊ အခန်း ၁၂ မှာ ဆက်လေ့လာပါမယ်။

၄။ Sort Array by Custom Rule (Largest Number)

အပြုသဘော ကိန်းပြည့် array `nums` ပေးထားသည်။ သူတို့ကို ဆက်စပ်ရေးရင် **အကြီးဆုံး ဂဏန်း (largest number)** ရအောင် စီပြီး string အဖြစ် ပြန်ပါ။

Example 1:

```
Input: nums = [3, 30, 34, 5, 9]
Output: "9534330"
Explanation: ဆက်စပ်ရေးရင် အကြီးဆုံး ဖြစ်အောင် 9, 5, 34, 3, 30 အစီအစဉ်ဖြင့် စီသည်။
```

ရှင်းလင်းချက်

ဒါက **Custom Comparator** ရဲ့ စွမ်းအား ပြတဲ့ ပြဿနာ ဖြစ်ပါတယ်။ ကိန်းတန်ဖိုး အလိုက် ရိုးရိုး မစီနိုင်ပါ (ဥပမာ - `30 > 3` ပေမယ့် `"3"` က ရှေ့လာရင် ပိုကြီးတဲ့ `"330"` ရတယ်)။ ဒါကြောင့် **ဆက်စပ်လိုက်ရင် ဘယ်ဟာ ပိုကြီးလဲ** ဆိုတဲ့ rule နဲ့ စီရပါမယ်။

- `a` နဲ့ `b` ၂ ခုအတွက် `a+b` နဲ့ `b+a` (string concatenation) ကို နှိုင်းယှဉ်သည်။
- `"3"+"30" = "330"` vs `"30"+"3" = "303"` → `330 > 303` ဖြစ်လို့ `3` က `30` ရှေ့ လာရမည်။

ဒီ comparator က element ၂ ခုကြားက မှန်ကန်တဲ့ အစီအစဉ်ကို ဆုံးဖြတ်ပေးပြီး၊ sort က element အားလုံးကို အဲ့ rule အတိုင်း စီပေးပါတယ်။

> **Edge case:** array အကုန် `0` ဖြစ်နေရင် (ဥပမာ `[0, 0]`) ရလဒ်က `"00"` မဟုတ်ဘဲ `"0"` ဖြစ်ရမယ်။ ဒါကြောင့် ရှေ့ဆုံးက `"0"` ဖြစ်ရင် `"0"` ပဲ ပြန်ပါ။

> **Time Complexity:** $O(n \log n \cdot d)$ - sort လုပ်ရာတွင် string နှိုင်းယှဉ်မှု (d = digit length) ပါဝင်သောကြောင့်။
> **Space Complexity:** $O(n)$ - string array အတွက်။

Java Solution

```java
import java.util.*;

class Solution {
    public String largestNumber(int[] nums) {
        String[] strs = new String[nums.length];
        for (int i = 0; i < nums.length; i++) {
            strs[i] = String.valueOf(nums[i]);
        }
```

```
        // Custom rule: ဆက်စပ်ရင် ကြီးတဲ့ဟာ ရှေ့လာသည်
        Arrays.sort(strs, (a, b) -> (b + a).compareTo(a + b));

        // အကုန် "0" ဖြစ်နေရင် "0" ပြန်သည် (ဥပမာ [0,0] → "0")
        if (strs[0].equals("0")) return "0";

        StringBuilder sb = new StringBuilder();
        for (String s : strs) sb.append(s);
        return sb.toString();
    }
}
```

၅။ Merge Intervals

Interval (အပိုင်းအခြား) တွေ ပါတဲ့ array `intervals` ပေးထားသည် (`[start, end]`)။ **ထပ်နေ (overlapping)** တဲ့ interval တွေကို ပေါင်းစည်းပြီး၊ ထပ်မနေတဲ့ interval list ကို ပြန်ပါ။

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: [1,3] နဲ့ [2,6] က ထပ်နေ (2 ≤ 3) လို့ [1,6] အဖြစ် ပေါင်းသည်။
```

ရှင်းလင်းချက်

ဒါက **Sorting ကို preprocessing အဖြစ် အသုံးချတဲ့ classic ပြဿနာ** ဖြစ်ပါတယ်။ Interval တွေ ရောနေတဲ့အခါ ဘယ်ဟာတွေ ထပ်နေလဲ ရှာဖို့ ခက်ပါတယ်။ ဒါပေမယ့် **start အလိုက် အရင်စီ** လိုက်ရင်၊ ထပ်နေတဲ့ interval တွေဟာ **ဘေးချင်းကပ်** နေမှာ ဖြစ်လို့ တစ်ပတ်တည်း လိုက်ပေါင်းနိုင်ပါတယ်။

- `intervals` ကို start အလိုက် sort လုပ်သည်။
- လက်ရှိ interval ရဲ့ start က ရှေ့ (current) interval ရဲ့ end ထက် **ငယ်/ညီ** ရင် (ထပ်နေရင်) → current ရဲ့ end ကို ၂ ခုထဲက အကြီးအဖြစ် ချဲ့သည်။
- မထပ်ရင် → current ကို result ထဲ ထည့်ပြီး interval အသစ်အဖြစ် ဆက်သည်။

> **Time Complexity:** $O(n \log n)$ - sort က အဓိက ကုန်ကျစရာ (ပေါင်းတဲ့ loop က $O(n)$ သာ)။
> **Space Complexity:** $O(n)$ - result list အတွက်။

Java Solution

```java
import java.util.*;

class Solution {
    public int[][] merge(int[][] intervals) {
        // start အလိုက် စီသည် (preprocessing)
        // a[0] - b[0] သုံးရင် ကိန်းကြီးတွေမှာ integer overflow ဖြစ်နိုင်လို့ Integer.compare သုံးသည်
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        List<int[]> result = new ArrayList<>();
```

```
        int[] current = intervals[0];
        result.add(current);

        for (int[] interval : intervals) {
            if (interval[0] <= current[1]) {
                // ထပ်နေသည် → end ကို ချဲ့သည်
                current[1] = Math.max(current[1], interval[1]);
            } else {
                // မထပ် → interval အသစ် စသည်
                current = interval;
                result.add(current);
            }
        }
        return result.toArray(new int[result.size()][]);
    }
}
```

> **မှတ်ချက်:** Interval ပြဿနာတွေ (booking, meeting room, schedule) အများစုဟာ **start သို့မဟုတ် end အလိုက် စီခြင်း** ကနေ စတင်ဖြေရှင်းလေ့ ရှိပါတယ်။ ဒီအကြောင်းကို အခန်း ၂၁ (Intervals) မှာ အသေးစိတ် ဆက်လေ့လာပါမယ်။

အခန်း ၁၂ - Heap / Priority Queue

အရင်အခန်းတွေမှာ Queue (အခန်း ၇) ဆိုတာ "အရင်ဝင်တာ အရင်ထွက် (FIFO)" ဆိုတဲ့ စည်းမျဉ်းနဲ့ အလုပ်လုပ်တာ တွေ့ခဲ့ပါတယ်။ ဒါပေမယ့် real-world မှာ "အရင်ဝင်တာ အရင်ထွက်" မဟုတ်ဘဲ **"အရေးကြီးဆုံးက အရင်ထွက်"** ဖြစ်ရတဲ့ အခြေအနေတွေ အများကြီး ရှိပါတယ် —

- ဆေးရုံ အရေးပေါ်ခန်း (ER) မှာ အရင်ရောက်တဲ့ လူနာ မဟုတ်ဘဲ **အခြေအနေ အဆိုးဆုံး** လူနာကို အရင်ကုသည်။
- Operating system က job အများကြီးထဲက **priority အမြင့်ဆုံး** job ကို အရင် run သည်။
- Map app က လမ်းကြောင်းရှာတဲ့အခါ **အနီးဆုံး/ကုန်ကျစရိတ် အနည်းဆုံး** အရင် စစ်သည်။

ဒီလို "အရေးကြီးဆုံးကို အမြန် ထုတ်ယူ" ဖို့ အတွက် အကောင်းဆုံး data structure က **Heap** ဖြစ်ပြီး၊ သူ့ကို အသုံးပြုထားတဲ့ abstract concept ကို **Priority Queue** လို့ ခေါ်ပါတယ်။

ဒီအခန်းမှာ Heap ဆိုတာ ဘာလဲ၊ ဘာကြောင့် priority operation တွေမှာ မြန်သလဲ၊ ဘယ်အချိန်မှာ sorted array အစား Heap သုံးသင့်လဲ ဆိုတာတွေကို လေ့လာသွားပါမယ်။ ပြီးတော့ Heap ကို အသုံးချတဲ့ classic ပြဿနာ ၅ ခု — top K, kth largest, merge k lists, task scheduler, median stream — ကို တစ်ဆင့်ချင်း ဖြေရှင်းကြည့်ပါမယ်။

Priority Queue ဆိုတာ ဘာလဲ

Priority Queue ဆိုတာ element တစ်ခုစီမှာ **priority (ဦးစားပေး အဆင့်)** ပါပြီး၊ ထုတ်ယူတဲ့အခါ **priority အမြင့်ဆုံး (သို့) အနိမ့်ဆုံး** element ကို အရင် ထုတ်ပေးတဲ့ queue တစ်မျိုး ဖြစ်ပါတယ်။ သာမန် Queue နဲ့ ကွာတာက —

	ထုတ်ယူပုံ
သာမန် Queue (FIFO)	အရင် ဝင်တာ အရင် ထွက် (ဝင်တဲ့ အစီအစဉ်အတိုင်း)
Priority Queue	priority အမြင့်ဆုံး/အနိမ့်ဆုံး အရင် ထွက် (ဝင်တဲ့ အစီအစဉ် မဟုတ်)

Priority Queue က **interface (concept)** ဖြစ်ပြီး၊ သူ့ကို အကောင်အထည်ဖော် (implement) ဖို့ နည်းလမ်း အများကြီး ရှိပါတယ်။ အကြမ်းဖျင်း နည်း ၃ မျိုး တွေးကြည့်ရအောင် —

Implementation	Insert	Extract Min/Max
Sorted Array (အမြဲ စီထား)	$O(n)$	$O(1)$
Unsorted Array	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$

Sorted array သုံးရင် ထုတ်ယူတာ မြန်ပေမယ့် (အမြဲ စီထားလို့)၊ element အသစ် ထည့်တိုင်း မှန်ကန်တဲ့ နေရာ ရှာ ထိုးထည့်ရလို့ $O(n)$ ကုန်ပါတယ်။ Heap က **insert နဲ့ extract ၂ ခုလုံးကို** $O(\log n)$ နဲ့ မျှတအောင် လုပ်ပေးတဲ့အတွက် insert/extract နှစ်ခုလုံး မကြာခဏ လုပ်ရတဲ့ workload အတွက် အကောင်းဆုံး ဖြစ်ပါတယ်။

> **မှတ်ချက်:** Sorted Array မှာ priority အမြင့်ဆုံး/အနိမ့်ဆုံး item ကို array ရဲ့ **နောက်ဆုံးဘက်** မှာ ထားနိုင်ရင် extract က $O(1)$ ဖြစ်ပါတယ်။ ရှေ့ဘက်ကနေ remove လုပ်ရတဲ့ implementation မျိုးမှာတော့ ကျန် element တွေ ရွှေ့ ရလို့ $O(n)$ ဖြစ်နိုင်ပါတယ်။

Heap structure ကို နားလည်ခြင်း

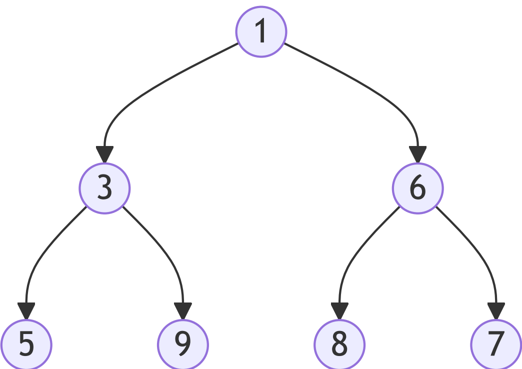
Heap ဟာ စည်းမျဉ်း ၂ ခု ရှိတဲ့ binary tree (သားသမီး အများဆုံး ၂ ခုစီ ရှိတဲ့ tree) ဖြစ်ပါတယ် —

1. **Heap Property (ဦးစားပေး စည်းမျဉ်း):**
    - **Min-Heap** — မိဘ node က သားသမီး node ထက် **ငယ် (သို့) တူ** ရမည်။ ဒါကြောင့် **အငယ်ဆုံး element က အမြဲ ထိပ် (root) မှာ** ရှိသည်။
    - **Max-Heap** — မိဘ node က သားသမီး node ထက် **ကြီး (သို့) တူ** ရမည်။ ဒါကြောင့် **အကြီးဆုံး element က အမြဲ ထိပ်မှာ** ရှိသည်။

1. **Complete Binary Tree (ပြည့်စုံ binary tree):** အလွှာတိုင်းကို ဘယ်ကနေ ညာ အပြည့် ဖြည့်ထားရသည် (နောက်ဆုံး အလွှာသာ မပြည့်လည်း ရ၊ ဒါပေမယ့် ဘယ်ဘက်ကနေ စဖြည့်ရသည်)။

> **သတိ:** Heap ဟာ Binary Search Tree (အခန်း ၁၄) **မဟုတ်ပါ**။ BST မှာ "ဘယ်ငယ်၊ ညာကြီး" ဆိုပြီး node အားလုံး အစီအစဉ်တကျ ရှိပေမယ့်၊ Heap မှာ မိဘ-သားသမီး ဆက်ဆံရေးကိုသာ ထိန်းသည်။ ဘေးချင်းကပ် (sibling) node တွေကြားမှာ အစီအစဉ် မရှိပါ။ ဒါကြောင့် Heap က "အငယ်ဆုံး/အကြီးဆုံး" ကိုသာ မြန်မြန် ပြောနိုင်ပြီး၊ "တန်ဖိုး `x` ရှိလား" ရှာဖို့တော့ မကောင်းပါ။

ဒီ Min-Heap ဥပမာကို ကြည့်ရအောင် — ထိပ်မှာ အငယ်ဆုံး `1` ရှိပြီး၊ မိဘတိုင်းက သားသမီးထက် ငယ်ပါတယ် —

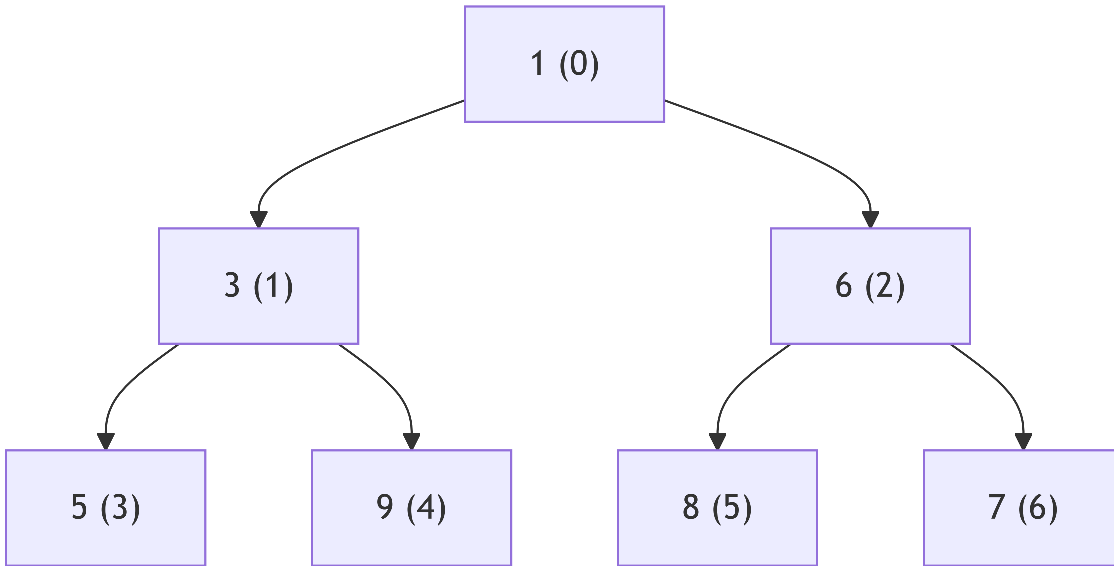


`1` = root (အငယ်ဆုံး)။ `1 < (3,6)` ၊ `3 < (5,9)` ၊ `6 < (8,7)` → Min-Heap rule ပြည့်။

Array အဖြစ် သိမ်းခြင်း

Heap ရဲ့ အရေးကြီးဆုံး အချက်က — **tree ပုံစံ ဖြစ်ပေမယ့် node, pointer တွေ မလိုဘဲ array သက်သက်** နဲ့ သိမ်းနိုင်တာ ဖြစ်ပါတယ်။ Linked list (အခန်း ၈) နဲ့ tree တွေက node တစ်ခုစီမှာ "နောက် node ဘယ်မှာလဲ" ဆိုတဲ့ pointer (reference) သိမ်းရပါတယ်။ Heap မှာတော့ complete binary tree ဖြစ်လို့ **နေရာ မလွတ်** ဘဲ ဖြည့်ထားတဲ့အတွက်၊ node တစ်ခုစီကို array index တစ်ခုနဲ့ တစ်ပြိုင်နက် ချိတ်ဆက်နိုင်ပြီး pointer လုံးဝ မလိုတော့ပါ။

အပေါ်က tree ကို အလွှာလိုက် (level order — ဘယ်ကနေ ညာ) ဖတ်ပြီး array ထဲ ထည့်ကြည့်ရအောင် —



node label = `တန်ဖိုး (index)` ။ array အဖြစ် ကြည့်ရင်:

```
index:   0   1   2   3   4   5   6
value: [ 1,  3,  6,  5,  9,  8,  7 ]
```

Index `i` ရှိ node တစ်ခုရဲ့ မိဘ နဲ့ သားသမီး တွေကို formula ၃ ခုနဲ့ တွက်ထုတ်နိုင်ပါတယ် —

```
parent(i) = (i - 1) / 2     ← Parent (integer division — အကြွင်း ဖြုတ်)
left(i)   = 2 * i + 1       ← left child
right(i)  = 2 * i + 2       ← right child
```

> **သတိ:** `(i - 1) / 2` ဟာ **integer division** (Java မှာ `Int` အချင်းချင်း စားရင် အကြွင်း အလိုအလျောက် ဖြုတ်) ဖြစ်ပါတယ်။ ဥပမာ — index `2` ရဲ့ မိဘက `(2 - 1) / 2 = 0` (`0.5` မဟုတ်)၊ index `1` ရဲ့ မိဘကလည်း `(1 - 1) / 2 = 0` ဖြစ်ပါတယ် — ဒါကြောင့် node `1` , `2` ၂ ခုလုံးရဲ့ မိဘက root ဖြစ်တာ မှန်ကန်ပါတယ်။

ဒီ formula တွေ ဘယ်လို အလုပ်လုပ်လဲ ဥပမာနဲ့ စစ်ကြည့်ရအောင် —

node (index)	တန်ဖိုး	left = `2i+1`	right = `2i+2`	parent = `(i-1)/2`

0	1	index 1 → 3	index 2 → 6	(root, မရှိ)
1	3	index 3 → 5	index 4 → 9	index 0 → 1
2	6	index 5 → 8	index 6 → 7	index 0 → 1

ဥပမာ — index `2` (`6`) ရဲ့ left child က `2×2+1 = 5` (`8`)၊ right child က `2×2+2 = 6` (`7`)၊ မိဘက `(2-1)/2 = 0` (`1`) ဖြစ်ပါတယ်။ tree ပုံနဲ့ တိုက်ကြည့်ရင် တိတိကျကျ ကိုက်ညီပါတယ်။ ဒီ index တွက်နည်းတွေက Heap Sort (အခန်း ၁၁) မှာ တွေ့ခဲ့တဲ့ `heapify` formula တွေ အတိအကျ ဖြစ်ပါတယ်။

> **ဘာကြောင့် ဒီ formula တွေ မှန်သလဲ?** Complete binary tree မှာ node တွေကို အလွှာလိုက် ဆက်တိုက် ထည့်ထားလို့၊ index `i` ရဲ့ ရှေ့မှာ node `i` ခု ရှိပါတယ်။ Binary tree ဖြစ်လို့ node တစ်ခုစီမှာ child ၂ ခုစီ ရှိတဲ့အတွက်၊ index တွေဟာ ၂ ဆ ပုံစံ (`2i+1` , `2i+2`) နဲ့ ဖြန့်သွားတာ ဖြစ်ပါတယ်။ ဒါကြောင့် node, pointer မလိုဘဲ သင်္ချာ တွက်ချက်မှု သက်သက်နဲ့ tree တစ်ခုလုံးကို ဖြတ်သန်းနိုင်တာ ဖြစ်ပါတယ်။

Heap ၏ အဓိက လုပ်ဆောင်ချက်များ

Heap မှာ အဓိက operation ၃ ခု ရှိပါတယ် — **Peek**, **Insert**, **Extract**။ Min-Heap ကို ဥပမာ ထားပြီး ကြည့်ရအောင်။

Peek — ထိပ်က element ကို ကြည့်ခြင်း ($O(1)$)

Heap ရဲ့ ထိပ် (root, index `0`) မှာ အငယ်ဆုံး (Min-Heap) / အကြီးဆုံး (Max-Heap) က အမြဲ ရှိနေလို့၊ ဖတ်ဖို့ $O(1)$ သာ ကြာပါတယ်။ ဒါက Heap ရဲ့ အဓိက အားသာချက် ဖြစ်ပါတယ်။

Insert — element ထည့်ခြင်း ($O(\log n)$)

element အသစ်ကို ဘယ်နေရာ ထည့်ရင် —

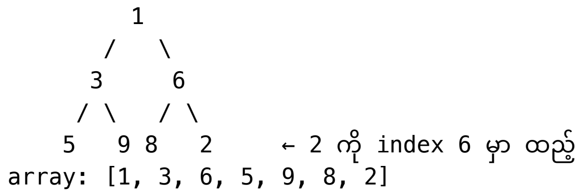
- **Heap property** (Parent <= child) နဲ့
- **Complete tree shape** (နေရာ မလွတ်) —

၂ ခုလုံး မပျက်ဘဲ ဖြစ်မလဲ ဆိုတာ စဉ်းစားရပါတယ်။ ဖြေရှင်းချက်က ၂ ဆင့် —

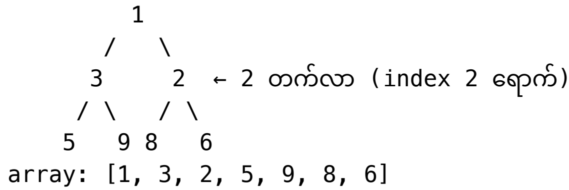
1. **နေရာချ:** element အသစ်ကို **array ရဲ့ နောက်ဆုံး** (tree ရဲ့ နောက်ဆုံး အလွှာ၊ ဘယ်ဘက် နေရာလွတ်) မှာ အရင် ထည့်သည်။ ဒါက tree shape ကို မပျက်စေပါ။
2. **sift up (အပေါ်တွန်းတင်):** ဒါပေမယ့် အခု heap rule ပျက်နေနိုင်တယ် (element အသစ်က သူ့ parent ထက် ငယ်နေနိုင်တယ်)။ ဒါကြောင့် **parent ထက် ငယ်နေသ၍ parent နဲ့ နေရာချင်း လဲတက်** သွားသည် (bubble up)။ Parent ထက် မငယ်တော့ရင် (သို့) root ရောက်ရင် ရပ်သည်။

`[1, 3, 6, 5, 9, 8]` Min-Heap ထဲ `2` ထည့်ကြည့်ရအောင် —

```
Step 1 — နောက်ဆုံး (index 6) မှာ ထည့်:
```




```
Step 2 — sift up: 2 ရဲ့ parent = index (6−1)/2 = 2 (တန်ဖိုး 6)
      2 < 6 → လဲ
```




```
Step 3 — 2 ရဲ့ parent = index (2−1)/2 = 0 (တန်ဖိုး 1)
      2 < 1? မဟုတ် → ရပ်
   ရလဒ်: [1, 3, 2, 5, 9, 8, 6]
```

element အသစ်က အများဆုံး root အထိ တက်နိုင်ပြီး၊ tree ရဲ့ အမြင့်က $\log n$ သာ ဖြစ်လို့ အဆိုးဆုံး $\log n$ ကြိမ်သာ လဲတက်ရပါတယ်။ ဒါကြောင့် $O(\log n)$ ဖြစ်ပါတယ်။

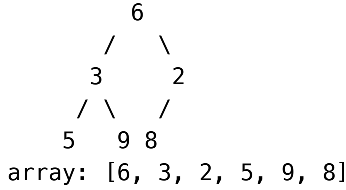
```
// Min-Heap ကို array (heap) နဲ့ size (count) ဖြင့် ကိုယ်တိုင် ရေးပြထားသည်
void insert(int[] heap, int[] size, int value) {
    int i = size[0];        // နောက်ဆုံး နေရာလွတ်
    heap[i] = value;        // Step 1: နောက်ဆုံးမှာ ထည့်
    size[0]++;

    // Step 2: sift up — မိဘထက် ငယ်နေသ၍ လဲတက်
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (heap[i] < heap[parent]) {
            int temp = heap[i]; heap[i] = heap[parent]; heap[parent] = temp;
            i = parent;     // တက်သွားတဲ့ နေရာကနေ ဆက်စစ်
        } else {
            break;          // မိဘထက် မငယ်တော့ → ရပ်
        }
    }
}
```

Extract — ထိပ်က element ထုတ်ခြင်း ($O(\log n)$)

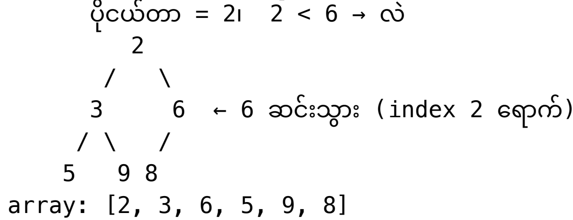
ထိပ် (root) ကို ထုတ်လိုက်ရင် root နေရာ ဟာသွားပါတယ်။ ဒါကို ဘယ်လို ပြန်ဖြည့်မလဲ? ထပ်ပြီး ၂ ဆင့် —

1. **နောက်ဆုံးကို ထိပ်တင်: array ရဲ့ နောက်ဆုံး element ကို root နေရာ** ရွှေ့တင်ပြီး၊ array အရွယ် ၁ လျှော့သည်။ ဒါက tree shape ကို မပျက်စေပါ (နောက်ဆုံး အလွှာက လျော့သွားရုံ)။
2. **sift down (အောက်ဆင်းချ):** အခု root က heap rule ပျက်နေနိုင်တယ် (root က child ထက် ကြီးနေနိုင်တယ်)။ ဒါကြောင့် **child ၂ ခုထဲက ပိုငယ်တာထက် ကြီးနေသ၍ အဲ့ ပိုငယ်တဲ့ child နဲ့ လဲဆင်း** ရသည်။ (Min-Heap မှာ ပိုငယ်တဲ့ child နဲ့ လဲမှ rule မပျက်ပါ။) children ထက် မကြီးတော့ရင် (သို့) leaf ရောက်ရင် ရပ်သည်။

```
Min-Heap [1, 3, 2, 5, 9, 8, 6] မှ root (1) ထုတ်:

Step 1 — root ထုတ်၊ နောက်ဆုံး (6) ကို ထိပ်တင်:
```



```
          6
        /   \
       3     2
      / \   /
     5   9 8
   array: [6, 3, 2, 5, 9, 8]

Step 2 — sift down: 6 ရဲ့ child = index 1 (3), index 2 (2)
```



```
         ပိုငယ်တာ = 2၊  2 < 6 → လဲ
          2
        /   \
       3     6  ← 6 ဆင်းသွား (index 2 ရောက်)
      / \   /
     5   9 8
   array: [2, 3, 6, 5, 9, 8]

Step 3 — 6 ရဲ့ child = index 5 (8) တစ်ခုသာ
         8 < 6? မဟုတ် → ရပ်
   ထုတ်လိုက်တာ: 1၊  ကျန် heap: [2, 3, 6, 5, 9, 8]
```

ဒီနေရာမှာလည်း root က အများဆုံး leaf အထိ ဆင်းနိုင်ပြီး tree အမြင့် $\log n$ သာ ဖြစ်လို့ $O(\log n)$ ဖြစ်ပါတယ်။

```c
// Min-Heap မှ အငယ်ဆုံး (root) ကို ထုတ်ပြန်သည်
int extractMin(int[] heap, int[] size) {
    int min = heap[0];              // ထိပ် = အဖြေ
    size[0]--;
    heap[0] = heap[size[0]];        // Step 1: နောက်ဆုံးကို ထိပ်တင်

    // Step 2: sift down — child ထဲ ပိုငယ်တာထက် ကြီးနေသ၍ လဲ ဆင်း
    int i = 0;
    while (true) {
        int left = 2 * i + 1, right = 2 * i + 2, smallest = i;
        if (left < size[0] && heap[left] < heap[smallest]) smallest = left;
        if (right < size[0] && heap[right] < heap[smallest]) smallest = right;
        if (smallest == i) break;    // child ထက် မကြီးတော့ → ရပ်
        int temp = heap[i]; heap[i] = heap[smallest]; heap[smallest] = temp;
        i = smallest;               // ဆင်းသွားတဲ့ နေရာကနေ ဆက် စစ်
    }
    return min;
}
```

> **sift up နဲ့ sift down ကွာခြားချက်:** sift **up** က element တစ်ခုကို **parent တစ်ခု**နဲ့သာ နှိုင်းယှဉ်ရလို့ ရိုးရှင်းသည်။ sift **down** က **သားသမီး ၂ ခု**နဲ့ နှိုင်းယှဉ်ပြီး ပိုငယ်တာ (Min-Heap) / ပိုကြီးတာ (Max-Heap) ကို ရွေး လဲရတဲ့အတွက် နည်းနည်း ပိုရှုပ်ပါတယ်။ ဒီ ၂ ခုက Heap operation အားလုံးရဲ့ အခြေခံ ဖြစ်ပါတယ်။ (shift up , shift down မဟုတ်ပါ။ Sift up , Sift down ဖြစ်ပါသည်။

> **အကျဉ်းချုပ်:** Heap ဟာ **Peek** $O(1)$, **Insert** $O(\log n)$, **Extract** $O(\log n)$ ဖြစ်ပါတယ်။ ဒါက "အကြီးဆုံး/အငယ်ဆုံးကို မကြာခဏ ထုတ်၊ element အသစ် မကြာခဏ ထည့်" ဆိုတဲ့ workload အတွက် sorted array (insert $O(n)$) ထက် များစွာ ပိုကောင်းပါတယ်။

Build Heap — array တစ်ခုလုံးကို Heap ပြောင်းခြင်း ($O(n)$)

element တွေ အကုန် တစ်ပြိုင်နက် ရှိနေပြီးသား array တစ်ခုကို Heap ဖြစ်အောင် တည်ဆောက်ချင်ရင် — တစ်ခုချင်း `insert` ခေါ်ရင် $O(n \log n)$ ကုန်ပါတယ်။ ဒါပေမယ့် ပိုကောင်းတဲ့ နည်း ရှိပါတယ် — **နောက်ဆုံး Parent node ကနေ စပြီး root အထိ၊ node တစ်ခုစီကို sift down** လုပ်တာ ဖြစ်ပါတယ်။

```
void buildHeap(int[] arr) {
    int n = arr.length;
    // နောက်ဆုံး parent node = (n/2 - 1) ကနေ root (0) အထိ
    for (int i = n / 2 - 1; i >= 0; i--) {
        siftDown(arr, n, i);   // node i ကို အောက်ကို ချိန်ညှိ
    }
}
```

> **ဘာကြောင့် $O(n)$ ဖြစ်သလဲ ($O(n \log n)$ မဟုတ်)?** node အများစုက tree ရဲ့ **အောက်ခြေနားမှာ** ရှိလို့ sift down က ၀ (သို့) ၁ အလွှာသာ ဆင်းရပါတယ် (leaf တွေက လုံးဝ မဆင်းရ)။ အပေါ်ပိုင်းက node **နည်းနည်း**သာ အလွှာများစွာ ဆင်းနိုင်ပါတယ်။ ဒါကြောင့် စုစုပေါင်း အလုပ်က $O(n \log n)$ မဟုတ်ဘဲ $O(n)$ ပဲ ဖြစ်ပါတယ်။ ဒါက Heap Sort (အခန်း ၁၁) ရဲ့ ပထမ အဆင့် (array → max-heap) ဖြစ်ပြီး၊ တစ်ခုချင်း insert လုပ်တာထက် ပိုမြန်ပါတယ်။

Heap vs Sorted Array — ဘယ်အချိန် ဘာသုံးမလဲ

Heap က **အမြဲ အပြည့် စီထားတာ မဟုတ်ဘဲ**၊ "အငယ်ဆုံး/အကြီးဆုံးက ထိပ်မှာ ရှိဖို့" လောက်ပဲ ထိန်းတာ ဖြစ်ပါတယ်။ ဒါက "ဘာကြောင့် insert မြန်သလဲ" ဆိုတဲ့ မေးခွန်းရဲ့ အဖြေ ဖြစ်ပါတယ် — အပြည့် မစီရတဲ့အတွက် အလုပ် နည်းပါတယ်။

အခြေအနေ	ရွေးချယ်မှု
Data အကုန် **အစီအစဉ်တကျ** လိုသည်	**Sort** (Sorted Array)
အကြီး/အငယ်ဆုံးကိုသာ မကြာခဏ ထုတ်/ထည့်သည်	**Heap**
Data **တစ်ခါတည်း** ပေးပြီး နောက် မပြောင်းတော့	**Sort** လုပ်ထားရင် ရ
Data **တဖြည်းဖြည်း ဝင်လာ (stream)** ပြီး အမြန် ထုတ်ရသည်	**Heap**

ဥပမာ — element ၁ သန်းထဲက **အကြီးဆုံး ၁၀ ခု** ပဲ လိုချင်ရင်၊ အကုန် sort လုပ်ရင် $O(n \log n)$ ကုန်ပါတယ်။ ဒါပေမယ့် အရွယ် ၁၀ ရှိတဲ့ Heap သုံးရင် $O(n \log 10) \approx O(n)$ နဲ့ ရပါတယ် (နောက် "Top K" ပြဿနာမှာ ပြပါမယ်)။

> **Heap Sort နဲ့ ဆက်နွှယ်မှု:** အခန်း ၁၁ မှာ တွေ့ခဲ့တဲ့ **Heap Sort** ဟာ ဒီ Heap structure ကိုပဲ အသုံးချတာ ဖြစ်ပါတယ် — element အားလုံးကို Heap ထဲ ထည့်ပြီး၊ ထိပ်ကနေ တစ်ခုချင်း ထုတ်ရင် စီပြီးသား အစီအစဉ် ရပါတယ်။ element `n` ခုကို extract လုပ်ရင် တစ်ခုစီ $O(\log n)$ ဖြစ်လို့ စုစုပေါင်း $O(n \log n)$ ဖြစ်ပါတယ်။ ဒါကြောင့် Heap ကို နားလည်ရင် Heap Sort ကိုပါ နားလည်ပြီး ဖြစ်ပါတယ်။

Programming Language ထဲက Priority Queue

လက်တွေ့မှာ Heap ကို ကိုယ်တိုင် ရေးစရာ မလိုပါ — language အများစုမှာ built-in ပါပြီးသား ဖြစ်ပါတယ်။ Java မှာ `PriorityQueue` က **default အားဖြင့် Min-Heap** ဖြစ်ပါတယ်။

```java
import java.util.PriorityQueue;
import java.util.Collections;

class HeapDemo {
    void demo() {
        // Min-Heap (default): အငယ်ဆုံး အရင်ထွက်
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        minHeap.offer(5);
        minHeap.offer(1);
        minHeap.offer(3);
        minHeap.peek();  // 1 (အငယ်ဆုံး၊ မထုတ်)
        minHeap.poll();  // 1 ထုတ် → ကျန် [3, 5]

        // Max-Heap: comparator ပြောင်းရန်
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.offer(5);
        maxHeap.offer(1);
        maxHeap.offer(3);
        maxHeap.peek();  // 5 (အကြီးဆုံး)
    }
}
```

> **မှတ်ချက်:** Java `PriorityQueue` က default Min-Heap ဖြစ်တာ မှတ်ထားပါ။ Max-Heap လိုရင် `Collections.reverseOrder()` (သို့) custom Comparator ထည့်ရပါတယ်။ Object တွေ အတွက်လည်း Comparator နဲ့ "ဘယ် field အလိုက် priority ပေးမလဲ" သတ်မှတ်နိုင်ပါတယ် (အခန်း ၁၁ က Custom Comparator အတိုင်း)။

Real-world Examples

Heap / Priority Queue ဟာ "အရေးကြီးဆုံးကို အရင်" ဆိုတဲ့ pattern ရှိတဲ့ system တိုင်းမှာ တွေ့ရပါတယ် —

- **Job Queue** — background job တွေကို priority အလိုက် run သည် (premium user ရဲ့ job ကို အရင်)။
- **Retry Queue** — fail သွားတဲ့ request တွေကို "နောက် retry လုပ်ရမယ့် အချိန် (timestamp)" အလိုက် Min-Heap ထဲ ထား၊ အချိန်ရောက်ဆုံးကို အရင် ထုတ် retry လုပ်သည်။
- **Scheduler / Event Loop** — OS scheduler နဲ့ timer system တွေက "နောက်ဆုံး အလုပ်လုပ်ရမယ့်အချိန်" အနီးဆုံး event ကို Min-Heap နဲ့ အရင် ထုတ်သည်။
- **Notification Priority** — notification တွေကို importance အလိုက် ပို့သည် (security alert ကို promotion ထက် အရင်)။
- **Top Selling Products** — sales stream ထဲကနေ "အရောင်းရဆုံး ၁၀ မျိုး" ကို Heap နဲ့ အမြဲ track လုပ်သည် (အကုန် sort မလုပ်ဘဲ)။
- **Dijkstra's Algorithm** (အခန်း ၁၉) — shortest path ရှာတဲ့အခါ "ကုန်ကျစရိတ် အနည်းဆုံး node" ကို Min-Heap နဲ့ အရင် ရွေးသည်။

ဒီ pattern တွေ အကုန်လုံးက တူညီပါတယ် — **"item အများကြီးထဲက အရေးကြီးဆုံးကို မကြာခဏ ထုတ်၊ item အသစ်တွေ မကြာခဏ ဝင်လာ"** ဆိုတဲ့ အခြေအနေပါ။ ဒီအချိန်မှာ Heap က အကောင်းဆုံး ဖြစ်ပါတယ်။

Questions

Heap / Priority Queue ကို လက်တွေ့ပြဿနာ ၅ ခုနဲ့ ချိတ်ဆက် လေ့လာကြည့်ရအောင်။ မေးခွန်းတိုင်းရဲ့ သော့ချက်က — **"ဘယ်အချိန်မှာ Min-Heap သုံးမလဲ၊ ဘယ်အချိန်မှာ Max-Heap သုံးမလဲ"** ဆိုတာ မှန်ကန်စွာ ရွေးတတ်ဖို့ ဖြစ်ပါတယ်။

အောက်က ပြဿနာတွေမှာ Heap ကို သုံးရတဲ့ အကြောင်းရင်းက မတူပါဘူး။ တချို့က "အကြီးဆုံး `k` ခု ထိန်းထားဖို့" သုံးတာ၊ တချို့က "အချိန်တိုင်း အကြီးဆုံး/အငယ်ဆုံး ထုတ်ဖို့" သုံးတာ၊ တချို့က "အလယ်တန်ဖိုးကို ထိန်းဖို့" သုံးတာ ဖြစ်ပါတယ်။

၁။ Kth Largest Element in an Array

ကိန်းပြည့် array `nums` နဲ့ ကိန်း `k` ပေးထားသည်။ **`k` ခုမြောက် အကြီးဆုံး (kth largest)** element ကို ပြန်ပါ (စီပြီးသား အစီအစဉ်မှာ `k` နေရာမြောက်ကို ဆိုလိုသည်၊ distinct မဟုတ်)။

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
Explanation: စီရင် [6,5,4,3,2,1] → ၂ ခုမြောက် အကြီးဆုံးက 5။
```

ရှင်းလင်းချက်

အလွယ်ဆုံး နည်းက အကုန် sort လုပ်ပြီး `k` နေရာမြောက် ယူတာ ဖြစ်ပေမယ့် $O(n \log n)$ ကုန်ပါတယ်။ ပိုကောင်းတဲ့ နည်းက **အရွယ် `k` ရှိတဲ့ Min-Heap** သုံးတာ ဖြစ်ပါတယ်။

အဓိက idea — **"အကြီးဆုံး `k` ခု"** ကို heap ထဲ ထားရင်၊ အဲ့ `k` ခုထဲက **အငယ်ဆုံး (heap ထိပ်)** က ကျွန်တော်တို့ လိုချင်တဲ့ kth largest ဖြစ်ပါတယ်။

- Min-Heap ထဲ element ထည့်သွားသည်။ heap အရွယ် `k` ထက် ကျော်ရင် **ထိပ် (အငယ်ဆုံး) ကို ထုတ်** ပစ်သည်။
- ဒါဆို heap ထဲမှာ "အကြီးဆုံး `k` ခု" သာ အမြဲ ကျန်ပြီး၊ ထိပ်က kth largest ဖြစ်သည်။

> **ဘာကြောင့် Max-Heap မဟုတ်ဘဲ Min-Heap သုံးသလဲ?** "အကြီးဆုံး ၂ ခု" ထဲက မလိုတာ (ပိုသေးတာ) ကို ဖြုတ်ပစ်ဖို့၊ ထိပ်မှာ "အငယ်ဆုံး" ရှိတဲ့ Min-Heap က အဆင်ပြေပါတယ်။ Max-Heap သုံးရင် ထိပ်က အကြီးဆုံး ဖြစ်နေလို့ မလိုတာ ဖြုတ်ရ ခက်ပါတယ်။

> **Time Complexity:** $O(n \log k)$ - element `n` ခုစီကို $O(\log k)$ heap operation လုပ်သောကြောင့်။ `k` သေးရင် $O(n \log n)$ ထက် မြန်သည်။
> **Space Complexity:** $O(k)$ - heap မှာ element `k` ခုသာ ထား။

Java Solution

```java
import java.util.PriorityQueue;

class Solution {
    public int findKthLargest(int[] nums, int k) {
        // Min-Heap: အကြီးဆုံး k ခုကိုသာ ထားသည်
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.offer(num);
            // အရွယ် k ကျော်ရင် အငယ်ဆုံး (ထိပ်) ကို ဖြုတ်
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }
        // ကျန်တဲ့ k ခုထဲက အငယ်ဆုံး = kth largest
        return minHeap.peek();
    }
}
```

၂။ Top K Frequent Elements

ကိန်းပြည့် array `nums` နဲ့ ကိန်း `k` ပေးထားသည်။ **Frequency အများဆုံး** ဖြစ်တဲ့ element `k` ခုကို ပြန်ပါ။

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1, 2]
```

```
Explanation: 1 က ၃ ကြိမ်၊ 2 က ၂ ကြိမ်၊ 3 က ၁ ကြိမ်။ အများဆုံး ၂ ခုက [1, 2]။
```

ရှင်းလင်းချက်

ဒီပြဿနာကို အခန်း ၁၁ မှာ **Bucket Sort** idea နဲ့ $O(n)$ ဖြေခဲ့ပါတယ်။ ဒီနေရာမှာ **Heap** နဲ့ ဖြေပုံကို ကြည့်ရအောင် ($O(n \log k)$) — Top K pattern ကို လေ့ကျင့်ဖို့ ဖြစ်ပါတယ်။

- **Step 1:** Hash Map (အခန်း ၄) နဲ့ element တစ်ခုစီရဲ့ frequency ရေတွက်သည် — `{1:3, 2:2, 3:1}` ။
- **Step 2:** frequency အလိုက် **အရွယ်** `k` **Min-Heap** ထဲ ထည့်သည်။ heap က `k` ကျော်ရင် frequency အနည်းဆုံးကို ဖြုတ်ပစ်သည်။
- **Step 3:** heap ထဲ ကျန်တဲ့ `k` ခုက "frequency အများဆုံး `k` ခု" ဖြစ်သည်။

ဒါက အရင်ပြဿနာ (Kth Largest) နဲ့ **pattern တူ**ပါတယ် — "frequency အကြီးဆုံး `k` ခု" ထားဖို့ Min-Heap သုံးတာ ဖြစ်ပါတယ်။

> **မှတ်ချက်:** အဖြေ array ရဲ့ အစီအစဉ် မလိုပါ။ `[1, 2]` နဲ့ `[2, 1]` နှစ်ခုလုံး မှန်ပါတယ် (heap ကနေ ထုတ်တဲ့ အစီအစဉ်ပေါ် မူတည်)။

> **Time Complexity:** $O(n \log k)$ - distinct element တစ်ခုစီကို $O(\log k)$ heap operation။
> **Space Complexity:** $O(n + k)$ - map ($O(n)$) နှင့် heap ($O(k)$)။

Java Solution

```java
import java.util.*;

class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // Step 1: frequency ရေတွက်
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) {
            count.put(num, count.getOrDefault(num, 0) + 1);
        }

        // Step 2: frequency အလိုက် Min-Heap (frequency အနည်းဆုံး ထိပ်မှာ)
        // count.get(a) - count.get(b) သုံးရင် overflow ဖြစ်နိုင်လို့ Integer.compare သုံး
        PriorityQueue<Integer> heap = new PriorityQueue<>(
            (a, b) -> Integer.compare(count.get(a), count.get(b))
        );
        for (int num : count.keySet()) {
            heap.offer(num);
            if (heap.size() > k) {
                heap.poll();   // frequency အနည်းဆုံး ဖြုတ်
            }
        }

        // Step 3: heap ထဲ ကျန်တာ k ခုက အဖြေ
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = heap.poll();
```

```
        }
        return result;
    }
}
```

၃။ Merge k Sorted Lists

စီပြီးသား linked list `k` ခု ပါတဲ့ array ပေးထားသည်။ အကုန်လုံးကို **တစ်ခုတည်း စီပြီးသား list** အဖြစ် ပေါင်းပြီး ပြန်ပါ။

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
```

ရှင်းလင်းချက်

List `k` ခုစလုံး စီပြီးသား ဖြစ်လို့ "list တွေရဲ့ ရှေ့ဆုံး node `k` ခုထဲက **အငယ်ဆုံးကို** အရင် ထုတ်" ရင် ရပါတယ်။ ဒီ "အငယ်ဆုံးကို အမြန် ရှာ" ဆိုတဲ့ အလုပ်က Min-Heap ရဲ့ အလုပ်အတိအကျ ဖြစ်ပါတယ်။

- list `k` ခုရဲ့ ပထမ node `k` ခုကို Min-Heap ထဲ ထည့်သည်။
- heap ထိပ်က အငယ်ဆုံး node ကို ထုတ်ပြီး result list မှာ ဆက်သည်။
- ထုတ်လိုက်တဲ့ node ရဲ့ **နောက် (next) node** ကို heap ထဲ ပြန်ထည့်သည်။
- heap ဗလာ ဖြစ်တဲ့အထိ ထပ်လုပ်သည်။

heap ထဲမှာ အချိန်တိုင်း node `k` ခုသာ ရှိလို့ operation တစ်ခုစီ $O(\log k)$ ဖြစ်ပါတယ်။

> **ဘာကြောင့် Heap သုံးသလဲ?** list `k` ခုရဲ့ ရှေ့ဆုံးတွေထဲက အငယ်ဆုံး ရှာဖို့ တစ်ခုချင်း နှိုင်းယှဉ်ရင် $O(k)$ ကုန်ပြီး၊ node `N` ခုအတွက် $O(N \times k)$ ဖြစ်ပါတယ်။ Heap သုံးရင် အငယ်ဆုံး ရှာတာ $O(\log k)$ ဖြစ်လို့ စုစုပေါင်း $O(N \log k)$ — `k` ကြီးရင် များစွာ ပိုမြန်ပါတယ်။

> **Time Complexity:** $O(N \log k)$ - node စုစုပေါင်း `N` ခုစီကို $O(\log k)$ heap operation (k = list အရေအတွက်)။
> **Space Complexity:** $O(k)$ - heap မှာ node `k` ခုသာ။

Java Solution

```java
import java.util.PriorityQueue;
import java.util.Comparator;

// ListNode အဖွဲ့အစည်း (အခန်း ၈ မှ)
class ListNode {
    int val;
    ListNode next;
```

```
        ListNode(int val) { this.val = val; }
    }

    class Solution {
        public ListNode mergeKLists(ListNode[] lists) {
            // Min-Heap: node ၏ val အလိုက် စီ
            // (a.val - b.val သုံးရင် ကိန်းကြီးတွေမှာ overflow ဖြစ်နိုင်လို့ comparingInt သုံး)
            PriorityQueue<ListNode> heap = new PriorityQueue<>(Comparator.comparingInt(a ->
    a.val));

            // list တစ်ခုစီ၏ ပထမ node ထည့်
            for (ListNode node : lists) {
                if (node != null) heap.offer(node);
            }

            ListNode dummy = new ListNode(0); // result ၏ ရှေ့ဆုံး အကူ node
            ListNode tail = dummy;

            while (!heap.isEmpty()) {
                ListNode smallest = heap.poll();   // အငယ်ဆုံး ထုတ်
                tail.next = smallest;              // result မှာ ဆက်
                tail = tail.next;
                // ထုတ်လိုက်တဲ့ node ၏ နောက် node ကို heap ထဲ ပြန်ထည့်
                if (smallest.next != null) {
                    heap.offer(smallest.next);
                }
            }
            return dummy.next;
        }
    }
```

၄။ Task Scheduler

CPU task တွေ (`tasks` , character A–Z) နဲ့ cooling time `n` ပေးထားသည်။ **တူညီတဲ့ task ၂ ခု ကြားမှာ အနည်းဆုံး `n` အလွတ် (interval) ရှိရမည်**။ task အကုန် ပြီးဖို့ လိုအပ်တဲ့ **အနည်းဆုံး interval အရေအတွက်** ကို ပြန်ပါ။

Example 1:

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
Explanation: A → B → idle → A → B → idle → A → B  (interval ၈ ခု)
             တူညီ A ၂ ခုကြားမှာ အနည်းဆုံး ၂ အလွတ် ရှိရသည်။
```

ရှင်းလင်းချက်

အဓိက idea — **frequency အများဆုံး task ကို အရင် လုပ်** သင့်ပါတယ် (မဟုတ်ရင် နောက်ဆုံးမှာ အဲ့ task တွေ စုပြီး idle များစွာ ဖြစ်မယ်)။ "frequency အများဆုံးကို အမြန် ရွေး" ဖို့ **Max-Heap** သုံးပါတယ်။

- frequency ရေတွက်ပြီး Max-Heap ထဲ ထည့်သည်။
- interval `n+1` အရှည် တစ်ခုစီမှာ — heap ထိပ်ကနေ frequency အများဆုံး task တွေကို (အများဆုံး `n+1` မျိုး) ထုတ် run သည်။
- run ပြီးတဲ့ task ရဲ့ frequency ကို ၁ လျှော့ပြီး၊ `0` မဟုတ်သေးရင် heap ထဲ ပြန်ထည့်သည်။

- heap ဗလာ ဖြစ်တဲ့အထိ ထပ်လုပ်ပြီး interval ရေတွက်သည်။

> **ဘာကြောင့် Max-Heap သုံးသလဲ?** ဒီပြဿနာက "ကျန်နေသေးတဲ့ task တွေထဲ frequency **အများဆုံး**ကို အရင် လုပ်" ဖို့ ဖြစ်လို့၊ ထိပ်မှာ အကြီးဆုံး ရှိတဲ့ Max-Heap က သင့်တော်ပါတယ်။ (အရင် ၃ ပြဿနာက "အကြီးဆုံး/အငယ်ဆုံး `k` ခု ထား" ဖို့ Min-Heap သုံးတာ — ဒီမှာတော့ "အကြီးဆုံးကို တိုက်ရိုက် ထုတ်သုံး" ဖို့ Max-Heap ဆိုတဲ့ ကွာခြားချက် သတိပြုပါ။)

> **Time Complexity:** ဒီ simulation နည်းက cycle တစ်ခုစီမှာ idle slot အပါအဝင် `n+1` slot ဖြတ်ရလို့ $O(\text{totalIntervals})$ ဖြစ်ပါတယ်။ task အမျိုးအစား ၂၆ မျိုးသာ ရှိလို့ heap operation က constant နီးပါး ဖြစ်သည်။ (သင်္ချာ formula နဲ့တွက်ရင် $O(N)$ နဲ့လည်း ရပေမယ့်၊ ဒီအခန်းအတွက် Heap simulation နည်းကို ပြထားတာ ဖြစ်ပါတယ်။)
> **Space Complexity:** $O(1)$ - heap မှာ အများဆုံး ၂၆ မျိုးသာ။

Java Solution

```
import java.util.*;

class Solution {
    public int leastInterval(char[] tasks, int n) {
        // Step 1: frequency ရေတွက်
        int[] freq = new int[26];
        for (char t : tasks) freq[t - 'A']++;

        // Step 2: Max-Heap (frequency အများဆုံး ထိပ်)
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        for (int f : freq) {
            if (f > 0) maxHeap.offer(f);
        }

        int intervals = 0;
        while (!maxHeap.isEmpty()) {
            List<Integer> temp = new ArrayList<>();
            int cycle = 0; // ဒီ cycle မှာ တကယ် run တဲ့ task အရေအတွက်
            // interval (n+1) တစ်ခုစီမှာ frequency အများဆုံး task တွေ ထုတ် run
            for (int i = 0; i < n + 1; i++) {
                if (!maxHeap.isEmpty()) {
                    int f = maxHeap.poll();
                    cycle++;
                    if (f - 1 > 0) temp.add(f - 1); // ၁ လျှော့ပြီး ကျန်ရင် မှတ်
                }
            }
            // run ပြီးသား task တွေ heap ထဲ ပြန်ထည့်
            for (int f : temp) maxHeap.offer(f);

            // heap ဗလာ ဖြစ်ရင် (နောက်ဆုံး cycle) တကယ် run တဲ့ အရေအတွက်သာ ပေါင်း (idle မ
ရေတွက်)၊
            // မဖြစ်သေးရင် idle အပါအဝင် (n+1) လုံး ပေါင်း
            intervals += maxHeap.isEmpty() ? cycle : n + 1;
        }
        return intervals;
    }
}
```

၅။ Find Median from Data Stream

ကိန်းတွေ တစ်ခုပြီးတစ်ခု ဝင်လာ (stream) သည်။ အချိန်မရွေး လက်ရှိ ကိန်းအားလုံးရဲ့ **median (အလယ်တန်ဖိုး)** ကို ပြန်ပေးနိုင်တဲ့ data structure တည်ဆောက်ပါ။ (median = စီပြီးသား အလယ်တန်ဖိုး၊ အရေအတွက် စုံ ဖြစ်ရင် အလယ် ၂ ခုရဲ့ ပျမ်းမျှ)။

Example 1:

```
addNum(1)     →  [1]
addNum(2)     →  [1,2]
findMedian()  →  1.5    (1 နဲ့ 2 ၏ ပျမ်းမျှ)
addNum(3)     →  [1,2,3]
findMedian()  →  2      (အလယ်)
```

ရှင်းလင်းချက်

ကိန်းဝင်လာတိုင်း အကုန် sort လုပ်ရင် $O(n \log n)$ ကုန်ပါတယ်။ ပိုကောင်းတဲ့ နည်းက **Heap ၂ ခု** သုံးတာ ဖြစ်ပါတယ် — ဒါက Heap ရဲ့ classic technique တစ်ခု ဖြစ်ပါတယ်။

idea — ကိန်းတွေကို အလယ်မှာ **၂ ပိုင်း ခွဲ** ထားသည် —

- **Max-Heap (`small`)** — အငယ်ပိုင်း (ဘယ်ခြမ်း) ကို သိမ်းသည်။ ထိပ်မှာ "ဘယ်ခြမ်းရဲ့ အကြီးဆုံး" ရှိသည်။
- **Min-Heap (`large`)** — အကြီးပိုင်း (ညာခြမ်း) ကို သိမ်းသည်။ ထိပ်မှာ "ညာခြမ်းရဲ့ အငယ်ဆုံး" ရှိသည်။

ဒီ ၂ ခုကို **balance** (အရေအတွက် ကွာတာ ၁ ထက် မပိုအောင်) ထိန်းထားရင် —

- median က heap ၂ ခုရဲ့ **ထိပ်** မှာ ရှိနေပါတယ်။ အရေအတွက် မညီရင် ပိုများတဲ့ heap ရဲ့ ထိပ်က median၊ ညီရင် ထိပ် ၂ ခုရဲ့ ပျမ်းမျှ ဖြစ်သည်။

ဒါကြောင့် `addNum` က $O(\log n)$၊ `findMedian` က $O(1)$ (ထိပ် ၂ ခု ဖတ်ရုံ) ဖြစ်ပါတယ်။

> **ဘာကြောင့် Heap ၂ ခု လိုသလဲ?** Median ဆိုတာ "အလယ်" ဖြစ်လို့ ဘယ်ခြမ်းရဲ့ **အကြီးဆုံး** (Max-Heap) နဲ့ ညာခြမ်းရဲ့ **အငယ်ဆုံး** (Min-Heap) ၂ ခု သိရင် လုံလောက်ပါတယ်။ Heap ၂ ခုက အဲ့ "နယ်စပ် ၂ ဖက်" ကို $O(1)$ နဲ့ ပေးနိုင်လို့ အကောင်းဆုံး ဖြစ်ပါတယ်။

> **Time Complexity:** `addNum` — $O(\log n)$၊ `findMedian` — $O(1)$။
> **Space Complexity:** $O(n)$ - ကိန်းအားလုံး heap ၂ ခုထဲ သိမ်း။

Java Solution

```
import java.util.*;

class MedianFinder {
    private PriorityQueue<Integer> small; // Max-Heap: အငယ်ပိုင်း
    private PriorityQueue<Integer> large; // Min-Heap: အကြီးပိုင်း
```

```java
    public MedianFinder() {
        small = new PriorityQueue<>(Collections.reverseOrder());
        large = new PriorityQueue<>();
    }

    public void addNum(int num) {
        small.offer(num);              // အရင် small ထဲ ထည့်
        large.offer(small.poll());     // small ၏ အကြီးဆုံးကို large သို့ ရွှေ့ (အစီအစဉ် မှန်အောင်)

        // balance: large က small ထက် များနေရင် ပြန်ညှိ
        if (large.size() > small.size()) {
            small.offer(large.poll());
        }
    }

    public double findMedian() {
        if (small.size() > large.size()) {
            return small.peek();       // အရေအတွက် မကိန်း — small ဘက် ပိုများ
        }
        // စုံ — ထိပ် ၂ ခု ပျမ်းမျှ ((double) cast အရင်လုပ်မှ int overflow မဖြစ်)
        return ((double) small.peek() + (double) large.peek()) / 2.0;
    }
}
```

အခန်း ၁၃ - Trees

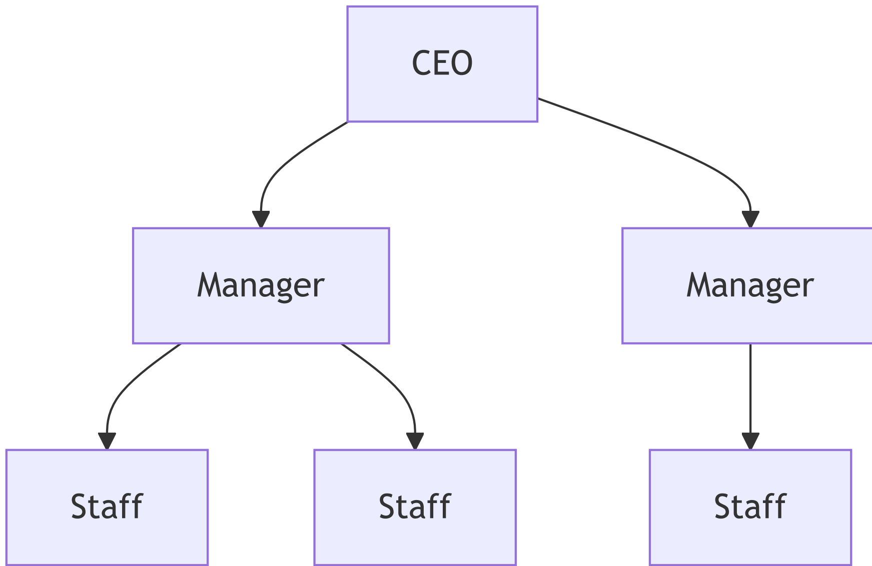
အရင်အခန်းတွေမှာ data တွေကို **Linear** ပုံစံနဲ့ သိမ်းတာ များခဲ့ပါတယ် — Array (အခန်း ၃) က ဆက်တိုက်၊ Linked List (အခန်း ၈) က node တစ်ခုပြီးတစ်ခု၊ Stack/Queue (အခန်း ၇) ကလည်း တစ်ကြောင်းတည်း။ ဒါပေမယ့် real-world data အများစုက တန်းလျား မဟုတ်ဘဲ **hierarchical** ဖွဲ့စည်းပုံ ဖြစ်ပါတယ် —

- ကွန်ပျူတာ ထဲက **folder** တွေ — folder တစ်ခုထဲမှာ folder တွေ၊ file တွေ ပါသည်။
- ကုမ္ပဏီ တစ်ခုရဲ့ **organization chart** — CEO အောက်မှာ manager တွေ၊ သူတို့အောက်မှာ staff တွေ။
- E-commerce site ရဲ့ **category** — "Electronics" အောက်မှာ "Phone", "Laptop"; "Phone" အောက်မှာ brand တွေ။
- Facebook/Reddit ရဲ့ **comment reply** — comment တစ်ခုအောက်မှာ reply တွေ၊ reply အောက်မှာ ထပ် reply တွေ။

ဒီလို "တစ်ခုအောက်မှာ တစ်ခု ဆက်ခွဲ" ဖွဲ့စည်းပုံကို ကိုယ်စားပြုဖို့ **Tree** data structure ကို သုံးပါတယ်။ ဒီအခန်းမှာ Tree ဆိုတာ ဘာလဲ၊ ဝေါဟာရတွေ (node, root, leaf, height, depth)၊ Binary Tree၊ ပြီးတော့ Tree ကို ဖြတ်သန်း (traversal) ဖို့ နည်း ၄ မျိုး — Preorder, Inorder, Postorder, Level order — ကို လေ့လာသွားပါမယ်။ နောက်ဆုံးမှာ classic ပြဿနာ ၅ ခုကို တစ်ဆင့်ချင်း ဖြေရှင်းကြည့်ပါမယ်။

Tree ဆိုတာ ဘာလဲ

Tree ဆိုတာ node တွေကို **parent-child (မိဘ-သားသမီး)** ဆက်ဆံရေးနဲ့ ချိတ်ဆက်ထားတဲ့ data structure ဖြစ်ပါတယ်။ သစ်ပင် တစ်ပင်ကို **မှောက်ထား** သလို — အမြစ် (root) က အပေါ်မှာ၊ အကိုင်းအခက် (branch) တွေက အောက်ဘက် ဖြန့်သွားတယ်လို့ မြင်ရင် ရပါတယ်။



`CEO` = root (အမြစ်), `Staff` တွေ = leaf (သားသမီး မရှိ)။

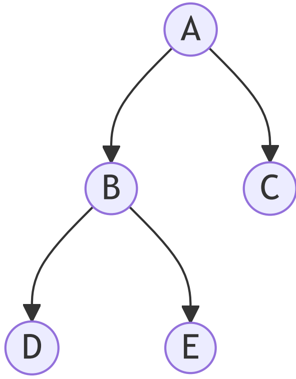
အဓိက ဝေါဟာရများ

ဝေါဟာရ	အဓိပ္ပါယ်
Node	tree ထဲက အချက်အလက် တစ်ခု (data + သားသမီးဆီ link)
Root	အပေါ်ဆုံး node (parent မရှိ)။ tree တစ်ခုမှာ ၁ ခုသာ
Parent	node တစ်ခုရဲ့ အပေါ်က node
Child	node တစ်ခုရဲ့ အောက်က node
Leaf	children မရှိတဲ့ node (အောက်ဆုံး)
Edge	node ၂ ခုကြား ချိတ်ဆက်မှု (link)
Subtree	node တစ်ခုနဲ့ သူ့အောက်က node အားလုံး ပေါင်းတဲ့ tree ငယ်

Height နဲ့ Depth

Tree မှာ မကြာခဏ ရောထွေးတတ်တဲ့ ဝေါဟာရ ၂ ခုက **height** နဲ့ **depth** ဖြစ်ပါတယ် —

- **Depth (အနက်)** — **root ကနေ** အဲ့ node အထိ edge အရေအတွက်။ root ရဲ့ depth က `0` ။
- **Height (အမြင့်)** — အဲ့ node ကနေ **အဝေးဆုံး leaf အထိ** edge အရေအတွက်။ leaf ရဲ့ height က `0` ။
- **Tree ၏ height** — root ရဲ့ height (အနက်ဆုံး leaf ရဲ့ depth)။



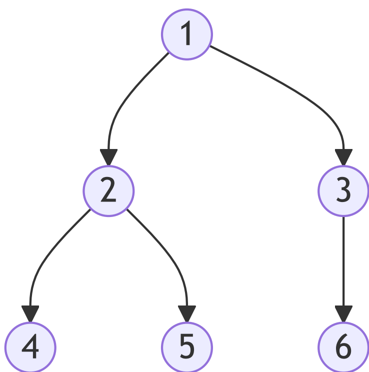
node	depth	height
A	0	2
B, C	1	1
D, E	2	0 (leaf)

Tree height = 2 (A ကနေ D/E အထိ edge ၂ ခု)။ depth က အလွှာတူ node တိုင်း တူပေမယ့်၊ height က node တစ်ခုချင်း ကွဲသည် — ဥပမာ C က leaf ဖြစ်လို့ height = 0။

> **မှတ်ချက်:** depth က "အပေါ်ကနေ ဘယ်လောက် နက်လဲ"၊ height က "အောက်ကို ဘယ်လောက် ကျန်သေးလဲ" လို့ မှတ်ရင် လွယ်ပါတယ်။ စာအုပ်/language အချို့မှာ edge အရေအတွက် မဟုတ်ဘဲ node အရေအတွက်နဲ့ ရေတွက်တာ ရှိလို့ (root depth = 1) သတိပြုပါ — ဒီစာအုပ်မှာ edge အရေအတွက်နဲ့ ရေတွက်ပါတယ်။

Binary Tree

Tree တစ်ခုမှာ node တစ်ခုစီ သားသမီး ဘယ်နှစ်ခု ရှိနိုင်လဲ ဆိုတာ ကန့်သတ်ချက် မရှိပါ (folder တစ်ခုထဲမှာ folder ၁၀ ခု ထားလို့ ရတယ်)။ ဒါပေမယ့် algorithm လေ့လာတဲ့အခါ အသုံးအများဆုံးက **Binary Tree — node တစ်ခုစီမှာ children အများဆုံး ၂ ခု** (left child, right child) ရှိတဲ့ tree ဖြစ်ပါတယ်။



node တစ်ခုစီမှာ left, right အများဆုံး ၂ ခုစီ။

Binary Tree node တစ်ခုကို code မှာ ဒီလို ဖော်ပြပါတယ် — Linked List (အခန်း ၈) က node နဲ့ ဆင်ပေမယ့် `next` တစ်ခုတည်း မဟုတ်ဘဲ **`left`**, **`right`** **၂ ခု** ရှိတာ ကွာပါတယ် —

```
// Binary Tree node — value တစ်ခု + သားသမီး ၂ ခုဆီ link
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) { this.val = val; }
}
```

> **Binary Tree အမျိုးအစားများ (အကြမ်းဖျင်း):**
> - **Full** — node တိုင်း သားသမီး `0` (သို့) `2` ခု ရှိ (၁ ခုတည်း မရှိ)။
> - **Complete** — အလွှာတိုင်း အပြည့်၊ နောက်ဆုံး အလွှာသာ ဘယ်ကနေ ဖြည့် (Heap — အခန်း ၁၂ — ပုံစံ)။
> - **Balanced** — ဘယ်/ညာ height ကွာဟမှု နည်း (search မြန်အောင်၊ အခန်း ၁၄ BST မှာ ဆက်လေ့လာရပါမယ်)။

Recursive Thinking

Tree ကို နားလည်ဖို့ အရေးကြီးဆုံး အချက်က — **Tree ဟာ recursive structure** ဖြစ်တာ ဖြစ်ပါတယ်။ node တစ်ခုရဲ့ left child ကိုယ်တိုင်က **tree တစ်ခု (left subtree)**၊ right child ကလည်း **tree တစ်ခု (right subtree)** ဖြစ်ပါတယ်။ ဒါကြောင့် Tree ပြဿနာ အများစုကို အခန်း ၁၀ က **recursion** နဲ့ ဖြေရှင်း နိုင်ပါတယ်။

Tree ပြဿနာ တွေးတဲ့ ပုံစံက အမြဲ တူပါတယ် —

1. **Base case:** node က `null` ဆိုရင် ဘာလုပ်မလဲ (များသောအားဖြင့် ရပ်)။
2. **Recursive case:** left subtree ကို ဖြေ၊ right subtree ကို ဖြေ၊ ပြီးတော့ လက်ရှိ node နဲ့ ပေါင်းစပ်။

```
solve(node):
    if node == null:  return base_value      ← base case
    left  = solve(node.left)                  ← left subtree ဖြေ
    right = solve(node.right)                 ← right subtree ဖြေ
    return combine(node, left, right)         ← ပေါင်းစပ်
```

ဒီ pattern ကို နားလည်ရင် Tree ပြဿနာ အများစုကို ဖြေနိုင်ပါပြီ။ အောက်က traversal တွေ၊ နောက်ပိုင်း ပြဿနာတွေ အကုန် ဒီ ပုံစံကနေ ဆင်းသက်လာတာ ဖြစ်ပါတယ်။

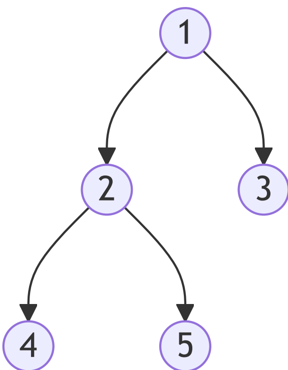
Tree Traversal

Traversal ဆိုတာ tree ထဲက node အားလုံးကို တစ်ခါ ဖြတ်သန်း (လည်ပတ်) ခြင်း ဖြစ်ပါတယ်။ Array ဆိုရင် ရှေ့ကနေ နောက် တန်းစီ ဖြတ်ရုံပါ။ ဒါပေမယ့် Tree က အကိုင်းအခက် ခွဲထားလို့ "ဘယ် node ကို အရင် ဖတ်မလဲ" ဆိုတဲ့ **အစီအစဉ်** အမျိုးမျိုး ရှိနိုင်ပါတယ်။

အဓိက ၂ မျိုး ရှိပါတယ် —

- **DFS (Depth-First Search)** — အကိုင်း တစ်ခုကို **အောက်ဆုံး အထိ** အရင် ဆင်းပြီးမှ ဘေး အကိုင်း ပြောင်း။ (recursion / stack နဲ့)
- **BFS (Breadth-First Search)** — **အလွှာလိုက်** (level order) — အပေါ်အလွှာ အကုန်ပြီးမှ အောက်အလွှာ။ (queue နဲ့)

DFS ကိုယ်တိုင် "လက်ရှိ node ကို ဘယ်အချိန် ဖတ်မလဲ" ပေါ်မူတည်ပြီး ၃ မျိုး ခွဲပါတယ် — **Preorder, Inorder, Postorder**။ အောက်က tree ကို ဥပမာ ထားပြီး ၄ မျိုးလုံး ကြည့်ရအောင် —



Preorder (Root → Left → Right)

လက်ရှိ node ကို အရင် ဖတ် ပြီးမှ left, right ဆင်းသည်။ Folder ထဲက file တွေ list လုပ်တာ၊ tree ကို copy/serialize လုပ်တာ မျိုးမှာ သုံးပါတယ်။

```
ဖတ်ပုံ:  1 → 2 → 4 → 5 → 3
        (root အရင်၊ ပြီးမှ ဘယ်ခြမ်း တစ်ခုလုံး၊ ပြီးမှ ညာခြမ်း)
```

```java
void preorder(TreeNode node) {
    if (node == null) return;        // base case
    System.out.print(node.val + " ");  // 1. root ဖတ်
    preorder(node.left);             // 2. left subtree
    preorder(node.right);            // 3. right subtree
}
```

Inorder (Left → Root → Right)

ဘယ်ခြမ်း အရင်၊ ပြီးမှ လက်ရှိ node၊ ပြီးမှ ညာခြမ်း။

```
ဖတ်ပုံ:  4 → 2 → 5 → 1 → 3
        (node တစ်ခုရဲ့ ဘယ်ဘက် အကုန်ပြီးမှ သူ့ကိုယ်သူ၊ ပြီးမှ ညာဘက်)
```

```
void inorder(TreeNode node) {
    if (node == null) return;
    inorder(node.left);             // 1. left subtree
    System.out.print(node.val + " ");  // 2. root ဖတ်
    inorder(node.right);            // 3. right subtree
}
```

Postorder (Left → Right → Root)

children ၂ ခုလုံး အရင်၊ လက်ရှိ node ကို **နောက်ဆုံး**။ folder size တွက်တာ (children အကုန် တွက်ပြီးမှ ကိုယ်တိုင်)၊ tree ကို delete လုပ်တာ မျိုးမှာ သုံးပါတယ်။

```
ဖတ်ပုံ:  4 → 5 → 2 → 3 → 1
        (children အကုန်ပြီးမှ parent — root ကို နောက်ဆုံး)
```

```
void postorder(TreeNode node) {
    if (node == null) return;
    postorder(node.left);           // 1. left subtree
    postorder(node.right);          // 2. right subtree
    System.out.print(node.val + " ");  // 3. root ဖတ် (နောက်ဆုံး)
}
```

> **DFS ၃ မျိုး မှတ်နည်း: "Pre/In/Post"** ဆိုတာ **root ကို ဘယ်အချိန် ဖတ်လဲ** ကို ဆိုလိုပါတယ် — **Pre** = root ကို **အရင်**၊ **In** = root ကို **အလယ်**, **Post** = root ကို **နောက်ဆုံး**။ left က right ထက် အမြဲ အရင် ဖြစ်ပါတယ်။

Level Order (BFS — အလွှာလိုက်)

DFS ၃ မျိုးက recursion (depth) နဲ့ ဆင်းတာ ဖြစ်ပြီး၊ **Level Order** ကတော့ **အလွှာတစ်ခုလုံး ပြီးမှ နောက်အလွှာ** ဖတ်တာ ဖြစ်ပါတယ်။ ဒါက **Queue (အခန်း ၇)** ကို သုံးပါတယ် — node တစ်ခု ဖတ်ပြီးတိုင်း childern တွေကို queue ထဲ ထည့်၊ queue ရှေ့ကနေ တစ်ခုချင်း ထုတ် ဖတ်သွားတာ ဖြစ်ပါတယ်။

```
ဖတ်ပုံ:  1 → 2 → 3 → 4 → 5
        (အလွှာ 0: [1]၊  အလွှာ 1: [2,3]၊  အလွှာ 2: [4,5])
```

```
import java.util.*;

void levelOrder(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);              // root ကို queue ထဲ ထည့်
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();    // ရှေ့ဆုံး ထုတ်
        System.out.print(node.val + " ");
        // သားသမီးတွေ queue ထဲ ထည့် (နောက်အလွှာ အတွက်)
        if (node.left != null)  queue.offer(node.left);
        if (node.right != null) queue.offer(node.right);
    }
```

```
}
```

> **DFS vs BFS ဘယ်အချိန် ဘာသုံးမလဲ:**
> - **DFS (recursion)** — code တို၊ "subtree တစ်ခုလုံး အဖြေ" လိုတဲ့ ပြဿနာ (depth, sum, path) တွေအတွက်။ tree အရမ်း နက်ရင် stack overflow (အခန်း ၁၀) ဖြစ်နိုင်ပါတယ်။
> - **BFS (queue)** — "အလွှာလိုက်" အဖြေ (level order, အနီးဆုံး node, tree ၏ width) လိုရင်။ tree ကျယ်ရင် queue memory များနိုင်ပါတယ်။

> **Traversal Complexity:** ၄ မျိုးလုံး node တိုင်းကို တစ်ခါစီ ဖတ်လို့ **Time** $O(n)$။ Space က DFS မှာ recursion stack အတွက် $O(h)$ (h = tree height)၊ BFS မှာ queue အတွက် $O(w)$ (w = တစ်အလွှာ၏ အများဆုံး node)။

Real-world Examples

Tree ဟာ "တစ်ခုအောက်မှာ တစ်ခု ဆက်ခွဲ" ဖွဲ့စည်းပုံ ရှိတဲ့ နေရာတိုင်းမှာ တွေ့ရပါတယ် —

- **File System** — folder အောက်မှာ folder/file တွေ။ folder size တွက်တာက postorder (သားသမီး အကုန် တွက်ပြီးမှ)။
- **Category Tree** — e-commerce ရဲ့ "Electronics → Phone → iPhone" မျိုး။ category page render လုပ်တာ preorder/level order။
- **Menu Tree** — app/website ရဲ့ navigation menu — submenu အောက်မှာ submenu။
- **Comment Replies** — comment အောက်က reply, reply အောက်က reply — recursion နဲ့ render။
- **Organization Hierarchy** — CEO → Manager → Staff။ "manager တစ်ယောက်အောက်က staff အကုန်" ရှာတာ subtree traversal။
- **DOM Tree** — HTML page ကိုယ်တိုင် Tree တစ်ခု — `<html>` root, အောက်မှာ `<body>` , `<div>` စသည်။

ဒီ pattern တွေ အကုန်လုံးက တူပါတယ် — **node တစ်ခုကို ဖြေဖို့ သူ့သားသမီး (subtree) တွေ အရင် ဖြေပြီး ပေါင်းစပ်** ဆိုတဲ့ recursive thinking ဖြစ်ပါတယ်။

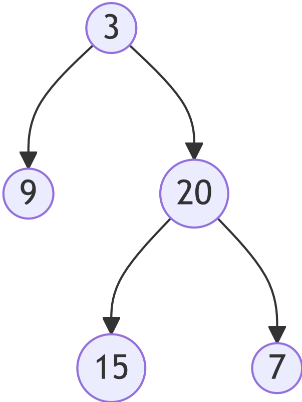
Questions

Tree ကို လက်တွေ့ ပြဿနာ ၅ ခုနဲ့ ချိတ်ဆက် လေ့လာကြည့်ရအောင်။ ပြဿနာတိုင်းရဲ့ သော့ချက်က — အပေါ်က **recursive thinking** ("base case ဘာလဲ၊ subtree ၂ ခုကို ဘယ်လို ပေါင်းစပ်မလဲ") ကို မှန်ကန်စွာ ရှာတတ်ဖို့ ဖြစ်ပါတယ်။

၁။ Maximum Depth of Binary Tree

Binary tree တစ်ခုရဲ့ root ပေးထားသည်။ **maximum depth** (root ကနေ အဝေးဆုံး leaf အထိ node အရေအတွက်) ကို ပြန်ပါ။

Example 1:



Output: `3` — 3 → 20 → 15 (သို့ 7) — node ၃ ခု နက်သည်။

ရှင်းလင်းချက်

ဒါက recursive thinking ရဲ့ အသန့်ရှင်းဆုံး ဥပမာ ဖြစ်ပါတယ်။ node တစ်ခုရဲ့ depth ကို ဘယ်လို တွက်မလဲ?

- **Base case:** node က `null` ဆိုရင် depth = `0` ။
- **Recursive case:** လက်ရှိ node ရဲ့ depth = `1 + (left subtree, right subtree ၂ ခုထဲက ပိုနက်တာ)` ။

ဆိုလိုတာက "ကိုယ့်အောက်က ၂ ဘက်ထဲ ဘယ်ဘက် ပိုနက်လဲ ကြည့်၊ အဲ့ထဲ ကိုယ့်ကို ၁ ပေါင်း" ဆိုတဲ့ သဘောပါ။

> **Time Complexity:** $O(n)$ - node တိုင်းကို တစ်ခါစီ။
> **Space Complexity:** $O(h)$ - recursion stack (h = height)။ အဆိုးဆုံး (skewed tree) $O(n)$။

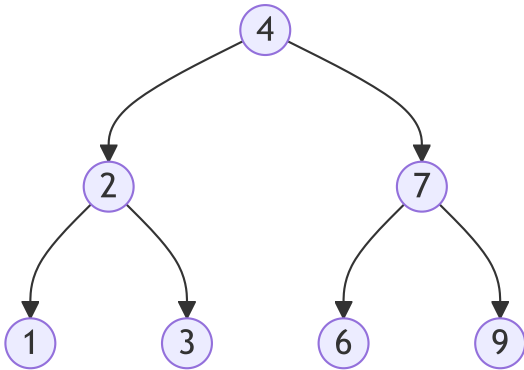
Java Solution

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;            // base case
        int left = maxDepth(root.left);        // ဘယ်ခြမ်း depth
        int right = maxDepth(root.right);      // ညာခြမ်း depth
        return 1 + Math.max(left, right);      // ပိုနက်တာ + ကိုယ်တိုင်
    }
}
```

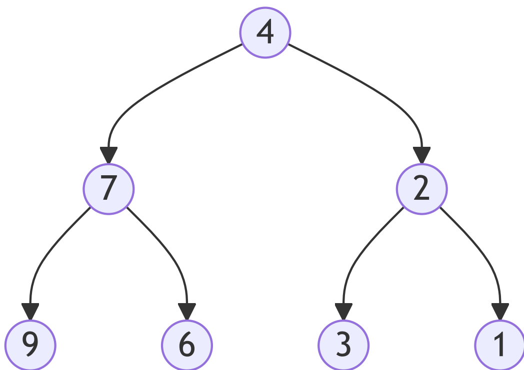
၂။ Invert Binary Tree

Binary tree တစ်ခုရဲ့ root ပေးထားသည်။ tree ကို **ဘယ်/ညာ ပြောင်းပြန်လှန် (mirror)** ပြီး root ကို ပြန်ပါ။

Example 1:



→ Output (ဘယ်/ညာ ပြန်လှန်):



ရှင်းလင်းချက်

"ပြောင်းပြန်လှန်" ဆိုတာ node တိုင်းရဲ့ **left နဲ့ right ကို နေရာချင်း လဲ** ဖို့ပါ။ recursive thinking နဲ့ —

- **Base case:** node က `null` ဆိုရင် ဘာမှ မလုပ်၊ `null` ပြန်။
- **Recursive case:** လက်ရှိ node ရဲ့ left, right ကို လဲ၊ ပြီးတော့ subtree ၂ ခုစလုံးကိုလည်း ပြန်လှန် (recursion)။

တစ်နည်းအားဖြင့် "ကိုယ့်သားသမီး ၂ ဦး လဲ၊ ပြီးတော့ သူတို့ကိုယ်စီကိုလည်း ပြန်လှန်ခိုင်း" ဆိုတဲ့ သဘောပါ။

> **Time Complexity:** $O(n)$ - node တိုင်းကို တစ်ခါစီ။
> **Space Complexity:** $O(h)$ - recursion stack။

Java Solution

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) return null;          // base case
        // left, right လဲ
```

```
            TreeNode temp = root.left;
            root.left = root.right;
            root.right = temp;
            // subtree ၂ ခုကိုလည်း ပြန်လှန်
            invertTree(root.left);
            invertTree(root.right);
            return root;
        }
    }
```

၃။ Same Tree

Binary tree ၂ ခု (`p` , `q`) ပေးထားသည်။ **တူညီလား** (structure တူ၊ value တူ) စစ်ပါ။

Example 1:

```
Input:    p:  1      q:  1
             / \        / \
            2   3      2   3
Output: true
```

Example 2:

```
Input:    p:  1      q:  1
             /            \
            2              2
Output: false  (structure မတူ — ဘယ်ဘက် vs ညာဘက်)
```

ရှင်းလင်းချက်

tree ၂ ခုကို တစ်ပြိုင်နက် ဖြတ်သန်းပြီး node တစ်ခုချင်း နှိုင်းယှဉ်ပါတယ်။

- **Base case 1:** node ၂ ခုလုံး `null` → ဒီနေရာ တူ (`true`)။
- **Base case 2:** တစ်ခုတည်း `null` (သို့) value မတူ → မတူ (`false`)။
- **Recursive case:** လက်ရှိ node တူရင်၊ left subtree ၂ ခု **နဲ့** right subtree ၂ ခု စလုံး တူမှ `true` ။

ဆိုလိုတာ "ကိုယ်တိုင်လည်း တူ၊ ဘယ်ခြမ်းလည်း တူ၊ ညာခြမ်းလည်း တူ" မှ tree တူတာ ဖြစ်ပါတယ်။

> **Time Complexity:** $O(n)$ - node တိုင်းကို တစ်ခါစီ (n = node အရေအတွက်)။
> **Space Complexity:** $O(h)$ - recursion stack။

Java Solution

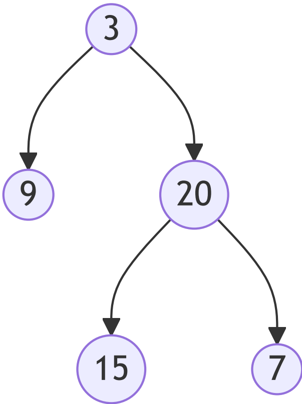
```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;     // ၂ ခုလုံး null → တူ
        if (p == null || q == null) return false;    // တစ်ခုတည်း null → မတူ
        if (p.val != q.val) return false;            // value မတူ → မတူ
        // ကိုယ်တိုင်တူ → ဘယ်/ညာ ၂ ခုစလုံး တူမှ true
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
```

```
        }
    }
```

၄။ Binary Tree Level Order Traversal

Binary tree တစ်ခုရဲ့ root ပေးထားသည်။ node တွေကို **အလွှာလိုက် (level order)** — အလွှာတစ်ခုစီကို list ခွဲ — ပြန်ပါ။

Example 1:



Output: `[[3], [9,20], [15,7]]`

ရှင်းလင်းချက်

ဒါက အပေါ်က **BFS (Queue)** ကို တိုက်ရိုက် အသုံးချတာ ဖြစ်ပါတယ်။ ဒါပေမယ့် ဒီတစ်ခါ အလွှာတစ်ခုစီကို **သီးခြား list** အဖြစ် ခွဲ ပြန်ဖို့ လိုတယ်။

အဓိက လှည့်ကွက် — loop တစ်ပတ် မစခင် **queue ထဲမှာ ရှိနေတဲ့ node အရေအတွက် (** `size` **)** ကို မှတ်သည်။ အဲ့ `size` က **လက်ရှိ အလွှာ၏ node အရေအတွက်** အတိအကျ ဖြစ်ပါတယ်။ အဲ့ အရေအတွက်အတိုင်း ထုတ်ပြီး တစ်အလွှာ ပြီးအောင် လုပ်သည်။

- queue ထဲ root ထည့်။
- queue မဗလာမချင်း — လက်ရှိ `size` (အလွှာ node အရေအတွက်) ကို မှတ်၊ အဲ့ အရေအတွက် အတိုင်း node ထုတ်ပြီး တစ် list ထဲ စု၊ သားသမီးတွေ queue ထဲ ထည့်။

> **Time Complexity:** $O(n)$ - node တိုင်းကို တစ်ခါစီ။
> **Space Complexity:** $O(n)$ - queue နဲ့ result (အဆိုးဆုံး အလွှာတစ်ခုမှာ node $n/2$ ခန့်)။

Java Solution

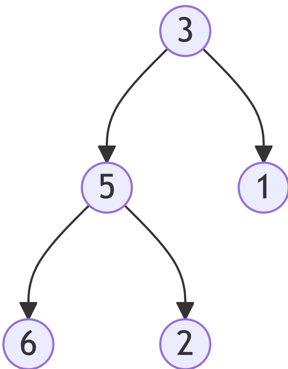
```java
import java.util.*;

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
```

```
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            int size = queue.size();            // လက်ရှိ အလွှာ node အရေအတွက်
            List<Integer> level = new ArrayList<>();
            for (int i = 0; i < size; i++) {     // ဒီ အလွှာအတွက်သာ
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null)  queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
            result.add(level);                  // တစ်အလွှာ ပြီး
        }
        return result;
    }
}
```

၅။ Lowest Common Ancestor of a Binary Tree

Binary tree တစ်ခုနဲ့ node ၂ ခု (`p` , `q`) ပေးထားသည်။ သူတို့ ၂ ခုရဲ့ **lowest common ancestor (LCA — အနိမ့်ဆုံး ဘုံ ဘိုးဘေး)** ကို ပြန်ပါ။ LCA ဆိုတာ `p` နဲ့ `q` ၂ ခုလုံးကို subtree အဖြစ် ထိန်းထားတဲ့ **အနက်ဆုံး node** ဖြစ်ပါတယ်။

Example 1:



`p = 6, q = 2` → Output: `5` (5 ၏ subtree ထဲ 6 နဲ့ 2 ၂ ခုလုံး ပါ)

ရှင်းလင်းချက်

ဒါက "comment thread ထဲ comment ၂ ခု ရဲ့ ဘုံ parent ရှာ" သို့ "folder ၂ ခု ရဲ့ ဘုံ folder ရှာ" မျိုး ပြဿနာ ဖြစ်ပါတယ်။ recursive thinking နဲ့ ရှင်းရှင်း ဖြေနိုင်ပါတယ် —

- **Base case:** node က `null` (သို့) `p` (သို့) `q` ဆိုရင် အဲ့ node ကို ပြန် (ဒီ subtree ထဲ ရှာတွေ့တဲ့ target)။
- **Recursive case:** left, right ၂ ဘက်မှာ ရှာ —
    - ၂ ဘက်စလုံး `null` မဟုတ် (target တစ်ခုစီ ၂ ဘက်မှာ ရှိ) → **လက်ရှိ node ကိုယ်တိုင်က LCA**။
    - တစ်ဘက်တည်းမှာ ရှိ → အဲ့ဘက်က ရလဒ်ကို အပေါ်ကို ပြန်တင်။

ဆိုလိုတာ — `p` နဲ့ `q` ၂ ခုက **ကွဲ** သွားတဲ့ နေရာ (node တစ်ခုက ဘယ်ဘက်၊ နောက်တစ်ခုက ညာဘက်) က LCA ဖြစ်ပါတယ်။

> **မှတ်ချက်:** ဒါက **သာမန် Binary Tree** အတွက် နည်း ဖြစ်ပါတယ်။ Binary Search Tree (အခန်း ၁၄) ဆိုရင် "ဘယ်ငယ်၊ ညာကြီး" rule ကို သုံးပြီး ပိုလွယ်လွယ် ($O(h)$) ရှာနိုင်ပါတယ်။

> **Time Complexity:** $O(n)$ - အဆိုးဆုံး node တိုင်းကို ဖတ်။
> **Space Complexity:** $O(h)$ - recursion stack။

Java Solution

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // base case: null သို့ target တစ်ခု တွေ့
        if (root == null || root == p || root == q) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        if (left != null && right != null) return root;  // ၂ ဘက်မှာ ကွဲ → ဒီ node က LCA
        return (left != null) ? left : right;             // တစ်ဘက်က ရလဒ် တင်
    }
}
```

အခန်း ၁၄ - Binary Search Tree

အခန်း ၁၃ မှာ **Binary Tree** ဆိုတာ node တစ်ခုစီ သားသမီး အများဆုံး ၂ ခု (left, right) ရှိတဲ့ tree ဖြစ်တာ၊ ပြီးတော့ traversal နည်း ၄ မျိုးကို လေ့လာခဲ့ပါတယ်။ ဒါပေမယ့် သာမန် Binary Tree မှာ node တွေရဲ့ တန်ဖိုး (value) တွေက **အစီအစဉ် မရှိ**ပါ — တန်ဖိုး x ရှိမရှိ ရှာဖို့ tree တစ်ခုလုံး (n node) ဖြတ်ရလို့ $O(n)$ ကုန်ပါတယ်။

အခန်း ၃ က **Binary Search** ကို ပြန်စဉ်းစားကြည့်ရအောင် — sorted array မှာ အလယ်ကို ကြည့်ပြီး "ရှာတဲ့ဟာ ပိုကြီးလား ပိုငယ်လား" အလိုက် တစ်ဝက် ဖြတ်ပစ်လို့ $O(\log n)$ နဲ့ ရှာနိုင်ခဲ့ပါတယ်။ ဒီ "တစ်ဝက် ဖြတ်ပစ်" idea ကို **Tree** ပေါ်မှာ တင်လိုက်ရင် ဘာဖြစ်မလဲ?

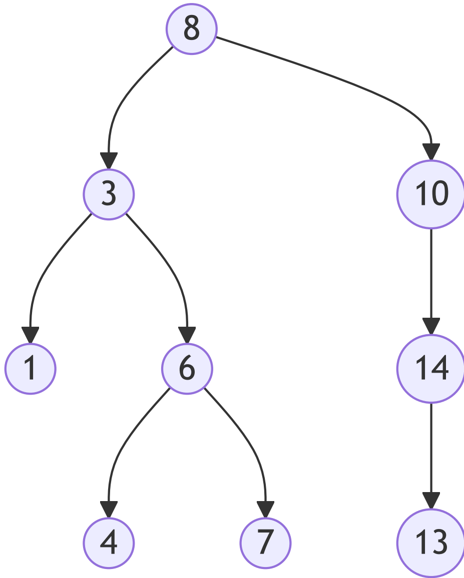
အဲ့ဒါက **Binary Search Tree (BST)** ဖြစ်ပါတယ် — **"ဘယ်ငယ်၊ ညာကြီး"** ဆိုတဲ့ စည်းမျဉ်း တစ်ခု ထည့်ထားတဲ့ Binary Tree ဖြစ်ပြီး၊ search/insert/delete တွေကို ပျမ်းမျှ $O(\log n)$ နဲ့ လုပ်နိုင်ပါတယ်။ ဒီအခန်းမှာ BST ဆိုတာ ဘာလဲ၊ "ဘယ်ငယ်၊ ညာကြီး" rule က ဘာကြောင့် search မြန်စေသလဲ၊ insert/delete ဘယ်လို လုပ်လဲ၊ balance ပျက်ရင် ဘာဖြစ်လဲ ဆိုတာတွေ လေ့လာပြီး၊ classic ပြဿနာ ၅ ခုကို တစ်ဆင့်ချင်း ဖြေရှင်းကြည့်ပါမယ်။

Binary Search Tree ဆိုတာ ဘာလဲ

Binary Search Tree (BST) ဆိုတာ node တိုင်းမှာ ဒီ စည်းမျဉ်း ပြည့်ရတဲ့ Binary Tree ဖြစ်ပါတယ် —

- **ဘယ် subtree** ထဲက node အားလုံးရဲ့ တန်ဖိုးက လက်ရှိ node ထက် **ငယ်** ရမည်။
- **ညာ subtree** ထဲက node အားလုံးရဲ့ တန်ဖိုးက လက်ရှိ node ထက် **ကြီး** ရမည်။
- ဘယ်/ညာ subtree ၂ ခုစလုံးကိုယ်တိုင်လည်း BST ဖြစ်ရမည် (recursive)။

အတိုချုံး **"ဘယ်ငယ်၊ ညာကြီး"** လို့ မှတ်ပါ။ ဒီ rule ပြည့်တဲ့ BST တစ်ခု ကြည့်ရအောင် —



- `8` (root): ဘယ်ဘက် (3,1,6,4,7) အကုန် < 8၊ ညာဘက် (10,14,13) အကုန် > 8
- `3` : ဘယ် (1) < 3 < ညာ (6,4,7)
- `10` : ညာ (14,13) > 10

> **သတိ — Heap နဲ့ မရောထွေးပါနဲ့:** အခန်း ၁၂ က **Heap** မှာ "မိဘ-သားသမီး" ဆက်ဆံရေးကိုသာ ထိန်းပြီး ဘေးချင်းကပ် (sibling) တွေကြား အစီအစဉ် မရှိပါ။ BST မှာတော့ **node အားလုံး** "ဘယ်ငယ်၊ ညာကြီး" အစီအစဉ်တကျ ရှိလို့ "တန်ဖိုး `x` ရှိလား" ကို မြန်မြန် ရှာနိုင်ပါတယ်။ Heap က "အကြီး/အငယ်ဆုံး" ထုတ်ဖို့ ကောင်းပြီး၊ BST က "ရှာဖို့/စီထားဖို့" ကောင်းပါတယ်။

BST node ကို code မှာ ဖော်ပြတာ အခန်း ၁၃ က `TreeNode` အတိုင်းပါပဲ —

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) { this.val = val; }
}
```

"ဘယ်ငယ်၊ ညာကြီး" rule က ဘာကြောင့် မြန်သလဲ

BST ရဲ့ အသက်က — node တစ်ခုကို ရောက်တိုင်း **ရှာတဲ့ တန်ဖိုးနဲ့ နှိုင်းယှဉ်ပြီး ဘယ် (သို့) ညာ တစ်ဘက်ကို ဆင်း** လို့ ရတာ ဖြစ်ပါတယ်။ ဒါက Binary Search (အခန်း ၃) က "အလယ်ကို ကြည့်ပြီး တစ်ဝက် ဖြတ်ပစ်" တာနဲ့ အတိအကျ တူပါတယ် —

- ရှာတဲ့ တန်ဖိုးက လက်ရှိ node ထက် **ငယ်** ရင် → **ဘယ်ဘက် ဆင်း** (ညာဘက် တစ်ခုလုံး မလို)။
- **ကြီး** ရင် → **ညာဘက် ဆင်း** (ဘယ်ဘက် တစ်ခုလုံး မလို)။
- **တူ** ရင် → တွေ့ပြီ။

အဆင့်တိုင်းမှာ subtree တစ်ခု (node တစ်ဝက်) ကို လုံးဝ ဖြတ်ပစ်လို့၊ tree က balance ဖြစ်ရင် အဆင့် $\log n$ ခုသာ ဆင်းရပါတယ်။ အပေါ်က tree မှာ `7` ကို ရှာကြည့်ရအောင် —

```
ရှာ: 7
  8 မှာ — 7 < 8 → ဘယ်ဆင်း     (10, 14, 13 ... တစ်ဝက် ဖြတ်)
  3 မှာ — 7 > 3 → ညာဆင်း     (1 ဖြတ်)
  6 မှာ — 7 > 6 → ညာဆင်း
  7 မှာ — 7 == 7 → တွေ့ပြီ ✓

   node ၄ ခုသာ ကြည့်ရ (tree မှာ node ၉ ခု ရှိ)
```

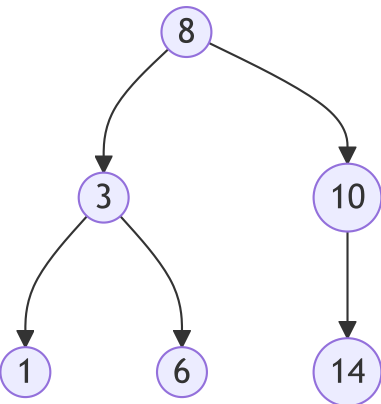
```
// BST ထဲ target ရှိမရှိ ရှာ (iterative)
boolean search(TreeNode root, int target) {
    TreeNode node = root;
    while (node != null) {
        if (target == node.val) return true;      // တွေ့ပြီ
        else if (target < node.val) node = node.left;   // ငယ် → ဘယ်
        else node = node.right;                   // ကြီး → ညာ
    }
    return false;                                 // ကုန်တဲ့အထိ မတွေ့
}
```

> **Complexity:** search က tree ရဲ့ **အမြင့် (h)** အတိုင်းသာ ဆင်းရလို့ $O(h)$ ဖြစ်ပါတယ်။ tree က balance ဖြစ်ရင် $h \approx \log n$ ဖြစ်လို့ $O(\log n)$။ (balance ပျက်ရင် ဘာဖြစ်လဲ နောက်မှာ ကြည့်ပါမယ်။)

Inorder Traversal

အခန်း ၁၃ က **Inorder traversal** (Left → Root → Right) ကို မှတ်မိမယ် ထင်ပါတယ်။ BST တစ်ခုကို Inorder ဖြတ်ရင် **တန်ဖိုးတွေ စီပြီးသား (sorted, ငယ်→ကြီး) ထွက်လာ** တာ ဖြစ်ပါတယ် — ဒါက BST ရဲ့ အရေးကြီးဆုံး ဂုဏ်သတ္တိ ဖြစ်ပါတယ်။

ဘာကြောင့်လဲ? Inorder က node တစ်ခုစီမှာ "ဘယ်ဘက် (ပိုငယ်တာတွေ) အရင်၊ ပြီးမှ ကိုယ်တိုင်၊ ပြီးမှ ညာဘက် (ပိုကြီးတာတွေ)" ဖတ်တာ ဖြစ်လို့ — "ဘယ်ငယ်၊ ညာကြီး" rule နဲ့ ပေါင်းလိုက်ရင် ငယ်ရာက ကြီးရာ အစီအစဉ်အတိုင်း ထွက်လာတာ ဖြစ်ပါတယ်။



Inorder: 1 → 3 → 6 → 8 → 10 → 14 (စီပြီးသား!)

```java
// BST ကို inorder ဖြတ်ရင် sorted ထွက်
void inorder(TreeNode node, List<Integer> result) {
    if (node == null) return;
    inorder(node.left, result);     // ဘယ် (ပိုငယ်) အရင်
    result.add(node.val);           // ကိုယ်တိုင်
    inorder(node.right, result);    // ညာ (ပိုကြီး) နောက်ဆုံး
}
```

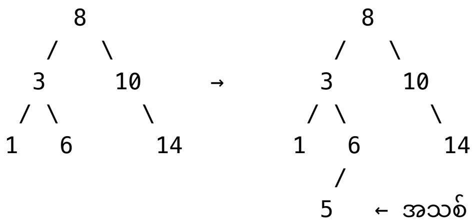
> **အသုံးဝင်ပုံ:** ဒီ ဂုဏ်သတ္တိကြောင့် "BST ကို sorted list ပြောင်း", "kth smallest ရှာ", "BST မှန်မမှန် စစ်" စတဲ့ ပြဿနာ အများစုကို Inorder နဲ့ ဖြေနိုင်ပါတယ်။ နောက်က Questions မှာ ဒါကို သုံးပါမယ်။

Insert — node အသစ် ထည့်ခြင်း

BST ထဲ တန်ဖိုး အသစ် တစ်ခု ထည့်တဲ့အခါ — **search လုပ်သလိုပဲ "ဘယ်ငယ်၊ ညာကြီး" အလိုက် ဆင်းသွား** ပြီး၊ နေရာလွတ် (`null`) ရောက်တဲ့အခါ အဲ့မှာ ချသည်။ ဒါက rule ကို မပျက်စေဘဲ မှန်ကန်တဲ့ နေရာ အလိုအလျောက် ရှာပေးပါတယ်။

`[8, 3, 10, 1, 6, 14]` BST ထဲ `5` ထည့်ကြည့်ရအောင် —

```
ထည့်: 5
  8 မှာ — 5 < 8 → ဘယ်ဆင်း
  3 မှာ — 5 > 3 → ညာဆင်း
  6 မှာ — 5 < 6 → ဘယ်ဆင်း
  6 ရဲ့ ဘယ်ဘက် null → ဒီမှာ ချ
```



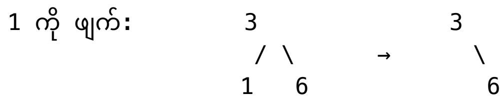

```java
// BST ထဲ value ထည့်ပြီး root ပြန်ပေး (recursive)
TreeNode insert(TreeNode root, int value) {
    if (root == null) return new TreeNode(value);   // နေရာလွတ် → ဒီမှာ ချ
    if (value < root.val) {
        root.left = insert(root.left, value);        // ငယ် → ဘယ်ဘက် ထည့်
    } else if (value > root.val) {
        root.right = insert(root.right, value);      // ကြီး → ညာဘက် ထည့်
    }
    // value == root.val ဆိုရင် (ထပ်) ဘာမှ မလုပ် — BST မှာ duplicate မထား
    return root;
}
```

> **Complexity:** insert ကလည်း မှန်ကန်တဲ့ နေရာ ရှာဖို့ tree ရဲ့ အမြင့်အတိုင်း ဆင်းရလို့ $O(h)$ (balance ဖြစ်ရင် $O(\log n)$) ဖြစ်ပါတယ်။

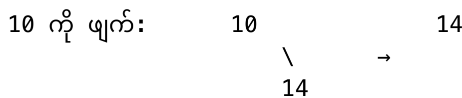
Delete — node ဖျက်ခြင်း (အခက်ဆုံး operation)

BST ထဲက node ဖျက်တာက အခက်ဆုံး operation ဖြစ်ပါတယ် — ဖျက်ပြီးနောက် "ဘယ်ငယ်၊ ညာကြီး" rule မပျက်အောင် ထိန်းရတဲ့အတွက် ဖြစ်ပါတယ်။ ဖျက်မယ့် node ကို ရှာတွေ့ပြီးတဲ့အခါ **အခြေအနေ ၃ မျိုး** ရှိပါတယ် —

အခြေအနေ ၁ — leaf (သားသမီး မရှိ): တိုက်ရိုက် ဖျက်ပစ်ရုံ (`null` ပြန်)။



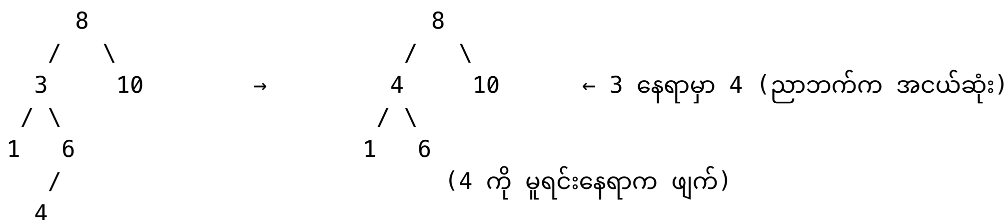
အခြေအနေ ၂ — သားသမီး ၁ ခု: အဲ့ node နေရာကို သူ့ **သားသမီး တစ်ခုတည်း**နဲ့ အစားထိုးရုံ။



အခြေအနေ ၃ — သားသမီး ၂ ခု: ဒါက အခက်ဆုံး။ အဲ့ node ကို ဖျက်ပေမယ့် rule မပျက်အောင် — **ညာ subtree ထဲက အငယ်ဆုံး node (inorder successor)** ကို ရှာ၊ အဲ့ တန်ဖိုးကို ဖျက်မယ့် node ပေါ် ကူးထည့်၊ ပြီးတော့ အဲ့ successor ကို ညာ subtree ထဲကနေ ဖျက်ပစ်သည်။

ဘာကြောင့် "ညာဘက်က အငယ်ဆုံး" လဲ? ဒါက ဖျက်မယ့် node ထက် **ကြီးတဲ့ တန်ဖိုးတွေထဲ အငယ်ဆုံး** ဖြစ်လို့ — အဲ့ နေရာ တင်ရင် ဘယ်ဘက် (အကုန် ပိုငယ်) ရော ညာဘက် (ကျန်တာ ပိုကြီး) ရော rule မပျက်ပါ။

3 ကို ဖျက် (သား ၂ ခု: 1, 6):




```
TreeNode delete(TreeNode root, int value) {
    if (root == null) return null;

    if (value < root.val) {
        root.left = delete(root.left, value);        // ဘယ်ဘက်မှာ ရှာ
    } else if (value > root.val) {
        root.right = delete(root.right, value);      // ညာဘက်မှာ ရှာ
    } else {
        // တွေ့ပြီ — အခြေအနေ ၃ မျိုး
        if (root.left == null) return root.right;    // သား ဝ/၁ (ဘယ်မရှိ)
```

```
        if (root.right == null) return root.left;     // သား ၁ (ညာမရှိ)

        // သား ၂ ခု — ညာဘက်က အငယ်ဆုံး (inorder successor) ရှာ
        TreeNode successor = root.right;
        while (successor.left != null) successor = successor.left;
        root.val = successor.val;                     // တန်ဖိုး ကူး
        root.right = delete(root.right, successor.val);  // successor ကို ဖျက်
    }
    return root;
}
```

> **Complexity:** delete ကလည်း node ရှာ + successor ရှာ — ၂ ခုစလုံး tree အမြင့်အတိုင်း ဖြစ်လို့ $O(h)$ ဖြစ်ပါတယ်။

Balance ပျက်ရင် ဘာဖြစ်လဲ — BST ၏ အားနည်းချက်

BST ရဲ့ operation အကုန်လုံး $O(h)$ (tree အမြင့်) ဖြစ်တာ သတိထားမိမယ် ထင်ပါတယ်။ tree က **balance** ဖြစ်ရင် $h \approx \log n$ ဖြစ်လို့ $O(\log n)$ — အရမ်းမြန်ပါတယ်။ ဒါပေမယ့် **balance ပျက်ရင်** h က n အထိ ကြီးနိုင်ပါတယ်။

ဥပမာ — **စီပြီးသား** data `[1, 2, 3, 4, 5]` ကို အစဉ်လိုက် insert လုပ်ရင် ဘာဖြစ်လဲ?

node တိုင်း ညာဘက်ပဲ ဆင်း — ဒါက Linked List (အခန်း ၈) နဲ့ အတူတူ! h = n

ဒီ "skewed (တစ်ဘက်စောင်း) tree" မှာ search/insert/delete အကုန် $O(n)$ ဖြစ်သွားပါတယ် — BST ရဲ့ အားသာချက် ပျောက်သွားပါတယ်။

Tree အခြေအနေ	အမြင့် h	Search/Insert/Delete
Balanced (ညီညာ)	$O(\log n)$	$O(\log n)$ ⚡

Skewed (တစ်ဘက်စောင်း)	$O(n)$	$O(n)$ 🐢

> **ဖြေရှင်းနည်း — Self-balancing BST:** ဒီ ပြဿနာကို ဖြေဖို့ insert/delete လုပ်တိုင်း tree ကို **အလိုအလျောက် balance ပြန်ညှိ** တဲ့ BST မျိုး ရှိပါတယ် — **AVL Tree**, **Red-Black Tree** စသည်။ ဒါတွေက balance ကို အမြဲ ထိန်းလို့ အဆိုးဆုံးမှာတောင် $O(\log n)$ အာမခံပါတယ်။ (Java ရဲ့ `TreeMap` / `TreeSet` , C++ ရဲ့ `std::map` တွေက Red-Black Tree နဲ့ လုပ်ထားတာ ဖြစ်ပါတယ်။) ဒီစာအုပ်မှာ self-balancing ရဲ့ အသေးစိတ်ကို မလေ့လာပေမယ့်၊ "လက်တွေ့မှာ ဒီ data structure တွေ သုံးတယ်" ဆိုတာ မှတ်ထားပါ။

Real-world Examples

BST (နဲ့ သူ့ self-balancing မျိုးကွဲတွေ) ဟာ "data ကို **စီထားပြီး** အမြန် ရှာ/ထည့်/ဖျက်" လိုတဲ့ နေရာတိုင်းမှာ တွေ့ရပါတယ် —

- **Database Index** — database က row တွေကို အမြန် ရှာဖို့ index ကို B-Tree / B+ Tree (BST နဲ့ ဆင်တဲ့ multi-way ordered search tree — binary မဟုတ်) နဲ့ သိမ်းသည်။ " `WHERE age > 30` " မျိုး range query မြန်စေသည်။
- **Map / Set (Sorted)** — Java `TreeMap` / `TreeSet` , C++ `std::map` — key တွေကို **စီပြီးသား** ထားရင်း $O(\log n)$ ရှာ/ထည့်/ဖျက်။
- **Auto-complete / Dictionary** — စာလုံး တစ်ခု ရိုက်တိုင်း "ဒီ prefix နဲ့ စတဲ့ စကားလုံးတွေ" ကို sorted order နဲ့ ပြ။
- **Range Query** — "ဈေးနှုန်း ၁၀၀၀ နဲ့ ၅၀၀၀ ကြား product တွေ" မျိုး — BST မှာ ဒီ range ကို မြန်မြန် ဖြတ်ထုတ်နိုင်။
- **Scheduling / Calendar** — event တွေကို အချိန်အလိုက် စီထား၊ "နောက်ထပ် event ဘယ်အချိန်လဲ" အမြန် ရှာ။

ဒီ pattern တွေ အကုန်လုံးက တူပါတယ် — **"data ကို စီထားရင်း၊ ရှာ/ထည့်/ဖျက် ၃ ခုစလုံးကို မကြာခဏ လုပ်ရတယ်"** ဆိုတဲ့ အခြေအနေပါ။ Sorted Array က ရှာတာ မြန်ပေမယ့် ထည့်/ဖျက်တာ $O(n)$ (shift လုပ်ရ)၊ Hash Map (အခန်း ၄) က ရှာ/ထည့်/ဖျက် မြန်ပေမယ့် **အစီအစဉ် မရှိ** — BST က ၃ ခုစလုံး $O(\log n)$ **နဲ့ အစီအစဉ်ပါ ထိန်း**ပေးတာ ဖြစ်ပါတယ်။

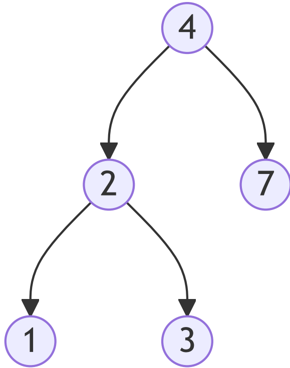
Questions

BST ကို လက်တွေ့ ပြဿနာ ၅ ခုနဲ့ ချိတ်ဆက် လေ့လာကြည့်ရအောင်။ ပြဿနာတိုင်းရဲ့ သော့ချက်က — **"ဘယ်ငယ်၊ ညာကြီး" rule (နဲ့ inorder က sorted ထွက်တာ)** ကို အသုံးချပြီး tree တစ်ဝက် ဖြတ်ပစ်တတ်ဖို့ ဖြစ်ပါတယ်။

၁။ Search in a Binary Search Tree

BST တစ်ခုရဲ့ root နဲ့ ကိန်း `val` ပေးထားသည်။ တန်ဖိုး `val` ရှိတဲ့ node ကို ရှာ၊ အဲ့ node ကို root ဖြစ်တဲ့ subtree ကို ပြန်ပါ။ မရှိရင် `null` ပြန်ပါ။

Example 1:



`val = 2` → Output: node `2` ၏ subtree (2, 1, 3)

ရှင်းလင်းချက်

ဒါက အပေါ်က **search** ကို တိုက်ရိုက် အသုံးချတာ ဖြစ်ပါတယ်။ "ဘယ်ငယ်၊ ညာကြီး" rule နဲ့ —

- `val` က လက်ရှိ node ထက် **ငယ်** ရင် → ဘယ်ဆင်း။
- **ကြီး** ရင် → ညာဆင်း။
- **တူ** ရင် → ဒီ node ပြန်။

တန်ဖိုး ထပ်ကြည့်ဖို့ subtree တစ်ဝက်ကို ဖြတ်ပစ်လို့ $O(h)$ နဲ့ ရပါတယ်။

> **Time Complexity:** $O(h)$ - tree အမြင့် (balance ဖြစ်ရင် $O(\log n)$)။
> **Space Complexity:** $O(1)$ - iterative (extra space မလို)။ recursive ဆိုရင် stack အတွက် $O(h)$။

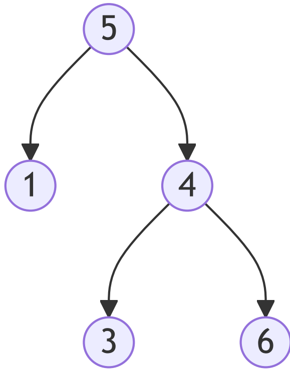
Java Solution

```java
class Solution {
    public TreeNode searchBST(TreeNode root, int val) {
        while (root != null && root.val != val) {
            root = (val < root.val) ? root.left : root.right;  // ငယ်→ဘယ်, ကြီး→ညာ
        }
        return root;   // တွေ့တဲ့ node (သို့) null
    }
}
```

၂။ Validate Binary Search Tree

Binary tree တစ်ခုရဲ့ root ပေးထားသည်။ အဲ့ tree က **မှန်ကန်တဲ့ BST** ဟုတ်မဟုတ် စစ်ပါ။

Example 1:



Output: `false` — `4` က `5` ၏ ညာဘက်မှာ ရှိပေမယ့် `5` ထက် ငယ်နေ၊ `3` ကလည်း `5` ထက် ငယ်ပေမယ့် ညာဘက်မှာ ရှိ

ရှင်းလင်းချက်

သတိ ထားရမယ့် ထောင်ချောက် — node တစ်ခုစီကို "left child < ကိုယ် < right child " လောက်သာ စစ်ရင် **မလုံလောက်**ပါ။ အပေါ်က ဥပမာမှာ `3` က သူ့မိဘ `4` ထက် ငယ်ပေမယ့်၊ tree တစ်ခုလုံး အနေနဲ့ `3` က root `5` ရဲ့ **ညာဘက်** subtree ထဲ ရှိနေတဲ့အတွက် `5` ထက် ကြီးရမယ် — ဒါ ချိုးဖောက်ထားပါတယ်။

မှန်ကန်တဲ့ နည်းက — node တစ်ခုစီအတွက် **ခွင့်ပြု အပိုင်းအခြား (min, max)** ကို လိုက်သယ်သွားတာ ဖြစ်ပါတယ် —

- ဘယ်ဘက် ဆင်းတဲ့အခါ — **အမြင့်ဆုံး (max)** က လက်ရှိ node တန်ဖိုး ဖြစ်လာသည် (ဘယ်ဘက် အကုန် ဒီထက် ငယ်ရမယ်)။
- ညာဘက် ဆင်းတဲ့အခါ — **အနိမ့်ဆုံး (min)** က လက်ရှိ node တန်ဖိုး ဖြစ်လာသည် (ညာဘက် အကုန် ဒီထက် ကြီးရမယ်)။
- node တစ်ခုစီ သူ့ `(min, max)` အတွင်း ရှိမှ မှန်။

> **အခြားနည်း:** BST ကို **inorder** ဖြတ်ရင် sorted ထွက်ရမယ် — ဒါကြောင့် inorder ဖတ်ရင်း "အရင် ဖတ်ထားတာထက် အမြဲ ကြီးနေလား" စစ်ရင်လည်း ရပါတယ်။ ဒီမှာ (min, max) range နည်းကို ပြထားပါတယ်။

> **Time Complexity:** $O(n)$ - node တိုင်းကို တစ်ခါစီ။
> **Space Complexity:** $O(h)$ - recursion stack။

Java Solution

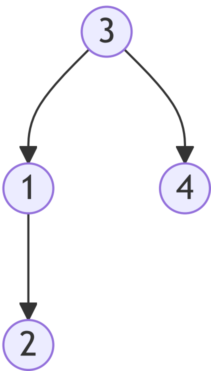
```java
class Solution {
    public boolean isValidBST(TreeNode root) {
        return validate(root, null, null);   // အစ — range ကန့်သတ်ချက် မရှိ
    }

    // node က (min, max) အတွင်း ရှိရမည် (null = ကန့်သတ်မရှိ)
```

```
        private boolean validate(TreeNode node, Integer min, Integer max) {
            if (node == null) return true;                       // base case
            if (min != null && node.val <= min) return false;    // အနိမ့် ဖောက်
            if (max != null && node.val >= max) return false;    // အမြင့် ဖောက်
            // ဘယ်ဆင်း — max ကို node.val အဖြစ် ကျုံ့၊ ညာဆင်း — min ကို node.val
            return validate(node.left, min, node.val)
                && validate(node.right, node.val, max);
        }
    }
```

၃။ Kth Smallest Element in a BST

BST တစ်ခုရဲ့ root နဲ့ ကိန်း `k` ပေးထားသည်။ **`k` ခုမြောက် အငယ်ဆုံး (kth smallest)** တန်ဖိုးကို ပြန်ပါ (`k` က ၁ ကနေ စသည်)။

Example 1:



`k = 1` → Output: `1` (inorder = [1,2,3,4] → ၁ ခုမြောက် အငယ်ဆုံး)

ရှင်းလင်းချက်

ဒါက BST ရဲ့ **inorder = sorted** ဂုဏ်သတ္တိ ကို တိုက်ရိုက် အသုံးချတာ ဖြစ်ပါတယ်။ BST ကို inorder ဖြတ်ရင် တန်ဖိုးတွေ **ငယ်→ကြီး** ထွက်လာလို့၊ inorder ရဲ့ **`k` ခုမြောက်** က kth smallest ဖြစ်ပါတယ်။

အရေးကြီးတာ — tree တစ်ခုလုံး inorder လုပ်စရာ မလို။ inorder ဖြတ်ရင်း **`k` ခုမြောက် ရောက်တာနဲ့ ရပ်** လို့ ရပါတယ်။

- inorder အစီအစဉ် (ဘယ် → ကိုယ် → ညာ) နဲ့ ဆင်း။
- node တစ်ခု ဖြတ်တိုင်း counter တိုး။ counter က `k` ရောက်ရင် အဲ့ node က အဖြေ။

> **Time Complexity:** $O(h+k)$ - အငယ်ဆုံးဆီ ဆင်းဖို့ $O(h)$ + node `k` ခု ဖတ်။ အဆိုးဆုံး $O(n)$။
> **Space Complexity:** $O(h)$ - recursion stack။

Java Solution

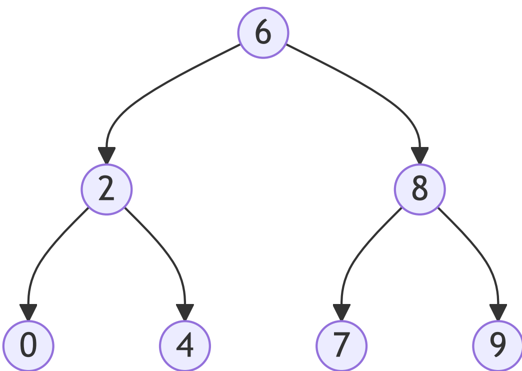
```
    class Solution {
        private int count = 0;
```

```
    private int result = 0;

    public int kthSmallest(TreeNode root, int k) {
        inorder(root, k);
        return result;
    }

    // true ပြန်ရင် "k ခုမြောက် တွေ့ပြီ" → recursion အကုန် ရပ်
    private boolean inorder(TreeNode node, int k) {
        if (node == null) return false;
        if (inorder(node.left, k)) return true;   // ဘယ် (ပိုငယ်) အရင်
        count++;                                   // node တစ်ခု ဖတ်ပြီ
        if (count == k) {                          // k ခုမြောက် ရောက်ပြီ
            result = node.val;
            return true;                           // → ဒီကနေ အပေါ်ကို ရပ်တန့်
        }
        return inorder(node.right, k);             // ညာ
    }
}
```

၄။ Lowest Common Ancestor of a BST

BST တစ်ခုနဲ့ node ၂ ခု (`p` , `q`) ပေးထားသည်။ သူတို့ ၂ ခုရဲ့ **lowest common ancestor (LCA)** ကို ပြန်ပါ။

Example 1:



`p = 2, q = 8` → Output: `6` (2 က ဘယ်ဘက်, 8 က ညာဘက် — ၆ မှာ ကွဲ)

ရှင်းလင်းချက်

အခန်း ၁၃ မှာ **သာမန် Binary Tree** အတွက် LCA ကို subtree ၂ ဘက် ရှာပြီး $O(n)$ နဲ့ ဖြေခဲ့ပါတယ်။ BST မှာတော့ **"ဘယ်ငယ်၊ ညာကြီး" rule** ကြောင့် ပိုလွယ်လွယ် ($O(h)$) ဖြေနိုင်ပါတယ်။

idea — node တစ်ခုမှာ ရပ်ပြီး `p` , `q` ၂ ခု ဘယ်ဘက်လား ညာဘက်လား ကြည့်ရုံပါ —

- `p` , `q` **၂ ခုလုံး လက်ရှိ node ထက် ငယ်** ရင် → LCA က ဘယ်ဘက်မှာ → **ဘယ်ဆင်း**။
- **၂ ခုလုံး ကြီး** ရင် → ညာဘက်မှာ → **ညာဆင်း**။
- **ကွဲ** သွားရင် (တစ်ခု ငယ်၊ တစ်ခု ကြီး၊ သို့ တစ်ခုက လက်ရှိ node နဲ့ တူ) → **လက်ရှိ node ကိုယ်တိုင်က LCA**။

ဆိုလိုတာ — `p` နဲ့ `q` ၂ ခု ပထမဆုံး **ကွဲသွားတဲ့ နေရာ** က LCA ဖြစ်ပါတယ်။

> **Time Complexity:** $O(h)$ - တစ်ဘက်တည်း ဆင်း (balance ဖြစ်ရင် $O(\log n)$)။
> **Space Complexity:** $O(1)$ - iterative။

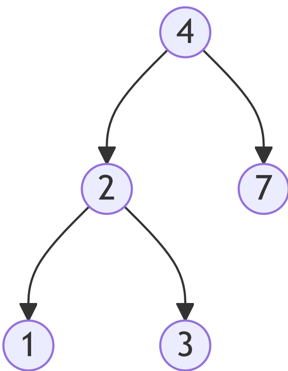
Java Solution

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        while (root != null) {
            if (p.val < root.val && q.val < root.val) {
                root = root.left;        // ၂ ခုလုံး ငယ် → ဘယ်
            } else if (p.val > root.val && q.val > root.val) {
                root = root.right;       // ၂ ခုလုံး ကြီး → ညာ
            } else {
                return root;             // ကွဲ → ဒီ node က LCA
            }
        }
        return null;
    }
}
```

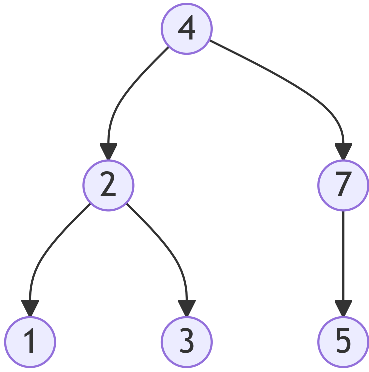
၅။ Insert into a Binary Search Tree

BST တစ်ခုရဲ့ root နဲ့ ကိန်း `val` ပေးထားသည်။ `val` ကို BST ထဲ ထည့်ပြီး (rule မပျက်စေဘဲ) update လုပ်ထားတဲ့ tree ရဲ့ root ကို ပြန်ပါ။ `val` က tree ထဲ မရှိဘူးလို့ အာမခံထားသည်။

Example 1:



`val = 5` ထည့်ပြီးရင် (5 > 4 → ညာ, 5 < 7 → ဘယ်, နေရာလွတ်မှာ ချ):



ရှင်းလင်းချက်

ဒါက အပေါ်က **Insert** ကို တိုက်ရိုက် အသုံးချတာ ဖြစ်ပါတယ်။ search လုပ်သလိုပဲ "ဘယ်ငယ်၊ ညာကြီး" အလိုက် ဆင်းသွားပြီး — **နေရာလွတ် (`null`) ရောက်တာနဲ့ အဲ့မှာ node အသစ် ချ**ရုံပါ။ ဒါက rule ကို မပျက်စေဘဲ မှန်ကန်တဲ့ နေရာ အလိုအလျောက် ရှာပေးပါတယ်။

- `val` က လက်ရှိ node ထက် **ငယ်** ရင် → ဘယ်ဘက် ဆက်ဆင်း။
- **ကြီး** ရင် → ညာဘက် ဆက်ဆင်း။
- `null` ရောက်ရင် → အဲ့မှာ node အသစ် ချ။

> **Time Complexity:** $O(h)$ - နေရာလွတ် ရှာဖို့ tree အမြင့်အတိုင်း ဆင်း (balance ဖြစ်ရင် $O(\log n)$)။
> **Space Complexity:** $O(h)$ - recursion stack။ iterative ဆိုရင် $O(1)$။

Java Solution

```
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) return new TreeNode(val);  // နေရာလွတ် → ဒီမှာ ချ
        if (val < root.val) {
            root.left = insertIntoBST(root.left, val);    // ငယ် → ဘယ်
        } else {
            root.right = insertIntoBST(root.right, val);  // ကြီး → ညာ
        }
        return root;
    }
}
```

အခန်း ၁၅ - Tries

အခန်း ၁၄ မှာ **Binary Search Tree** က "ဘယ်ငယ်၊ ညာကြီး" rule နဲ့ တန်ဖိုးတွေကို စီထားပြီး $O(\log n)$ နဲ့ ရှာတာ လေ့လာခဲ့ပါတယ်။ ဒါပေမယ့် BST (ရော Hash Map အခန်း ၄ ပါ) က "တန်ဖိုး `x` **အတိအကျ** ရှိလား" ဆိုတာ ဖြေဖို့ ကောင်းပေမယ့် — **" `cat` နဲ့ စတဲ့ စကားလုံး အကုန် ဘာတွေလဲ"** မျိုး **prefix (ရှေ့ဆုံး အပိုင်း)** နဲ့ ရှာတဲ့ ပြဿနာမှာ မကောင်းပါ။

ဥပမာ — Hash Map ထဲ စကားလုံး တစ်သိန်း ထည့်ထားပြီး " `app` နဲ့ စတာ ဘယ်နှလုံးလဲ" မေးရင် — Hash Map က key တစ်ခုလုံး အတိအကျ ကိုက်မှ ရှာတာမို့ စကားလုံး **တစ်သိန်းလုံး** တစ်ခုချင်း prefix စစ်ရ ($O(n \cdot L)$ — n = စကားလုံး အရေအတွက်၊ L = prefix အရှည်)။ ဒါက ဖုန်း keyboard က autocomplete, search box က suggestion မျိုး "စာရိုက်တိုင်း ချက်ချင်း ပြ" ရမယ့် နေရာမှာ နှေးလွန်းပါတယ်။

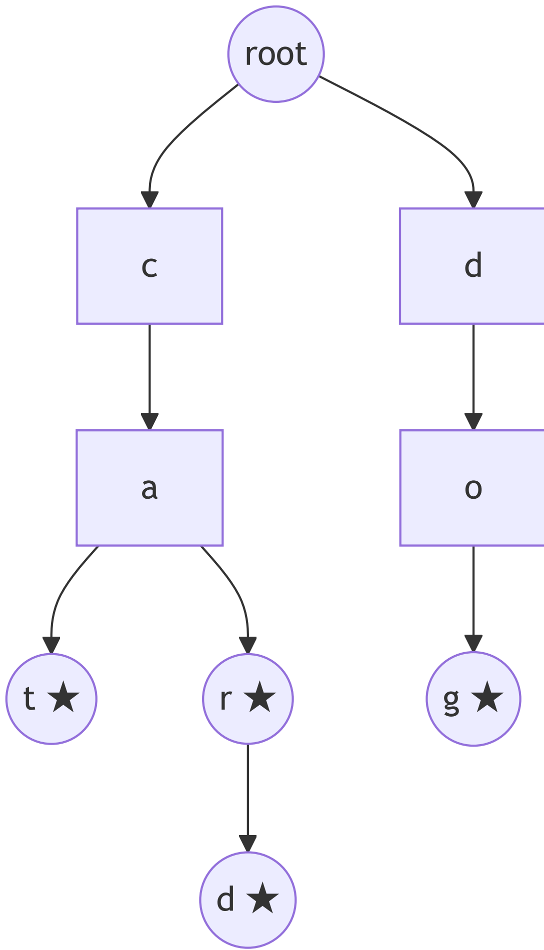
ဒီ ပြဿနာကို ဖြေဖို့ **Trie** (prefix tree လို့လည်း ခေါ်) ကို သုံးပါတယ် — စကားလုံးတွေကို **စာလုံး (character) တစ်လုံးချင်း** node အဖြစ် tree ပုံစံ သိမ်းထားတဲ့ data structure ဖြစ်ပါတယ်။ ဒီအခန်းမှာ Trie ဆိုတာ ဘာလဲ၊ insert/search/prefix ဘယ်လို လုပ်လဲ၊ Hash Table နဲ့ ဘာကွာလဲ၊ memory က ဘယ်လောက် ကုန်လဲ ဆိုတာတွေ လေ့လာပြီး classic ပြဿနာ ၄ ခုကို တစ်ဆင့်ချင်း ဖြေကြည့်ပါမယ်။

> **အမည်အကြောင်း:** Trie ဆိုတာ re**trie**val (ပြန်ထုတ်ယူခြင်း) ကနေ လာတာဖြစ်ပြီး၊ "ထရိုင်း" (try) လို့ ဖတ်သူ ရော "ထရီး" (tree) လို့ ဖတ်သူ ရော ရှိပါတယ်။

Trie ဆိုတာ ဘာလဲ

Trie ဆိုတာ စကားလုံးတွေကို **စာလုံး တစ်လုံးချင်းစီ** node အဖြစ် သိမ်းထားတဲ့ tree ဖြစ်ပါတယ်။ အဓိက idea က — **prefix တူတဲ့ စကားလုံးတွေ** ဟာ tree ထဲမှာ **လမ်းကြောင်း (path) အစ ပိုင်း တူတူ** ကို မျှဝေသုံးတာ ဖြစ်ပါတယ်။

`["cat", "car", "card", "dog"]` ၄ လုံးကို Trie ထဲ သိမ်းကြည့်ရအောင် —



★ = `isEnd` (ဒီ node မှာ စကားလုံး တစ်လုံး ဆုံး)။ စကားလုံးတွေကို ဒီလို ဖတ်ရ:

- `"cat"` : root → c → a → t★
- `"car"` : root → c → a → r★
- `"card"` : root → c → a → r → d★
- `"dog"` : root → d → o → g★

သတိထားရမှာ —

- **"ca"** ဆိုတဲ့ prefix ကို `cat` , `car` , `card` ၃ လုံးလုံး **မျှသုံး**ထားသည် (node `c` → `a` တစ်ကြောင်းတည်း)။
- node တစ်ခုစီမှာ **"ဒီမှာ စကားလုံး ဆုံးသလား" (** `isEnd` **)** အမှတ်အသား ရှိရသည် — `car` (r★) နဲ့ `card` (d★) ကို ခွဲဖို့။ `r` က ဆုံးမှတ်ဖြစ်ပေမယ့် သူ့အောက်မှာ `d` ဆက်ရှိနေသေးသည်။
- root က **ဗလာ** (စာလုံး မထား) — ဒါက "စကားလုံး အားလုံး စတဲ့ နေရာ" ဖြစ်သည်။

Trie node ကို code မှာ ဖော်ပြတာ —

```
class TrieNode {
    TrieNode[] children = new TrieNode[26];  // 'a'..'z' အတွက် ၂၆ ကွက်
    boolean isEnd = false;                   // ဒီ node မှာ စကားလုံး ဆုံးသလား
}
```

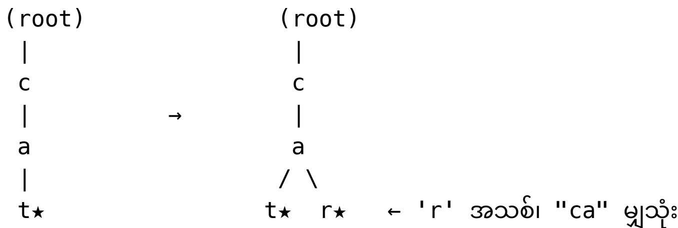
> `children` ၂ မျိုး: Java မှာ အင်္ဂလိပ် lowercase ၂၆ လုံးသာ ဆိုရင် `TrieNode[26]` array သုံးတာ မြန်ပါတယ် (`c - 'a'` နဲ့ index ထုတ်)။

Insert — စကားလုံး ထည့်ခြင်း

Trie ထဲ စကားလုံး တစ်လုံး ထည့်တဲ့အခါ — root ကနေ စပြီး **စာလုံး တစ်လုံးချင်း လိုက်ဆင်း** သွားသည်။ လမ်းကြောင်း မရှိသေးရင် node အသစ် ဆောက်၊ ရှိပြီးသားဆို မျှသုံး။ နောက်ဆုံး စာလုံး ရောက်ရင် အဲ့ node ကို `isEnd = true` မှတ်သည်။

`cat` ရှိပြီးသား Trie ထဲ `car` ထည့်ကြည့်ရအောင် —

```
ထည့်: "car"
  root → 'c' ရှိပြီးသား → မျှသုံး (ဆင်း)
  c    → 'a' ရှိပြီးသား → မျှသုံး (ဆင်း)
  a    → 'r' မရှိ → node အသစ် ဆောက် (ဆင်း)
  r    → စကားလုံး ဆုံး → isEnd = true ★
```




```
// Trie ထဲ word ထည့်
void insert(String word) {
    TrieNode node = root;
    for (char c : word.toCharArray()) {
        int i = c - 'a';                        // 'a'→0, 'b'→1, ...
        if (node.children[i] == null) {
            node.children[i] = new TrieNode();  // လမ်းကြောင်း မရှိ → ဆောက်
        }
        node = node.children[i];                // တစ်ဆင့် ဆင်း
    }
    node.isEnd = true;                          // နောက်ဆုံး → စကားလုံး ဆုံးမှတ်
}
```

> **Complexity:** insert က စကားလုံး အရှည် L အတိုင်းသာ ဆင်းရလို့ $O(L)$ ဖြစ်ပါတယ် — Trie ထဲ စကားလုံး ဘယ်နှလုံး ရှိရှိ (n ဘယ်လောက်ကြီးကြီး) insert က L ပေါ်သာ မူတည်တာ အရေးကြီးပါတယ်။

Search — စကားလုံး ရှာခြင်း

စကားလုံး တစ်လုံး Trie ထဲ ရှိမရှိ ရှာတာက — insert လိုပဲ root ကနေ **စာလုံး တစ်လုံးချင်း လိုက်ဆင်း** ပြီး၊ ၂ ချက် စစ်ရသည် —

1. စာလုံး တစ်ခုခု အတွက် လမ်းကြောင်း **မရှိ** ရင် → `false` (မရှိ)။
2. စာလုံး အကုန် ဆင်းပြီး နောက်ဆုံး node မှာ **`isEnd == true`** မှ → `true` ။ `isEnd` မဟုတ်ရင် ဒါက "prefix သာ ဖြစ်ပြီး စကားလုံး အပြည့် မဟုတ်" လို့ ဆိုလိုသည်။

ဒုတိယ ချက်က အရေးကြီးပါတယ် — `card` ထည့်ထားတဲ့ Trie မှာ `car` ရှာရင် လမ်းကြောင်း `c→a→r` တွေ့ပေမယ့်၊ `r` မှာ `isEnd` မဟုတ်ရင် `card` သာ ထည့်ထားပြီး `car` က မထည့်ရသေး လို့ ဆိုလိုပါတယ်။

```
// word အပြည့်အစုံ ရှိမရှိ
boolean search(String word) {
    TrieNode node = findNode(word);          // လမ်းကြောင်း လိုက်ဆင်း
    return node != null && node.isEnd;        // ဆုံးမှတ် ဖြစ်မှ true
}

// prefix လမ်းကြောင်း လိုက်ဆင်းပြီး နောက်ဆုံး node ပြန် (မရှိရင် null)
private TrieNode findNode(String prefix) {
    TrieNode node = root;
    for (char c : prefix.toCharArray()) {
        int i = c - 'a';
        if (node.children[i] == null) return null;   // လမ်းကြောင်း ပြတ် → မရှိ
        node = node.children[i];
    }
    return node;
}
```

Starts With — prefix နဲ့ ရှာခြင်း

Trie ရဲ့ **အသက်** က ဒီ operation ဖြစ်ပါတယ် — " `pre` ဆိုတဲ့ prefix နဲ့ စတဲ့ စကားလုံး **တစ်ခုခု** ရှိလား" ဆိုတာ စစ်တာ။ search နဲ့ ကွာတာ တစ်ခုတည်း — နောက်ဆုံး node မှာ **`isEnd`** **စစ်စရာ မလို**ပါ။ လမ်းကြောင်း ရှိ**ရုံ**နဲ့ "ဒီ prefix နဲ့ စတာ ရှိတယ်" လို့ ဆိုနိုင်ပါတယ်။

```
Trie: ["cat", "car", "card", "dog"]

startsWith("ca")?  root→c→a လမ်းကြောင်း ရှိ → true ✓
startsWith("do")?  root→d→o လမ်းကြောင်း ရှိ → true ✓
startsWith("ba")?  root→'b' မရှိ → false ✗
```

```
// prefix နဲ့ စတဲ့ စကားလုံး ရှိမရှိ
boolean startsWith(String prefix) {
    return findNode(prefix) != null;      // လမ်းကြောင်း ရှိရုံ (isEnd မစစ်)
}
```

ဒါက ဘာကြောင့် Hash Map ထက် ကောင်းသလဲ: Hash Map နဲ့ " `ca` နဲ့ စတာ ရှိလား" မေးရင် key **အကုန်လုံး** (n လုံး) prefix စစ်ရ ($O(n \cdot L)$)။ Trie နဲ့ဆိုရင် prefix အရှည် L အတိုင်းသာ ဆင်းရလို့ $O(L)$ — Trie ထဲ စကားလုံး ဘယ်နှသိန်း ရှိရှိ မမူတည်ပါ။

Memory Trade-off — Trie က နေရာ ဘယ်လောက် ကုန်လဲ

Trie က မြန်ပေမယ့် **memory** အတွက် ပေးဆပ်ရပါတယ်။ node တစ်ခုစီမှာ children pointer (Java မှာ `TrieNode[26]` — ၂၆ ကွက်) ထားရလို့ — စကားလုံး အနည်းငယ်သာ ထည့်ထားရင် ကွက်အများစု `null` ဖြစ်ပြီး နေရာ အလဟဿ ဖြစ်နိုင်ပါတယ်။

	Hash Map (Set)	Trie
Exact search ("word ရှိလား")	$O(L)$ မြန်	$O(L)$ မြန်
Prefix search ("ca နဲ့စတာ ရှိလား")	$O(n \cdot L)$ နှေး	$O(L)$ မြန်
Memory	စကားလုံး အရွယ်အတိုင်း	များနိုင် (node/pointer အများ)
Sorted order	မရ (random)	ရ (စာလုံးအလိုက် စီပြီးသား)

> **ဘယ်ဟာ ဘယ်အချိန် သုံးမလဲ:** "exact ရှိမရှိ" သာ လိုရင် → **Hash Map** (ရိုးရှင်း၊ memory သက်သာ)။ "**prefix** နဲ့ရှာ / autocomplete / sorted order" လိုရင် → **Trie**။ ဒါက Trie ကို တကူးတက သုံးရတဲ့ အဓိက အကြောင်းပါ — prefix operation က Trie ရဲ့ ထူးခြားချက် ဖြစ်သည်။

> **Memory ချွေတာနည်း:** စကားလုံး များပြားလာရင် node တွေ မြောက်မြားလာတာကို လျှော့ဖို့ **Radix Tree (Compressed Trie)** — "ကွဲစရာ မရှိတဲ့ string အပိုင်း" ကို node တစ်ခုတည်းမှာ ပေါင်းသိမ်းတဲ့ နည်း — ရှိပါတယ်။ ဒီစာအုပ်မှာ အသေးစိတ် မလေ့လာပေမယ့်၊ "Trie ကို memory အတွက် optimize လုပ်တဲ့ နည်း ရှိတယ်" ဆိုတာ မှတ်ထားပါ။

Real-world Examples

Trie ဟာ "**prefix** နဲ့ရှာ / စာလုံးအလိုက် စီထား" လိုတဲ့ နေရာတိုင်းမှာ တွေ့ရပါတယ် —

- **Autocomplete** — search box / ဖုန်း keyboard မှာ `app` ရိုက်တာနဲ့ "apple, application, apply..." prefix ကိုက်တာတွေ ချက်ချင်း ပြ။
- **Search Suggestion** — Google search bar က စာလုံး တစ်လုံး ရိုက်တိုင်း အကြံပြုချက်တွေ ထွက်လာတာ။
- **Dictionary App** — စာလုံး ရှိမရှိ စစ်ခြင်း၊ "ဒီ prefix နဲ့စတဲ့ စကားလုံး အကုန်" စာရင်း ထုတ်ခြင်း။
- **Spell Checker** — စာလုံး Trie ထဲ ရှိမရှိ စစ်၊ မရှိရင် prefix တူတာတွေနဲ့ ပြင်ဆင်ချက် အကြံပြု။
- **Command / IP Route Matching** — terminal command auto-complete၊ router က IP address ကို "longest prefix match" နဲ့ လမ်းကြောင်း ရွေး။

ဒီ pattern တွေ အကုန်လုံး **"စာလုံး အစ ပိုင်း (prefix) တူတာတွေကို အမြန် စုစည်း/ရှာ"** ဆိုတဲ့ အခြေအနေပါ — Trie က prefix ကို မျှသုံးထားလို့ ဒီအလုပ်ကို သဘာဝကျကျ လုပ်ပေးနိုင်တာ ဖြစ်ပါတယ်။

Questions

Trie ကို လက်တွေ့ ပြဿနာ ၄ ခုနဲ့ ချိတ်ဆက် လေ့လာကြည့်ရအောင်။ ပြဿနာတိုင်းရဲ့ သော့ချက်က — **စကားလုံးတွေကို စာလုံးအလိုက် node ပြောင်းပြီး prefix မျှသုံးတဲ့** Trie structure ကို အသုံးချတတ်ဖို့ ဖြစ်ပါတယ်။

၁။ Implement Trie (Prefix Tree)

`insert` , `search` , `startsWith` ၃ ခု ပါတဲ့ Trie class ကို တည်ဆောက်ပါ။

Example 1:

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");     // true   (အပြည့်အစုံ ရှိ)
trie.search("app");       // false  (လမ်းကြောင်း ရှိပေမယ့် isEnd မဟုတ်)
trie.startsWith("app");   // true   (prefix ရှိ)
trie.insert("app");
trie.search("app");       // true   (အခု ထည့်ပြီးပြီ)
```

ရှင်းလင်းချက်

ဒါက အပေါ်က **insert / search / startsWith** ၃ ခုကို တစ်စုတည်း စုစည်းတာ ဖြစ်ပါတယ်။ သော့ချက် ၂ ချက် —

- node တစ်ခုစီမှာ `children` (နောက်စာလုံးတွေ) နဲ့ `isEnd` (ဆုံးမှတ်လား) ၂ ခု ထား။
- `search` က **`isEnd` စစ်**သည်၊ `startsWith` က **လမ်းကြောင်း ရှိရုံ** စစ်သည် — ဒါက ၂ ခုရဲ့ တစ်ခုတည်းသော ကွာခြားချက်။

> **Time Complexity:** insert/search/startsWith အားလုံး $O(L)$ - စကားလုံး/prefix အရှည်။
> **Space Complexity:** insert အတွက် $O(L)$ (node အသစ် အများဆုံး L ခု)။

Java Solution

```java
class Trie {
    private TrieNode root = new TrieNode();

    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEnd = false;
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int i = c - 'a';
            if (node.children[i] == null) node.children[i] = new TrieNode();
            node = node.children[i];
        }
        node.isEnd = true;
    }
```

```java
    public boolean search(String word) {
        TrieNode node = find(word);
        return node != null && node.isEnd;       // ဆုံးမှတ် ဖြစ်မှ
    }

    public boolean startsWith(String prefix) {
        return find(prefix) != null;              // လမ်းကြောင်း ရှိရုံ
    }

    private TrieNode find(String s) {
        TrieNode node = root;
        for (char c : s.toCharArray()) {
            int i = c - 'a';
            if (node.children[i] == null) return null;
            node = node.children[i];
        }
        return node;
    }
}
```

၂။ Word Search II

`m × n` စာလုံး grid `board` နဲ့ စကားလုံး စာရင်း `words` ပေးထားသည်။ grid ထဲ — အနီးကပ် (အပေါ်/အောက်/ဘယ်/ညာ) ကွက်တွေ ဆက်တိုက် ဖတ်ပြီး ဖွဲ့လို့ ရတဲ့ စကားလုံး **အကုန်** ကို ပြန်ပါ။ ကွက်တစ်ခုကို စကားလုံး တစ်လုံးအတွင်း **တစ်ခါသာ** သုံးရသည်။

Example 1:

```
Input: board = [["o","a","a","n"],
                ["e","t","a","e"],
                ["i","h","k","r"],
                ["i","f","l","v"]]
       words = ["oath","pea","eat","rain"]
Output: ["oath","eat"]
   ("oath": o→a→t→h လမ်းကြောင်း ရှိ၊  "eat": e→a→t ရှိ၊  "pea"/"rain" မရှိ)
```

ရှင်းလင်းချက်

စကားလုံး တစ်လုံးချင်း grid မှာ ရှာ (DFS) လုပ်ရင် — `words` များလာ၊ grid ကြီးလာတဲ့အခါ **ထပ်ခါတလဲလဲ ဖြတ်**ရလို့ နှေးပါတယ်။ သော့ချက်က — `words` အကုန်လုံးကို **Trie** ထဲ ထည့်ပြီး၊ grid ကို **တစ်ခါတည်း** DFS ဖြတ်ရင်း Trie လမ်းကြောင်းကို တပြိုင်တည်း လိုက်တာ ဖြစ်ပါတယ်။

- grid ကွက်တစ်ခုစီကနေ DFS စ — လက်ရှိ စာလုံးက Trie node မှာ child ရှိမှ **ဆက်ဖြတ်** (မရှိရင် ချက်ချင်း ရပ် = **pruning**)။
- Trie node မှာ `isEnd` တွေ့ရင် → အဲ့ စကားလုံး တွေ့ပြီ → result ထဲ ထည့်။
- ကွက်ကို "သုံးပြီ" အမှတ်အသား လုပ် (Choose) → ၄ ဘက် ဆက်ဖြတ် (Explore) → ပြန်ဖြုတ် (Undo — အခန်း ၁၆ backtracking ပုံစံ)။

Trie ကြောင့် — "ဒီ prefix နဲ့ စတဲ့ စကားလုံး တစ်လုံးမှ မရှိ" ဆိုတာ သိတာနဲ့ grid လမ်းကြောင်းကို ချက်ချင်း ဖြတ်ပစ်လို့၊ စကားလုံး တစ်လုံးချင်း သီးခြား ရှာတာထက် များစွာ မြန်ပါတယ်။

> **Time Complexity:** အကြမ်းဖျင်း $O(M \cdot 4 \cdot 3^L)$ - grid ကွက် M ခု၊ DFS အမြင့် L (စကားလုံး အရှည်ဆုံး)။ Trie က prune လုပ်လို့ လက်တွေ့မှာ ပိုမြန်။
> **Space Complexity:** $O(K)$ - Trie size (words စုစုပေါင်း စာလုံး အရေအတွက်)။

Java Solution

```java
class Solution {
    private List<String> result = new ArrayList<>();

    public List<String> findWords(char[][] board, String[] words) {
        TrieNode root = buildTrie(words);                // words → Trie
        for (int r = 0; r < board.length; r++) {
            for (int c = 0; c < board[0].length; c++) {
                dfs(board, r, c, root);                   // ကွက်တိုင်းကနေ စ
            }
        }
        return result;
    }

    private void dfs(char[][] board, int r, int c, TrieNode node) {
        if (r < 0 || r >= board.length || c < 0 || c >= board[0].length) return;
        char ch = board[r][c];
        if (ch == '#' || node.children[ch - 'a'] == null) return;  // သုံးပြီး / prune
        node = node.children[ch - 'a'];
        if (node.word != null) {                         // isEnd → စကားလုံး တွေ့ပြီ
            result.add(node.word);
            node.word = null;                            // ထပ် မထည့်အောင်
        }
        board[r][c] = '#';                               // 1. Choose (သုံးပြီ မှတ်)
        dfs(board, r + 1, c, node);                      // 2. Explore (၄ ဘက်)
        dfs(board, r - 1, c, node);
        dfs(board, r, c + 1, node);
        dfs(board, r, c - 1, node);
        board[r][c] = ch;                                // 3. Undo (ပြန်ဖြုတ်)
    }

    private TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for (String w : words) {
            TrieNode node = root;
            for (char c : w.toCharArray()) {
                int i = c - 'a';
                if (node.children[i] == null) node.children[i] = new TrieNode();
                node = node.children[i];
            }
            node.word = w;                               // ဆုံးမှတ်မှာ စကားလုံး သိမ်း
        }
        return root;
    }

    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        String word = null;                              // isEnd အစား စကားလုံး သိမ်း
    }
}
```

၃။ Replace Words (Shortest Root)

အမြစ်စကားလုံး (root) စာရင်း `dictionary` နဲ့ ဝါကျ `sentence` ပေးထားသည်။ ဝါကျထဲက စကားလုံး တစ်ခုစီအတွက် — အဲ့စကားလုံးကို prefix အဖြစ် ထားနိုင်တဲ့ **အတိုဆုံး root** နဲ့ အစားထိုးပါ။ root မရှိရင် မူရင်းအတိုင်း ထား။

Example 1:

```
Input: dictionary = ["cat","bat","rat"]
       sentence = "the cattle was rattled by the battery"
Output: "the cat was rat by the bat"
   ("cattle"→"cat", "rattled"→"rat", "battery"→"bat")
```

ရှင်းလင်းချက်

root တွေကို Trie ထဲ ထည့်ထားပြီး — ဝါကျက စကားလုံး တစ်ခုစီကို Trie ထဲ **စာလုံးချင်း ဆင်းရင်း** `isEnd` **တွေ့တာနဲ့ ရပ်** ရပါတယ်။ `isEnd` ပထမဆုံး တွေ့တဲ့ နေရာက **အတိုဆုံး root** ဖြစ်ပါတယ် (တိုတိုက အရင် ဆုံးတာမို့)။

- စကားလုံးကို Trie မှာ စာလုံးချင်း ဆင်း။
- `isEnd` တွေ့ → အဲ့အထိ prefix က root → အစားထိုး။
- လမ်းကြောင်း ပြတ် (child မရှိ) သို့ စကားလုံး ကုန်တဲ့အထိ `isEnd` မတွေ့ → မူရင်း ထား။

> **Time Complexity:** $O(M+N)$ - M = dictionary စာလုံး စုစုပေါင်း (root တွေ Trie ထဲ insert)၊ N = ဝါကျ စာလုံး စုစုပေါင်း (lookup)။
> **Space Complexity:** $O(M)$ - dictionary စုစုပေါင်း စာလုံး အရေအတွက် (Trie size)။

Java Solution

```java
class Solution {
    public String replaceWords(List<String> dictionary, String sentence) {
        TrieNode root = new TrieNode();
        for (String w : dictionary) insert(root, w);     // root အကုန် ထည့်

        StringBuilder sb = new StringBuilder();
        for (String word : sentence.split(" ")) {
            if (sb.length() > 0) sb.append(" ");
            sb.append(shortestRoot(root, word));         // တစ်လုံးချင်း အစားထိုး
        }
        return sb.toString();
    }

    private String shortestRoot(TrieNode root, String word) {
        TrieNode node = root;
        StringBuilder prefix = new StringBuilder();
        for (char c : word.toCharArray()) {
            int i = c - 'a';
            if (node.children[i] == null) return word;   // လမ်းကြောင်း ပြတ် → မူရင်း
            prefix.append(c);
            node = node.children[i];
            if (node.isEnd) return prefix.toString();    // ပထမ root → အတိုဆုံး
```

```
        }
        return word;                                    // root မဆုံး → မူရင်း
    }

    private void insert(TrieNode root, String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int i = c - 'a';
            if (node.children[i] == null) node.children[i] = new TrieNode();
            node = node.children[i];
        }
        node.isEnd = true;
    }

    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEnd = false;
    }
}
```

၄။ Longest Common Prefix

စကားလုံး array `strs` ပေးထားသည်။ အကုန်လုံးမှာ **တူညီတဲ့ အရှည်ဆုံး ရှေ့ဆုံး အပိုင်း (longest common prefix)** ကို ပြန်ပါ။ မရှိရင် `""` ပြန်ပါ။

Example 1:

```
Input: strs = ["flower", "flow", "flight"]
Output: "fl"
   (၃ လုံးလုံး "fl" နဲ့ စ၊ ဒါပေမယ့် "flo" က "flight" မှာ မရှိ)
```

ရှင်းလင်းချက်

ဒါကို Trie idea နဲ့ စဉ်းစားနိုင်ပါတယ် — စကားလုံး အကုန် Trie ထဲ ထည့်ရင်၊ common prefix ဆိုတာ root ကနေ "**လမ်းခွဲ မရှိဘဲ (child ၁ ခုတည်း) ဆက်ဆင်းနိုင်တဲ့**" အပိုင်း ဖြစ်ပါတယ်။ လမ်းခွဲ (≥ 2 children) တွေ့တာ သို့ စကားလုံး တစ်ခု ဆုံး (`isEnd`) တာနဲ့ common prefix ရပ်ပါတယ်။

> **ပိုရိုးတဲ့နည်း:** ဒီပြဿနာအတွက် Trie တကယ် မဆောက်ဘဲ — ပထမ စကားလုံးကို စံထား၊ ကျန်တာနဲ့ စာလုံးချင်း တိုက်စစ်ပြီး ကွဲတဲ့ နေရာမှာ ဖြတ်ရင် ပိုလွယ်ပါတယ် (idea အတူတူ — "လမ်းခွဲ မဖြစ်မချင်း ဆက်သွား")။ အောက်မှာ ဒီနည်းကို ပြထားသည်။

> **Time Complexity:** $O(N)$ - စာလုံး စုစုပေါင်း (အဆိုးဆုံး အကုန် တိုက်စစ်)။
> **Space Complexity:** $O(1)$ - extra space မလို။

Java Solution

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs.length == 0) return "";
```

```
        // ပထမ စကားလုံးကို စံထား၊ စာလုံး (column) တစ်လုံးချင်း တိုက်စစ်
        for (int i = 0; i < strs[0].length(); i++) {
            char c = strs[0].charAt(i);
            for (int j = 1; j < strs.length; j++) {
                // အဲ့ စကားလုံး ကုန်ပြီ သို့ စာလုံး မကိုက် → အဲ့အထိ ပြန်
                if (i == strs[j].length() || strs[j].charAt(i) != c)
                    return strs[0].substring(0, i);      // final substring တစ်ခုတည်း
            }
        }
        return strs[0];
    }
}
```

အခန်း ၁၆ - Backtracking

အခန်း ၁၃–၁၅ မှာ tree တွေကို **recursion** နဲ့ DFS လေ့လာခဲ့ပါတယ် — node တစ်ခုကို ရောက်ရင် "ဘယ်ဘက် ဆင်း၊ ပြန်တက်၊ ညာဘက် ဆင်း" လုပ်တာ။ အခု ဒီ idea ကို tree မရှိဘဲ — **ဖြစ်နိုင်ခြေ အကုန်လုံးကို စနစ်တကျ စမ်းကြည့်ရတဲ့** ပြဿနာတွေမှာ သုံးကြည့်ပါမယ်။

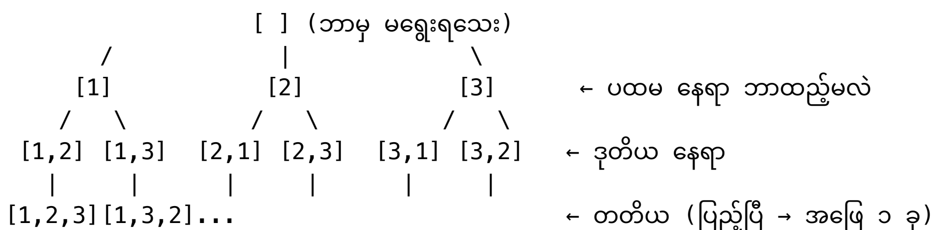
ဥပမာ — "ဂဏန်း `[1,2,3]` ကို ဖြစ်နိုင်တဲ့ အစီအစဉ် (permutation) **အကုန်** ထုတ်ပါ" ဆိုရင် `[1,2,3]` , `[1,3,2]` , `[2,1,3]` … ၆ မျိုး ရှိပါတယ်။ ဒါမျိုး "ရွေးချယ်စရာတွေ ဆင့်ကဲ ပေါင်းပြီး **valid အဖြေ အကုန်** ရှာ" ရတဲ့ ပြဿနာတွေကို **brute force** (အကုန် စမ်း) နဲ့ ဖြေနိုင်ပေမယ့် — ရိုးရိုး brute force က မလိုတဲ့ လမ်းကြောင်းတွေပါ စမ်းမိလို့ အချိန် ပိုကုန်ပါတယ်။

Backtracking ဆိုတာ — brute force ထဲက **smart search pattern** ဖြစ်ပါတယ်။ "ရွေး (Try/Choose) → စမ်း (Explore) → မရရင် ပြန်ဖြုတ် (Undo)" သုံးဆင့်ကို ထပ်ခါတလဲလဲ လုပ်ပြီး၊ **မဖြစ်နိုင်တော့တဲ့ လမ်းကြောင်းတွေကို စောစော ဖြတ်ပစ် (pruning)** လို့ ရတာ ဖြစ်ပါတယ်။ ဒီအခန်းမှာ backtracking ရဲ့ "Choose–Explore–Undo" pattern ကို လေ့လာပြီး classic ပြဿနာ ၅ ခု ဖြေကြည့်ပါမယ်။

Backtracking ဆိုတာ ဘာလဲ — Choose, Explore, Undo

Backtracking ကို **"ဆုံးဖြတ်ချက် သစ်ပင် (decision tree)"** အဖြစ် မြင်ရင် အလွယ်ဆုံးပါ။ အဆင့်တိုင်းမှာ "ရွေးစရာ" တွေ ရှိပြီး၊ တစ်ခုစီကို လမ်းခွဲ (branch) တစ်ခုအဖြစ် ဆင်းကြည့်တာ — ဆင်းလို့ မရတော့ရင် ပြန်တက်ပြီး တစ်ခြား လမ်းခွဲ စမ်းတာ ဖြစ်ပါတယ်။

`[1,2,3]` ရဲ့ permutation ရှာတဲ့ decision tree —



သစ်ပင်ထဲ လမ်းကြောင်း တစ်ခုစီကို ဆင်းတဲ့အခါ **သုံးဆင့်** ထပ်ခါ လုပ်သည် —

1. **Choose (ရွေး):** ရွေးစရာ တစ်ခုကို ယူ၊ လက်ရှိ အဖြေ (path) ထဲ ထည့်။
2. **Explore (စမ်း):** အဲ့ ရွေးချယ်မှုနဲ့ **ဆက်ပြီး** အောက်ကို recursion ဆင်း။
3. **Undo (ပြန်ဖြုတ်):** recursion ပြန်တက်လာရင် — အဆင့် ၁ မှာ ထည့်ထားတာကို **ပြန်ထုတ်** (path ကို မူလအတိုင်း ပြန်ထား)၊ နောက် ရွေးစရာ စမ်းဖို့။

ဒီ **Undo** က backtracking ရဲ့ အသက်ပါ — "ဒီ လမ်းကြောင်း စမ်းပြီးပြီ၊ အခု သန့်ရှင်းပြန်လုပ်ပြီး နောက်တစ်ခု စမ်းမယ်" ဆိုတဲ့ သဘောဖြစ်သည်။

```
path = []
choose 1 → path = [1]
    explore... (1 နဲ့ ဆက်တာ အကုန် စမ်း)
undo 1   → path = []       ← ပြန်သန့်
choose 2 → path = [2]
    explore...
undo 2   → path = []
...
```

ပုံသေ Pattern (Template)

Backtracking ပြဿနာ အများစုက **Choose** → **Explore** → **Undo** ဆိုတဲ့ Pattern တစ်ခုတည်းကို အခြေခံထားတာပါ။ Problem ပေါ်မူတည်ပြီး အသေးစိတ် logic ပဲ ပြောင်းသွားတာ ဖြစ်ပါတယ်။

ပထမ အောက်က code က **pseudocode idea** ဖြစ်ပြီး Java code မဟုတ်ပါ — structure ကို မြင်ဖို့သာ။

```
backtrack(path):

    if လက်ရှိအခြေအနေက အဖြေရပြီး:
            result.add(path ကို copy ကူး)
            return
    for ရွေးစရာ တစ်ခုစီ:

        if မရွေးသင့်ရင်:
            continue

        path ထဲ ထည့်        // 1. Choose

        backtrack(path)      // 2. Explore

        path ထဲက ပြန်ဖျက်    // 3. Undo
```

အဲ့ဒီ idea ကို Java မှာ ရေးကြည့်ရင် ဒီလို ဖြစ်သည် — ဥပမာ အနေနဲ့ **length `n` ရဲ့ binary string အကုန်** (`000` , `001` , `010` , ...) ထုတ်တဲ့ ပြောင်းဖြတ်ကြည့်ပြီး Choose → Explore → Undo ကို ကြည့်ပါမယ်။ **Comment ၃ ခု** က backtracking ရဲ့ အသက်ဖြစ်ပြီး၊ ဒီအခန်းရဲ့ ပြဿနာ အကုန်လုံးက ဒီ structure တူထည်း သုံးတာပါ။

```java
import java.util.*;

class Main {
    static List<String> result = new ArrayList<>();

    public static void main(String[] args) {
        backtrack(3, new StringBuilder());
        System.out.println(result);
        // [000, 001, 010, 011, 100, 101, 110, 111]   ← 2³ = ၈ မျိုး
    }

    static void backtrack(int n, StringBuilder path) {
```

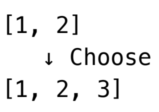
```
        if (path.length() == n) {                // base case — n လုံး ပြည့်ပြီ
            result.add(path.toString());         // copy ကူးပြီး မှတ်
            return;
        }
        for (char c : new char[]{'0', '1'}) {    // ရွေးစရာ ၂ ခု (0 သို့ 1)
            path.append(c);                      // 1. Choose
            backtrack(n, path);                  // 2. Explore
            path.deleteCharAt(path.length()-1);  // 3. Undo
        }
    }
}
```

> `path.toString()` **(copy ကူးခြင်း):** `result` ထဲ `path` ကို တိုက်ရိုက် မထည့်ရပါ — `path` က တစ်ခုတည်း ပြန်ပြင်နေတဲ့ object ဖြစ်လို့၊ copy မကူးရင် နောက်ပြီးတာ `result` ထဲက အဖြေတွေပါ ပြောင်းသွားပါမယ်။ List သုံးရင် `new ArrayList<>(path)` ၊ String သုံးရင် `.toString()` နဲ့ copy ကူးရပါတယ်။ **undo လိုတိုင်း copy လည်း လိုပါ**။

Undo ဘာကြောင့် လိုတာလဲ

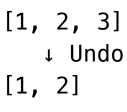
ဥပမာ `path = [1, 2]` ရှိနေတယ်ဆိုပါစို့။

`3` ကို ရွေးလိုက်ရင်



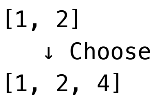
```
[1, 2]
   ↓ Choose
[1, 2, 3]
```

အဲဒီ branch ကို စမ်းပြီးသွားရင် `3` ကို ပြန်ဖျက်ရပါတယ်။



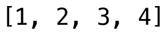
```
[1, 2, 3]
   ↓ Undo
[1, 2]
```

ဒါမှ နောက်ထပ် `4` ကို ဆက်စမ်းနိုင်မှာ ဖြစ်ပါတယ်။



```
[1, 2]
   ↓ Choose
[1, 2, 4]
```

Undo မလုပ်ရင်



```
[1, 2, 3, 4]
```

လိုမျိုး ဖြစ်သွားပြီး branch အဟောင်းရဲ့ data တွေ ကျန်နေတဲ့အတွက် အဖြေမှားသွားပါလိမ့်မယ်။

> **`path.removeLast()` (Undo) က Backtracking ရဲ့ အဓိက အစိတ်အပိုင်း ဖြစ်ပါတယ်။** Choose လုပ်ပြီးတိုင်း Explore လုပ်မယ်၊ Explore ပြီးတိုင်း Undo လုပ်မယ်။ ဒီ Pattern ကို Backtracking ပြဿနာ အများစုမှာ အမြဲတွေ့ရပါတယ်။

> **Backtracking vs Recursion:** Backtracking က **Recursion ကို အသုံးပြုတဲ့ Algorithm** တစ်မျိုး ဖြစ်ပါတယ်။ သာမန် Recursion (ဥပမာ factorial, tree traversal) က လမ်းကြောင်းတစ်ခုကိုပဲ ဆင်းသွားတာ များပါတယ်။ Backtracking ကတော့ **လမ်းကြောင်း အမျိုးမျိုးကို စမ်း၊ မရရင် ပြန်ဖြုတ် (Undo)၊ ရတဲ့အဖြေ အားလုံးကို စု** တာ ဖြစ်ပါတယ်။ ဒါကြောင့် **Choose → Explore → Undo** ဆိုတဲ့ Pattern က Backtracking ရဲ့ အမှတ်အသား ဖြစ်ပါတယ်။

Pruning — မလိုတဲ့ လမ်းကြောင်း ဖြတ်ခြင်း

Brute force က ဖြစ်နိုင်ခြေ **အကုန်** စမ်းတာမို့ နှေးပါတယ်။ Backtracking ရဲ့ အားသာချက်က — လမ်းခွဲ တစ်ခုကို ဆင်းတဲ့အခါ "ဒီကနေ ဆက်သွားရင် valid အဖြေ **ဘယ်တော့မှ မဖြစ်နိုင်တော့ဘူး**" ဆိုတာ သိရင် — အဲ့ subtree တစ်ခုလုံးကို **ချက်ချင်း ဖြတ်ပစ် (prune)** လို့ ရတာ ဖြစ်ပါတယ်။

```
ဥပမာ — "ပေါင်းလဒ် 5 ဖြစ်အောင် ဂဏန်း ရွေး" မှာ [2,4,...] ရွေးပြီး sum=6 ဖြစ်သွားရင်
→ 5 ထက် ကျော်သွားပြီ → ဒီ subtree ဆက်ဆင်းစရာ မလို → prune
```

Pruning က decision tree ရဲ့ ကြီးမားတဲ့ အကိုင်းတွေကို ဖြတ်ချလို့ — အမှန်တကယ် စမ်းရတဲ့ လမ်းကြောင်း အရေအတွက် သိသိသာသာ လျော့ကျသွားပါတယ်။ ဒါက backtracking ကို "ရိုးရိုး brute force" ထက် မြန်စေတဲ့ အဓိက အချက်ပါ။

> **Complexity:** ဒါတွေက ဖြစ်နိုင်ခြေ အကုန် ထုတ်ရတဲ့ ပြဿနာတွေမို့ — အဆိုးဆုံး complexity က မကြာခဏ **exponential** ($O(2^n)$, $O(n!)$ စသည်) ဖြစ်ပါတယ်။ Pruning က လက်တွေ့မှာ မြန်အောင် ကူပေမယ့် အဆိုးဆုံး bound ကို မပြောင်းနိုင်တာ များပါတယ်။

Real-world Examples

Backtracking ဟာ "**valid ဖြစ်နိုင်ခြေ အကုန်** ရှာ / **constraint ပြည့်တဲ့ အစီအစဉ်** ရှာ" လိုတဲ့ နေရာတိုင်းမှာ တွေ့ရပါတယ် —

- **Permission / Role Combinations** — user တစ်ဦးကို ပေးနိုင်တဲ့ permission set အကုန် (သို့) rule ပြည့်တဲ့ combination တွေ ထုတ်ခြင်း။
- **Form Rules Combination** — field တွေရဲ့ "ဒါ ရွေးရင် ဟို မရွေးရ" မျိုး constraint ပြည့်တဲ့ valid form အခြေအနေ အကုန် ရှာ။
- **Search All Valid Configurations** — feature flag, settings, pricing rule စတဲ့ configuration တွေထဲက rule ပြည့်တဲ့ အခြေအနေ အကုန် ရှာ။

- **Scheduling Options** — အချိန်/resource မထပ်အောင် (constraint) event တွေ စီစဉ်နိုင်တဲ့ နည်းလမ်း အကုန် ရှာ။
- **Puzzle Solver** — **Sudoku** (ကွက်တိုင်း ၁–၉ ဖြည့်၊ row/column/box မထပ်), **N-Queens** (queen တွေ မတိုက်အောင် ချ), crossword — အကုန် "ရွေး → စမ်း → ဖောက်ရင် ပြန်ဖြုတ်" pattern။
- **Maze / Path Finding** — လမ်းကြောင်းတိုင်း စမ်း၊ ပိတ်နေရင် ပြန်ဆုတ်ပြီး တစ်ခြားလမ်း စမ်း။

ဒီ pattern တွေ အကုန်လုံး တူပါတယ် — **"ရွေးချယ်စရာ ဆင့်ကဲ ပေါင်းရင်း၊ constraint ဖောက်ရင် ပြန်ဆုတ်၊ ပြည့်ရင် အဖြေ မှတ်"**။ ဒါက backtracking ရဲ့ အသုံးချပုံ အနှစ်ချုပ်ပါ။

Questions

Backtracking ကို ပြဿနာ ၅ ခုနဲ့ လေ့လာကြည့်ရအောင်။ ပြဿနာတိုင်းရဲ့ သော့ချက်က — **Choose** → **Explore** → **Undo** pattern ကို မှန်မှန် တပ်ဆင်တတ်ဖို့ နဲ့ base case / pruning ကို ရှာတတ်ဖို့ ဖြစ်ပါတယ်။

၁။ Subsets

ထပ်တူ မရှိတဲ့ ကိန်း array `nums` ပေးထားသည်။ ဖြစ်နိုင်တဲ့ **subset (အစုခွဲ) အကုန်** (power set) ကို ပြန်ပါ။

Example 1:

```
Input: nums = [1,2,3]
Output: [[], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3]]
   (subset ၈ ခု = 2³)
```

ရှင်းလင်းချက်

subset တစ်ခုစီအတွက် — element တစ်ခုစီကို **"ထည့်မလား / မထည့်ဘူးလား"** ဆုံးဖြတ်ရပါတယ်။ ဒါက index တစ်ခုစီကနေ "ထည့်တဲ့ branch" နဲ့ "မထည့်တဲ့ branch" ၂ ခွဲ ဆင်းတဲ့ decision tree ဖြစ်သည်။

ပိုလွယ်တဲ့ မြင်ပုံ — `start` index ကနေ စပြီး၊ နောက်က element တွေ တစ်ခုစီကို path ထဲ ထည့် (Choose) → ဆက်ဆင်း (Explore) → ပြန်ဖြုတ် (Undo)။ node တိုင်းက valid subset ဖြစ်လို့ — ဆင်းတိုင်း မှတ်ပါတယ် (base case သီးသန့် မလို)။

> **Time Complexity:** $O(n \cdot 2^n)$ - subset 2^n ခု၊ တစ်ခုစီ copy ယူ $O(n)$။
> **Space Complexity:** $O(n)$ - recursion depth + path (output မရေတွက်)။

Java Solution

```
class Solution {
```

```
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] nums, int start, List<Integer> path,
                           List<List<Integer>> result) {
        result.add(new ArrayList<>(path));      // node တိုင်း valid subset → မှတ်
        for (int i = start; i < nums.length; i++) {
            path.add(nums[i]);                  // 1. Choose
            backtrack(nums, i + 1, path, result);  // 2. Explore (i+1 ကစ)
            path.remove(path.size() - 1);       // 3. Undo
        }
    }
}
```

၂။ Permutations

ထပ်တူ မရှိတဲ့ ကိန်း array `nums` ပေးထားသည်။ ဖြစ်နိုင်တဲ့ **အစီအစဉ် (permutation) အကုန်** ကို ပြန်ပါ။

Example 1:

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
   (permutation ၆ ခု = 3!)
```

ရှင်းလင်းချက်

Subset နဲ့ ကွာတာ — permutation မှာ **အစီအစဉ် အရေးကြီး**ပြီး element **အကုန်လုံး** သုံးရသည်။ ဒါကြောင့် "သုံးပြီးသား element ကို ထပ် မသုံးရ" ဆိုတဲ့ track လုပ်ဖို့ **`used[]`** (သို့ contains စစ်) လိုပါတယ်။

- **Base case:** path အရှည် = `nums` အရှည် → permutation ပြည့်ပြီ → မှတ်။
- element တစ်ခုစီ — သုံးပြီးသား မဟုတ်ရင် → Choose (mark used + path ထည့်) → Explore → Undo (unmark + path ဖြုတ်)။

> **Time Complexity:** $O(n \cdot n!)$ - permutation $n!$ ခု၊ တစ်ခုစီ copy $O(n)$။
> **Space Complexity:** $O(n)$ - recursion depth + used array။

Java Solution

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, new boolean[nums.length], new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] nums, boolean[] used, List<Integer> path,
```

```
                              List<List<Integer>> result) {
        if (path.size() == nums.length) {          // base case — ပြည့်ပြီ
            result.add(new ArrayList<>(path));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (used[i]) continue;                 // pruning — သုံးပြီးသား ကျော်
            used[i] = true;  path.add(nums[i]);    // 1. Choose
            backtrack(nums, used, path, result);   // 2. Explore
            used[i] = false; path.remove(path.size() - 1);  // 3. Undo
        }
    }
}
```

၃။ Combinations

ကိန်း `n` နဲ့ `k` ပေးထားသည်။ `[1..n]` ထဲက ကိန်း **`k` လုံး** ရွေးတဲ့ combination **အကုန်** ကို ပြန်ပါ (အစီအစဉ် မရေး)။

Example 1:

```
Input: n = 4, k = 2
Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
   (combination ၆ ခု = C(4,2))
```

ရှင်းလင်းချက်

Permutation နဲ့ကွာတာ — combination မှာ **အစီအစဉ် မရေး** (`[1,2]` နဲ့ `[2,1]` တူ)။ ဒါကြောင့် Subsets လို **`start` index** ကို သုံးပြီး "နောက်ပြန် မလှည့်" အောင် လုပ်ရင်၊ ထပ်နေတာ အလိုအလျောက် ရှောင်ပါတယ်။

- **Base case:** path အရှည် = `k` → combination ပြည့်ပြီ → မှတ်။
- `start`ကစ၊ ကိန်းတစ်ခုစီ Choose → Explore (`i+1` ကစ) → Undo။

> **Pruning:** "ကျန် ကိန်း မလောက်တော့ရင်" ဆက်ဆင်းစရာ မလို — `for` loop ရဲ့ အဆုံးကို `n - (k - path.size()) + 1` အထိ ကန့်သတ်ထားရင် မလိုတဲ့ branch တွေ မစမ်းရတော့ပါ။

> **Time Complexity:** $O(k \cdot C(n,k))$။
> **Space Complexity:** $O(k)$ - recursion depth + path။

Java Solution

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(n, k, 1, new ArrayList<>(), result);
        return result;
    }
```

```
    private void backtrack(int n, int k, int start, List<Integer> path,
                           List<List<Integer>> result) {
        if (path.size() == k) {                      // base case — k လုံး ပြည့်
            result.add(new ArrayList<>(path));
            return;
        }
        int remaining = k - path.size();
        for (int i = start; i <= n - remaining + 1; i++) {
            path.add(i);                             // 1. Choose
            backtrack(n, k, i + 1, path, result);   // 2. Explore (i+1 → နောက်မပြန်)
            path.remove(path.size() - 1);           // 3. Undo
        }
    }
}
```

၄။ Combination Sum

ထပ်တူ မရှိတဲ့ ကိန်း array `candidates` နဲ့ ပစ်မှတ် `target` ပေးထားသည်။ ပေါင်းလဒ် `target` ဖြစ်တဲ့ combination **အကုန်** ကို ပြန်ပါ။ ကိန်း တစ်ခုကို **အကြိမ်ကြိမ် ပြန်သုံးလို့** ရသည်။

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
  (2+2+3 = 7,  7 = 7)
```

ရှင်းလင်းချက်

ဒါက Combinations နဲ့ ဆင်ပေမယ့် ၂ ချက် ကွာသည် — (၁) တစ်ခု ပြန်သုံးလို့ ရတာမို့ Explore မှာ `i+1` မဟုတ်ဘဲ `i` ကစ (ကိုယ့်ကို ပြန်ရွေးခွင့်)။ (၂) base case က count မဟုတ်ဘဲ **sum**။

- ပေါင်းရင်း `target` ကို လျှော့သွား (`remain`)။
- **Base case ၂ မျိုး:** `remain == 0` → အဖြေ မှတ်။ `remain < 0` → **pruning** (ကျော်ပြီ → ဖြတ်)။
- candidate တစ်ခုစီ Choose → Explore (`i` ကစ — ပြန်သုံးခွင့်) → Undo။

> **Pruning ၏ အရေးပါပုံ:** `remain < 0` ဖြစ်တာနဲ့ ဆက်မဆင်းတော့တာက decision tree ရဲ့ ကြီးမားတဲ့ အပိုင်းတွေ ဖြတ်ချပေးသည်။

> **Time Complexity:** အဆိုးဆုံး exponential (target/candidate ပေါ် မူတည်)။
> **Space Complexity:** $O(target/min(candidates))$ - recursion depth။

Java Solution

```
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(candidates, target, 0, new ArrayList<>(), result);
        return result;
```

```
    }

    private void backtrack(int[] cand, int remain, int start,
                           List<Integer> path, List<List<Integer>> result) {
        if (remain == 0) {                          // base case — target ပြည့်
            result.add(new ArrayList<>(path));
            return;
        }
        if (remain < 0) return;                     // pruning — ကျော်ပြီ
        for (int i = start; i < cand.length; i++) {
            path.add(cand[i]);                      // 1. Choose
            backtrack(cand, remain - cand[i], i, path, result);  // 2. Explore (i → ပြန်
သုံး)
            path.remove(path.size() - 1);           // 3. Undo
        }
    }
}
```

၅။ N-Queens

`n × n` chessboard ပေါ်မှာ queen `n` ကောင်ကို — တစ်ကောင်နဲ့ တစ်ကောင် **မတိုက်အောင်** (row, column, diagonal မထပ်အောင်) ချနိုင်တဲ့ နည်းလမ်း **ဘယ်နှမျိုး ရှိလဲ** (board အပြည့်အစုံ မဟုတ်ဘဲ အရေအတွက်ကိုသာ ပြန်ပေးမည် — count variant)။

Example 1:

```
Input: n = 4
Output: 2   (4×4 မှာ မတိုက်အောင် ချနိုင်တဲ့ ပုံစံ ၂ မျိုး)

  . Q . .       . . Q .
  . . . Q       Q . . .
  Q . . .       . . . Q
  . . Q .       . Q . .
```

ရှင်းလင်းချက်

ဒါက backtracking ရဲ့ classic ပြဿနာပါ။ queen တွေ row တစ်ခုစီမှာ တစ်ကောင်စီ ချရမယ် ဆိုတာ သေချာလို့ — **row တစ်ခုစီအတွက် "column ဘယ်ဟာမှာ ချမလဲ"** ကို ရွေးရတာ ဖြစ်သည်။

- **Base case:** row `n` ရောက် (queen `n` ကောင် အကုန် ချပြီး) → valid အဖြေ ၁ ခု။
- row တစ်ခုစီ — column တစ်ခုစီကို စမ်း၊ **safe ဖြစ်မှ** (column / diagonal ၂ ဘက် မထပ်) Choose → Explore (နောက် row) → Undo။
- **Pruning** — safe မဟုတ်တဲ့ column ကို ချက်ချင်း ကျော် (ဒါက N-Queens ကို brute force ထက် အများကြီး မြန်စေသည်)။

safe စစ်ဖို့ — column တစ်ခုစီ၊ diagonal ၂ မျိုး (`row+col` တူ = `/` ၊ `row-col` တူ = `\`) သုံးထားသလား မှတ်ထားရင် $O(1)$ နဲ့ စစ်လို့ ရသည်။

> **Time Complexity:** အဆိုးဆုံး $O(n!)$ (row တစ်ခုစီ column ရွေးစရာ လျော့သွား)။
> **Space Complexity:** $O(n)$ - column/diagonal set + recursion depth။

Java Solution

```java
class Solution {
    private int count = 0;

    public int totalNQueens(int n) {
        boolean[] cols = new boolean[n];          // column သုံးပြီးသားလား
        boolean[] diag1 = new boolean[2 * n];      // "\" : row - col + n
        boolean[] diag2 = new boolean[2 * n];      // "/" : row + col
        backtrack(0, n, cols, diag1, diag2);
        return count;
    }

    private void backtrack(int row, int n, boolean[] cols,
                           boolean[] diag1, boolean[] diag2) {
        if (row == n) { count++; return; }         // base case — n ကောင် ချပြီး
        for (int col = 0; col < n; col++) {
            int d1 = row - col + n, d2 = row + col;
            if (cols[col] || diag1[d1] || diag2[d2]) continue;  // pruning — တိုက်မယ်
            cols[col] = diag1[d1] = diag2[d2] = true;   // 1. Choose
            backtrack(row + 1, n, cols, diag1, diag2);  // 2. Explore (နောက် row)
            cols[col] = diag1[d1] = diag2[d2] = false;  // 3. Undo
        }
    }
}
```

> **Sudoku Solver ဆက်စပ်မှု:** Sudoku ဖြေတာလည်း N-Queens နဲ့ idea တူပါတယ် — ကွက်လွတ် တစ်ခုစီမှာ 1–9 တစ်ခုစီ စမ်း (Choose) → row/column/box မထပ်မှ ဆက် (Explore) → မရရင် ပြန်ဖြုတ် (Undo)။ "constraint ဖောက်ရင် ချက်ချင်း prune" တာက ၂ ခုလုံးရဲ့ အသက်ပါ။

အခန်း ၁၇ - Graphs

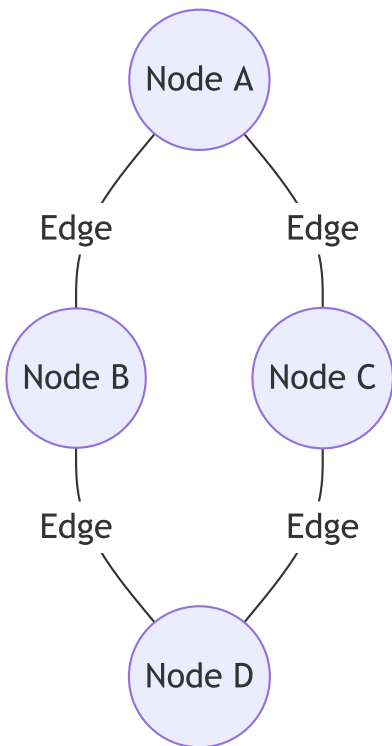
ရှေ့ အခန်းတွေ မှာ **tree** တွေကို လေ့လာခဲ့ပါတယ်။ node တွေ parent → child ဆက်စပ်ပြီး၊ အပေါ်ဆုံး မှာ root ၁ ခု၊ cycle မရှိတဲ့ structure ပါ။ ဒါပေမယ့် real-world relationship အများစုက tree လောက် သေသပ် မှု မရှိပါဘူး။ Facebook မှာ A က B ရဲ့ သူငယ်ချင်း၊ B က C ရဲ့၊ C က ပြန်ပြီး A ရဲ့ သူငယ်ချင်း ဖြစ်နေတာမျိုး ၊ မြို့တွေကြားက လမ်းတွေ က တစ်ခုကို တစ်ခုက မှီခို နေတာ မျိုးပေါ့။

"ဘယ်ဟာက ဘယ်ဟာနဲ့ ဆက်စပ်နေလဲ" ဆိုတဲ့ relationship တွေကို model လုပ်ဖို့ — **Graph** ကို သုံးပါတယ်။ တကယ်တော့ tree ဆိုတာ graph ရဲ့ **အထူး အမျိုးအစား** (cycle မရှိ၊ ဆက်စပ်နေတဲ့ graph) တစ်မျိုးပါပဲ။ Graph က တော့ Node အခြင်းခြင်း ဆက်သွယ်မှု တွေ ကို ရှိပါတယ်။

Graph ဆိုတာ ဘာလဲ — Vertex နဲ့ Edge

Array, linked list တွေက sequential (တစ်ခုပြီးတစ်ခု) ဖြစ်ပြီး၊ tree တွေက hierarchical (parent → child) ဖြစ်တယ်။ **Graph** ကတော့ **non-linear data structure** တစ်မျိုးပါ။ vertex တွေကြား relationship ကို ဘယ်လိုပုံစံနဲ့မဆို ကိုယ်စားပြုလို့ ရတယ်။ Graph ဆိုတာ အပိုင်း ၂ ခုနဲ့ ဖွဲ့စည်းထားတာ ဖြစ်ပါတယ်။

- **Vertex (node):** အချက်တစ်ခု — ဥပမာ user တစ်ဦး၊ မြို့တစ်မြို့၊ task တစ်ခု။
- **Edge:** vertex ၂ ခုကြား **ဆက်သွယ်မှု (connection)** — ဥပမာ "သူငယ်ချင်း ဖြစ်တယ်"၊ "လမ်း ရှိတယ်"၊ "မှီခိုတယ်"။



- **Node/Vertex:** A, B, C, D
- **Edge:** A–B, A–C, B–D, C–D

အပေါ်က ဥပမာမှာ — user ၄ ဦး (A, B, C, D) ရှိပြီး၊ မျဉ်း (edge) တစ်ခုစီက "သူငယ်ချင်း ဖြစ်တယ်" ကို ဆိုလိုပါတယ်။ Tree နဲ့ မတူတာက graph မှာ root မရှိတာပါ။

ဘယ် vertex က မဆို စလို့ ရ၊ cycle (A–B–D–C–A လို cycle) ရှိလို့ ရပါတယ်။

Tree vs Graph

အခန်း ၁၃ မှာ ပြောခဲ့တာ ပြန်သတိရစေချင်ပါတယ် — **tree ဆိုတာ graph ရဲ့ အထူး အမျိုးအစား** တစ်မျိုးပါ။ Tree တိုင်းဟာ graph တစ်ခု ဖြစ်ပေမယ့် graph တိုင်းက tree မဖြစ်ပါ။ တကယ်တော့ **Linked List, Tree, Heap အကုန်လုံးက graph ရဲ့ အထူးပုံစံ** တွေပါပဲ။

	Tree	**Graph**
Structure	hierarchy (parent → child)	ဘယ်လို ဆက်ထားလဲ ရ
Root	အပေါ်ဆုံး node ၁ ခု ရှိ	root မရှိ
Cycle	မရှိ	ရှိနိုင်
Parent	node တစ်ခုက parent အများကြီး မရ	parent ဆိုတာ မရှိ — ဘယ် vertex ဘယ်နှခုနဲ့ မဆို ဆက်
ဥပမာ	folder structure, DOM tree	social network, လမ်းပုံ, dependency

> **အခြေခံချုပ်** — graph မှာ rule နည်းလေ၊ ပို general လေ။ Tree က rule တွေ ပိုများပြီး (root ၁ ခု, cycle မရှိ, parent ၁ ခု) structure ပို သေသပ်တယ်။ Rule တွေ တိုးတာက graph ကို ပိုပြီး ထိန်းချုပ်လွယ်စေပြီး၊ algorithm တွေကိုလည်း ရိုးရှင်းစေတယ်။ ဥပမာ — tree မှာ cycle မရှိလို့ traversal က ပို ရိုးရှင်းတယ်။

Graph အမျိုးအစားများ

Directed vs Undirected

- **Undirected graph** — edge က **၂ ဖက်စလုံး** သွားလို့ ရ။ "A က B ရဲ့ သူငယ်ချင်း" ဆို B က လည်း A ရဲ့ သူငယ်ချင်း (Facebook friend)။
- **Directed graph** — edge မှာ **ဦးတည်ချက် (direction)** ရှိ။ "A က B ကို follow လုပ်တယ်" ဆို B က A ကို follow လုပ်တာ မဟုတ်သေး (Twitter/Instagram follow)။

Directed (follow)

A → B

Undirected (friend)

A — B

Undirected: A↔B ၂ ဖက် တူ။ Directed: A→B သာ၊ B→A မဟုတ်။

Weighted vs Unweighted (အလေးချိန် ရှိ / မရှိ)

- **Unweighted graph** — edge က "ဆက်နေသလား / မဆက်ဘူးလား" ပဲ ဆိုလို။
- **Weighted graph** — edge တစ်ခုစီမှာ **တန်ဖိုး (weight)** ရှိပါတယ်။ — ဥပမာ မြို့ ၂ မြို့ကြား **အကွာအဝေး**၊ network link ရဲ့ **latency**၊ လမ်းကြောင်း **ကုန်ကျစရိတ်**။

A –5– B, A –8– C, B –2– D, C –3– D

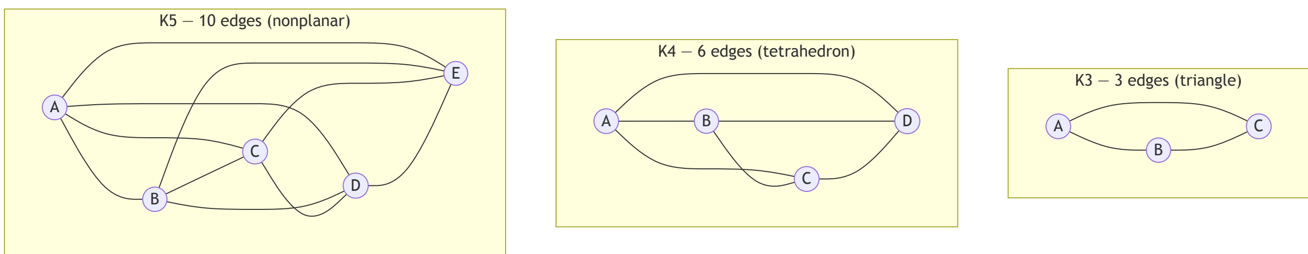
A→B = 5 km, B→D = 2 km … (edge weight = အကွာအဝေး)

> **Weighted graph** ကို အသုံးချတဲ့ shortest-path algorithm (Dijkstra စသည်) တွေကို အခန်း ၁၉ မှာ လေ့လာပါမယ်။

Complete vs Incomplete (အပြည့် / မပြည့်)

- **Complete graph (K_n)** — vertex တိုင်းက **ကျန် vertex အကုန်နဲ့** ဆက်နေတယ် (မဆက်တဲ့ အိမ်နီးချင်း တစ်ခုမျှ မရှိ)။ vertex `n` ခုဆိုရင် edge စုစုပေါင်း $n(n-1)/2$ ခု (triangular number) ပါ။ ဥပမာ လူ ၅ ဦးက အချင်းချင်း အကုန် သိကျွမ်းနေတဲ့ အစည်းအဝေး။
- **Incomplete graph** — အနည်းဆုံး vertex ၂ ခုကြား edge မရှိတဲ့ graph (real-world network တွေက အများအားဖြင့် ဒီအမျိုးအစားပါ)။

> K_n **notation** — K စာလုံးက ဂျာမန် *komplett* (ပြည့်စုံမှု) ကနေ ဆင်းသက်တယ်လို့ အချို့က ဆိုကြပြီး၊ graph theory ပါဆောင်ရွက်ခဲ့တဲ့ **Kazimierz Kuratowski** ကို ဂုဏ်ပြုမှု လည်း ပါပါတယ်။ (Wikipedia)



Edge အရေအတွက် sequence (triangular numbers):

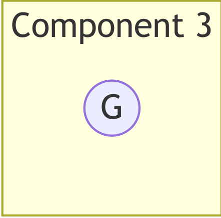
K_n	K_1	K_2	K_3	K_4	K_5	K_6	K_7
edges	0	1	3	6	10	15	21

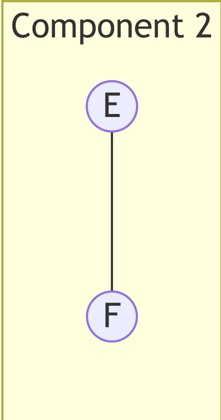
vertex `n` ခုစီရင် **degree =** $n-1$ ဖြစ်ပြီး — K_n ဟာ **regular graph** (vertex အကုန် degree တူ) တစ်မျိုးပါ။

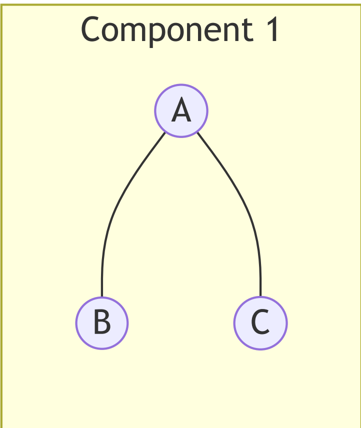
> **Geometric အဓိပ္ပာယ်** — K_3 က triangle၊ K_4 က tetrahedron (၃ ဘက် ပုံ) ၏ edge skeleton ဖြစ်ပါတယ်။ K_1 မှ K_4 အထိ planar (မျဉ်းတွေ မဖြတ်ဘဲ ဆွဲလို့ ရ) ဖြစ်ပေမယ့် — K_5 **ကစပြီး nonplanar** ဖြစ်ပါတယ်။ (Kuratowski's theorem)။
>
> Real-world မှာ complete graph ဟာ ရှားပါတယ် (လူအားလုံး အချင်းချင်း သိဖို့ ဆိုတာ ဖြစ်နိုင်ခြေ နည်း)။ ဒါကြောင့် real-world network တွေက **sparse (incomplete)** ဖြစ်ပြီး — **Adjacency List** ကို default အဖြစ် သုံးကြတာပါ။

အရေးကြီး Term များ — Degree, Path, Cycle

- **Degree** — vertex တစ်ခုမှာ ဆက်နေတဲ့ edge အရေအတွက်။ (Directed graph မှာ — အဝင် edge = **in-degree**၊ အထွက် edge = **out-degree**။)
- **Path** — vertex တစ်ခုကနေ နောက်တစ်ခုသို့ edge တွေ ဆက်ပြီး သွားတဲ့ လမ်းကြောင်း။ ဥပမာ A → B → D က A ကနေ D သို့ path တစ်ခု။
- **Cycle** — vertex တစ်ခုကနေ ထွက်ပြီး **ပြန်ရောက်လာတဲ့** path (A → B → D → C → A)။ Tree မှာ cycle မရှိ၊ graph မှာ ရှိနိုင်။
- **Connected Components** — တစ်ခုနဲ့ တစ်ခု ဆက်စပ်နေတဲ့ vertex အုပ်စု။ Graph တစ်ခုထဲမှာ တစ်ခုနဲ့ မဆက်တဲ့ အုပ်စု အများကြီး ရှိနိုင်တယ်။







G က degree 0 (သီးသန့်)။ → Connected component ၃ ခု: {A,B,C}, {E,F}, {G}

Graph ကို Code ထဲ ဘယ်လို ကိုယ်စားပြုမလဲ

Graph ကို memory ထဲ သိမ်းဖို့ နည်းလမ်း ၂ မျိုး အဓိက ရှိပါတယ် — **Adjacency List** နဲ့ **Adjacency Matrix**။ ဘယ်ဟာ ရွေးမလဲ ဆိုတာ graph က **dense (edge များ)** လား **sparse (edge နည်း)** လား ပေါ် မူတည်ပါတယ်။

Adjacency List (vertex တစ်ခုစီရဲ့ "အိမ်နီးချင်း" စာရင်း)

vertex တစ်ခုစီအတွက် — သူနဲ့ တိုက်ရိုက် ဆက်နေတဲ့ vertex တွေရဲ့ list ကို သိမ်းတာ။ Real-world မှာ **အသုံးအများဆုံး** ဖြစ်ပါတယ် (edge နည်းတဲ့ graph အတွက် memory သက်သာ)။

```
A: [B, C]
B: [A, D]
C: [A, D]
D: [B, C]
```

```java
// vertex 0..n-1 အတွက် adjacency list
List<List<Integer>> graph = new ArrayList<>();
for (int i = 0; i < n; i++) graph.add(new ArrayList<>());

// undirected edge ထည့်တာ — ၂ ဖက်စလုံး
graph.get(u).add(v);
graph.get(v).add(u);
```

Adjacency Matrix (n × n ဇယား)

`matrix[i][j] = 1` ဆို vertex `i` နဲ့ `j` ဆက်နေ၊ `0` ဆို မဆက်။ vertex နည်းပြီး edge များတဲ့ (dense) graph အတွက် ရှင်းလင်းပြီး — "i နဲ့ j ဆက်လား" ကို $O(1)$ နဲ့ စစ်လို့ ရတယ်။ ဒါပေမယ့် vertex `n` ခုဆို နေရာ $O(n^2)$ အမြဲ ယူသည်။

```
   A  B  C  D
A [0, 1, 1, 0]
B [1, 0, 0, 1]
C [1, 0, 0, 1]
D [0, 1, 1, 0]
```

နှိုင်းယှဉ်ချက်

	Adjacency List	Adjacency Matrix
Space	$O(V+E)$	$O(V^2)$
"u–v ဆက်လား" စစ်	$O(\deg(u))$	$O(1)$
အိမ်နီးချင်း အကုန် ရှာ	$O(\deg(u))$	$O(V)$
သင့်တော်	edge နည်း (sparse)	edge များ (dense)

> **V = vertex အရေအတွက်၊ E = edge အရေအတွက်။** Real-world graph အများစုက sparse (edge နည်း) ဖြစ်လို့ — **Adjacency List** ကို default အဖြစ် သုံးကြပါတယ်။

Real-world Examples

Graph ဟာ "**တစ်ခုနဲ့ တစ်ခု ဆက်စပ်နေတဲ့**" data မှန်သမျှမှာ တွေ့ရပါတယ် —

- **Users and Friends** — social network — user = vertex၊ friendship = undirected edge။ "common friend ရှာ"၊ "သိကျွမ်းခြင်း အဆင့် ၂ ဆင့်" တွေက graph ပြဿနာ။
- **City Routes / Maps** — မြို့ = vertex၊ လမ်း = weighted edge (အကွာအဝေး)။ GPS route finding က graph ပေါ်မှာ shortest path ရှာတာ။
- **Package Dependencies** — npm/pip package = vertex၊ "A က B ကို မှီခို" = directed edge။ install order ရှာတာ၊ circular dependency ရှိမရှိ စစ်တာ။
- **Permission Graph** — role → permission → resource ဆက်စပ်မှု၊ permission inheritance (admin က editor ရဲ့ permission အကုန် ရ)။
- **Workflow / State Machine** — order status (pending → paid → shipped → delivered) လို state တွေ ကူးပြောင်းမှု — state = vertex၊ transition = directed edge။
- **Task Dependency** — project task တွေ — "task A ပြီးမှ task B စလို့ ရ" — directed graph ဖြစ်ပြီး၊ scheduling/build system မှာ အသုံးဝင်။

ဒီ pattern တွေ အကုန်လုံး တူပါတယ် — **"အရာဝတ္ထု (vertex) တွေနဲ့ သူတို့ကြားက ဆက်စပ်မှု (edge) ကို model လုပ်ပြီး၊ ဆက်စပ်မှု ပေါ်မှာ မေးခွန်း ဖြေ"**။ ဒါက graph ရဲ့ အသုံးချပုံ အနှစ်ချုပ်ပါ။

Graph ရဲ့ အားသာချက်များ

- **လွတ်လပ်တဲ့ structure** — array, linked list, tree တွေလို ကန့်သတ်ချက် (sequential / hierarchy) မရှိဘဲ — ဆက်စပ်မှု ဘယ်လိုပုံစံနဲ့မဆို ကိုယ်စားပြုလို့ ရပါတယ်။
- **Real-world problem တွေကို model လုပ်နိုင်** — pathfinding, data clustering, network analysis, machine learning စတဲ့ နယ်ပယ် အများအပြားမှာ သင့်တော်။
- **ဆက်စပ်မှုကို ရှင်းရှင်းလင်းလင်း မြင်ခွင့် ပေး** — vertex–edge သဘောတရားက — ရှုပ်ထွေးတဲ့ relationship တွေကို မြင်သာအောင် ပြင်ပေးတယ်။
- **ဘယ် data မဆို graph ဖြစ်နိုင်** — item တွေနဲ့ သူတို့ကြားက ဆက်စပ်မှု ပေါ်လာတာနဲ့ — graph တစ်ခု ဖြစ်သွားပါပြီ။

Questions

Graph ကို ပြဿနာ ၄ ခုနဲ့ လေ့လာကြည့်ရအောင်။ ပြဿနာ အများစုက — graph ကို **adjacency list** အဖြစ် ပြောင်းပြီး၊ vertex တွေကို **traversal (DFS/BFS)** လုပ်ရင်း **visited** မှတ်ထားတာ ဖြစ်ပါတယ်။ (DFS/BFS ကို အခန်း ၁၈ မှာ အသေးစိတ် ဆက်လေ့လာမယ် — ဒီမှာ graph အပေါ် ပထမဆုံး အသုံးချကြည့်ပါမယ်။)

၁။ Find if Path Exists

vertex `n` ခု (`0..n-1`) နဲ့ undirected edge list `edges` ပေးထားသည်။ `source` ကနေ `destination` သို့ **path ရှိ/မရှိ** ပြန်ပါ။

Example 1:

```
Input: n = 4, edges = [[0,1],[1,2],[2,0],[3,1]], source = 0, destination = 3
Output: true
   (0 → 1 → 3  လမ်းကြောင်း ရှိသည်)
```

ရှင်းလင်းချက်

edge list ကို **adjacency list** အဖြစ် ပြောင်းပြီး — `source` ကနေ DFS/BFS နဲ့ ဆက်နေသမျှ vertex အကုန် ရောက်ကြည့်ပါတယ်။ `destination` ကို ရောက်ရင် `true` ။ **`visited` set** က — ဆင်းပြီးသား vertex ကို ပြန်မဆင်းအောင် (cycle ကြောင့် infinite loop မဖြစ်အောင်) တားသည်။

- adjacency list ဆောက် (undirected — ၂ ဖက်စလုံး ထည့်)။
- `source` ကစ DFS — `visited` မှတ်၊ အိမ်နီးချင်း တစ်ခုစီ ဆက်ဆင်း။
- `destination` တွေ့ရင် `true` ၊ အကုန် ဆင်းပြီး မတွေ့ရင် `false` ။

> **Time Complexity:** $O(V+E)$ - vertex နဲ့ edge တစ်ခုစီ တစ်ခါစီ ဖြတ်။
> **Space Complexity:** $O(V+E)$ - adjacency list + visited + recursion stack။

Java Solution

```java
class Solution {
    public boolean validPath(int n, int[][] edges, int source, int destination) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) graph.add(new ArrayList<>());
        for (int[] e : edges) {                 // undirected — ၂ ဖက်
            graph.get(e[0]).add(e[1]);
            graph.get(e[1]).add(e[0]);
        }
        return dfs(graph, source, destination, new boolean[n]);
    }

    private boolean dfs(List<List<Integer>> graph, int node, int dest, boolean[] visited)
{
        if (node == dest) return true;          // ရောက်ပြီ
        visited[node] = true;
        for (int next : graph.get(node)) {
            if (!visited[next] && dfs(graph, next, dest, visited)) return true;
        }
```

```
            return false;
        }
    }
```

၂။ Number of Connected Components

vertex `n` ခု (`0..n-1`) နဲ့ undirected edge list `edges` ပေးထားသည်။ **connected component** (တစ်ခုနဲ့ တစ်ခု ဆက်စပ်နေတဲ့ အုပ်စု) ဘယ်နှ ခု ရှိလဲ ပြန်ပါ။

Example 1:

```
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2
   ({0,1,2} နဲ့ {3,4} ၂ အုပ်စု)
```

ရှင်းလင်းချက်

vertex အကုန်ကို တစ်ခုချင်း ကြည့်ပါတယ် — **မ visit ရသေးတဲ့** vertex တစ်ခု တွေ့ရင် → **component အသစ် တစ်ခု** စပြီ → count တိုး၊ ပြီးတော့ DFS နဲ့ အဲ့ component ထဲက vertex အကုန်ကို `visited` မှတ်ပစ်။ ဒီနည်းနဲ့ component တစ်ခုစီကို **တစ်ခါပဲ** ရေတွက်ပါတယ်။

- adjacency list ဆောက်။
- vertex `0..n-1` loop — `visited` မဟုတ်ရင် `count++` + DFS (အုပ်စုတစ်ခုလုံး မှတ်)။

> **Time Complexity:** $O(V+E)$။
> **Space Complexity:** $O(V+E)$ - adjacency list + visited။

Java Solution

```java
class Solution {
    public int countComponents(int n, int[][] edges) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) graph.add(new ArrayList<>());
        for (int[] e : edges) {
            graph.get(e[0]).add(e[1]);
            graph.get(e[1]).add(e[0]);
        }
        boolean[] visited = new boolean[n];
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {                // component အသစ်
                count++;
                dfs(graph, i, visited);       // အုပ်စုတစ်ခုလုံး မှတ်
            }
        }
        return count;
    }

    private void dfs(List<List<Integer>> graph, int node, boolean[] visited) {
        visited[node] = true;
        for (int next : graph.get(node)) {
```

```
                if (!visited[next]) dfs(graph, next, visited);
            }
        }
    }
```

၃။ Detect Cycle (Undirected Graph)

vertex `n` ခု နဲ့ undirected edge list `edges` ပေးထားသည်။ graph ထဲမှာ **cycle ရှိ/မရှိ** ပြန်ပါ။

Example 1:

```
Input: n = 4, edges = [[0,1],[1,2],[2,0],[2,3]]
Output: true
   (0 → 1 → 2 → 0  ဝိုင်းပြန်နေသည်)
```

ရှင်းလင်းချက်

Undirected graph မှာ — DFS ဆင်းရင်း `visited` **ပြီးသား** ဖြစ်တဲ့ vertex တစ်ခုကို ပြန်တွေ့ရင် cycle ဖြစ်နိုင်တယ်။ ဒါပေမယ့် သတိ — A → B ဆင်းပြီး B ကနေ A ကို ပြန်ကြည့်တာက (undirected ဆိုတော့ A–B edge က ၂ ဖက်) cycle **မဟုတ်**။ ဒါကြောင့် **ဘယ်က လာလဲ (** `parent` **)** ကို မှတ်ထားပြီး — visited ဖြစ်တဲ့ vertex က `parent` **မဟုတ်ရင်မှ** cycle ဟု သတ်မှတ်သည်။

- DFS မှာ `parent` parameter ထည့်။
- အိမ်နီးချင်း `next` — မ visit ရသေးရင် ဆက်ဆင်း (parent = node)၊ visit ပြီးသား + **parent မဟုတ်** ဆို → cycle။
- component အကုန်ကို စစ်ဖို့ vertex အကုန် loop။

> **Time Complexity:** $O(V+E)$။
> **Space Complexity:** $O(V+E)$။

Java Solution

```
class Solution {
    public boolean hasCycle(int n, int[][] edges) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) graph.add(new ArrayList<>());
        for (int[] e : edges) {
            graph.get(e[0]).add(e[1]);
            graph.get(e[1]).add(e[0]);
        }
        boolean[] visited = new boolean[n];
        for (int i = 0; i < n; i++) {
            if (!visited[i] && dfs(graph, i, -1, visited)) return true;
        }
        return false;
    }

    private boolean dfs(List<List<Integer>> graph, int node, int parent, boolean[]
visited) {
        visited[node] = true;
```

```
            for (int next : graph.get(node)) {
                if (!visited[next]) {
                    if (dfs(graph, next, node, visited)) return true;
                } else if (next != parent) {       // visited + parent မဟုတ် → cycle
                    return true;
                }
            }
            return false;
        }
    }
```

> **Directed graph ၏ cycle** က နည်းနည်း ကွာသည် — "လက်ရှိ DFS လမ်းကြောင်းပေါ်မှာ ရှိနေတဲ့ (in-progress) vertex" ကို ပြန်တွေ့မှ cycle (back-edge) ဟု သတ်မှတ်သည်။ ဒါကို topological sort နဲ့ တွဲပြီး အခန်း ၁၉ မှာ ဆက်လေ့လာပါမယ်။

၄။ Clone Graph

connected undirected graph ရဲ့ node တစ်ခု (`node`) ပေးထားသည်။ graph တစ်ခုလုံးရဲ့ **deep copy (clone)** ကို ဆောက်ပြီး ပြန်ပါ။ node တစ်ခုစီမှာ `val` နဲ့ အိမ်နီးချင်း list `neighbors` ရှိသည်။

Example 1:

```
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
   (node 1 ↔ 2,4 ;  node 2 ↔ 1,3 ;  ... တူညီတဲ့ copy အသစ်)
```

ရှင်းလင်းချက်

ဒီပြဿနာရဲ့ အခက်ဆုံး အပိုင်းက — graph မှာ **cycle** ရှိလို့ node တစ်ခုကို clone လုပ်တိုင်း သူ့ neighbor တွေကိုလည်း clone လုပ်ရင်း **အဆုံးမရှိ loop** ဖြစ်နိုင်တာ။ ဒါကို ဖြေဖို့ — **"မူရင်း node → clone node" mapping** (HashMap) ကို သိမ်းထားပြီး၊ node တစ်ခုကို **တစ်ခါပဲ** clone လုပ်ပါတယ်။ ပြန်တွေ့ရင် map ထဲက clone ကို ပြန်သုံး။

- `node` ကို clone ဆောက်ပြီး map ထဲ ထည့်။
- neighbor တစ်ခုစီ — map ထဲ ရှိရင် ပြန်ယူ၊ မရှိရင် recursion နဲ့ clone ဆောက်။
- map က visited အလုပ်ပါ လုပ်ပေးလို့ — cycle ရှိလည်း loop မဖြစ်။

> **Time Complexity:** $O(V + E)$ - node နဲ့ edge တစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(V)$ - map + recursion stack။

Java Solution

```
// LeetCode ပေးထားတဲ့ Node definition ကို သုံးပါတယ်:
// class Node {
//     public int val;
//     public List<Node> neighbors;
//     public Node(int val) { this.val = val; this.neighbors = new ArrayList<>(); }
```

```
// }
class Solution {
    private Map<Node, Node> cloned = new HashMap<>();

    public Node cloneGraph(Node node) {
        if (node == null) return null;
        if (cloned.containsKey(node)) return cloned.get(node);  // clone ပြီးသား → ပြန်သုံး

        Node copy = new Node(node.val);        // clone အသစ်
        cloned.put(node, copy);                // map ထဲ ထည့် (cycle ကာကွယ်)
        for (Node nb : node.neighbors) {
            copy.neighbors.add(cloneGraph(nb)); // neighbor တွေ clone
        }
        return copy;
    }
}
```

အခန်း ၁၈ - BFS and DFS

အခန်း ၁၇ မှာ graph ကို **ကိုယ်စားပြု** — adjacency list / matrix ကို လေ့လာခဲ့ပါတယ်။ ပြဿနာ ဖြေတဲ့အခါ — "source ကစ ဆက်နေသမျှ vertex အကုန် ရောက်ကြည့်"၊ "component အုပ်စု ရေတွက်" စတဲ့နေရာတွေမှာ graph ကို **လှည့်ပတ်ရှာဖွေ (traverse)** ခဲ့ပါတယ်။ ဒီ traversal ၂ မျိုးက — **BFS (Breadth-First Search)** နဲ့ **DFS (Depth-First Search)** ပါ။

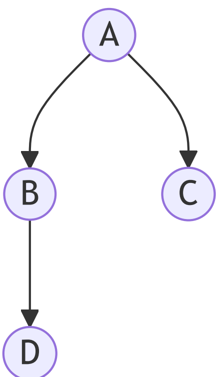
ဒီနှစ်ခုက graph/tree အလုပ်တိုင်းရဲ့ **အခြေခံ** ဖြစ်ပါတယ် — ကွာတာက "vertex တွေကို **ဘယ်အစဉ်** နဲ့ ရှာသွားလဲ" ဆိုတာပဲ —

- **DFS** — လမ်းကြောင်းတစ်ခုကို **အဆုံးထိ နက်နက်** ဆင်းပြီးမှ နောက်လမ်း ပြန်လှည့် (နက်ရှိုင်းအရင်)။
- **BFS** — start ကနေ **အနီးဆုံး အဆင့်** vertex တွေ အရင် ၊ ပြီးမှ နောက်အဆင့် (ပတ်လည် တဖြည်းဖြည်း ကျယ်)။

ဒီအခန်းမှာ — DFS/BFS ကို code ထဲ ဘယ်လို ရေးမလဲ၊ ဘယ်အချိန် ဘယ်ဟာ ရွေးမလဲ၊ ပြီးတော့ classic ပြဿနာ ၅ ခု (islands, flood fill, rotten oranges, course schedule, shortest path) ကို ဖြေကြည့်ပါမယ်။

DFS — နက်ရှိုင်းအရင် (Depth-First)

DFS က — vertex တစ်ခုကစ၊ အိမ်နီးချင်းတစ်ခုကို ရွေးပြီး **အဲ့လမ်းကို အဆုံးထိ နက်နက်** ဆင်းသွားတယ်။ ပိတ်သွားမှ (ဆက်မသွားနိုင်တော့မှ) **နောက်ပြန်လှည့် (backtrack)** ပြီး မစစ်ရသေးတဲ့ နောက်လမ်းကို ဆက်စစ်တယ်။



DFS အစဉ်: A → B → D (အဆုံး) → backtrack → C

DFS ကို ရေးနည်း ၂ မျိုး ရှိ — **recursion** (call stack ကို သုံး) နဲ့ **explicit stack** (ကိုယ်တိုင် stack သုံး)။ နှစ်ခုလုံး က "နောက်ဆုံးရောက် vertex ကို အရင်ဆက်" ဆိုတဲ့ **LIFO (stack)** သဘောတရားပဲ။

DFS — Recursion

အရိုးရှင်းဆုံး ရေးနည်း — function က သူ့ကိုယ်သူ ပြန်ခေါ်ပြီး၊ programming language ရဲ့ **call stack** က backtrack ကို အလိုအလျောက် လုပ်ပေးတယ်။

```
void dfs(List<List<Integer>> graph, int node, boolean[] visited) {
    visited[node] = true;                   // visit မှတ်
    // ... node ကို process လုပ်
    for (int next : graph.get(node)) {
        if (!visited[next]) dfs(graph, next, visited);  // မ visit ရသေးရင် ဆက်ဆင်း
    }
}
```

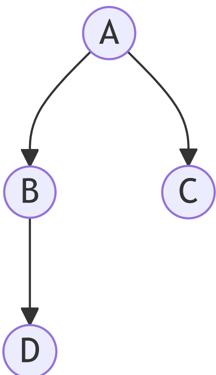
DFS — Explicit Stack

Recursion နက်လွန်းရင် **stack overflow** ဖြစ်နိုင်တယ်။ အဲ့ဒါကို ရှောင်ဖို့ — **stack** ကို ကိုယ်တိုင်သုံးပြီး iterative ရေးနိုင်တယ်။

```
void dfs(List<List<Integer>> graph, int start, boolean[] visited) {
    Deque<Integer> stack = new ArrayDeque<>();
    stack.push(start);
    while (!stack.isEmpty()) {
        int node = stack.pop(); // နောက်ဆုံးထည့်တာ အရင်ထွက် (LIFO)
        if (visited[node]) continue;
        visited[node] = true; // visit မှတ်
        for (int next : graph.get(node)) {
            if (!visited[next]) stack.push(next);
        }
    }
}
```

BFS — ပတ်လည်အရင် (Breadth-First)

BFS က — start vertex ကနေ **အနီးဆုံး အဆင့် (distance 1)** vertex အကုန်ကို အရင်ရှာ၊ ပြီးမှ **distance 2** အဆင့်၊ စသဖြင့် တလွှာချင်း ပတ်လည် ကျယ်သွားတယ်။ ဒါကို **queue (FIFO)** နဲ့ လုပ်တယ် — အရင်ထည့်တဲ့ vertex ကို အရင်ထုတ်ပြီး ဆက်စစ်။



BFS အစဉ်: A → B → C → D (အဆင့်လိုက်)

- အဆင့် 0: A
- အဆင့် 1: B, C
- အဆင့် 2: D

```
void bfs(List<List<Integer>> graph, int start, boolean[] visited) {
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(start);
    visited[start] = true; // queue ထဲ ထည့်တုန်းကပဲ မှတ်
    while (!queue.isEmpty()) {
        int node = queue.poll(); // အရင်ထည့်တာ အရင်ထွက် (FIFO)
        // ... node ကို process လုပ်
        for (int next : graph.get(node)) {
            if (!visited[next]) {
                visited[next] = true;
                queue.offer(next);
            }
        }
    }
}
```

> **သတိ:** BFS မှာ vertex ကို **queue ထဲ ထည့်တဲ့အချိန်မှာ** `visited` မှတ်ပါ — poll လုပ်တဲ့အချိန် မဟုတ်ပါ။ မဟုတ်ရင် vertex တစ်ခုတည်းကို queue ထဲ အကြိမ်များစွာ ထည့်မိနိုင်တယ်။

Visited Set — Cycle ကာကွယ်ခြင်း

Graph မှာ **cycle** ရှိနိုင်သည့် အတွက် visit ပြီးသား vertex ကို ပြန်မစစ်အောင် **`visited` set** ကို မှတ်ထားဖို့ **မဖြစ်မနေ လို**ပါတယ်။ မဟုတ်ရင် A → B → A → B ... **infinite loop** ဖြစ်သွားမယ်။

- **Tree** မှာ cycle မရှိလို့ — `visited` မလိုဘဲ traverse လို့ ရတယ် (parent ဆီ ပြန်မသွားရင်)။
- **Graph** မှာ — `visited` မဖြစ်မနေ လို။
- **Grid (matrix)** မှာ — visited array သုံးတာ၊ ဒါမှမဟုတ် cell ကို တန်ဖိုးပြောင်း (ဥပမာ `'1'` → `'0'`) ၍ "visit ပြီး" ကို မှတ်တာ။

ဘယ်အချိန် BFS, ဘယ်အချိန် DFS

နှစ်ခုလုံး vertex အကုန် ရောက်တာ တူပေမယ့် — **ရှာသွားတဲ့ အစဉ်** ကွာသည့် အတွက် ပြဿနာ အလိုက် ရွေးချယ်ရတယ်။

	BFS	DFS
Structure	Queue (FIFO)	Stack / Recursion (LIFO)
ရှာပုံ	အဆင့်လိုက် ပတ်လည်	လမ်းတစ်ခု အဆုံးထိ နက်နက်

Shortest path (unweighted)	✓ **ရ** (အနီးဆုံး အရင်)	✗ မရ
Memory	အဆင့်တစ်ခုလုံး queue ထဲ (ကျယ်ရင် များ)	လမ်းကြောင်း အနက် (နက်ရင် များ)
သင့်တော်	nearest / level / shortest	path exist / cycle / backtracking

အလွယ်မှတ်:

- "**အနီးဆုံး / အနည်းဆုံး အဆင့် / shortest path**" ဆို → **BFS**။
- "**path ရှိလား / အကုန်ရှာ / cycle / backtracking**" ဆို → **DFS** (recursion ရေးရ လွယ်)။

Real-world Examples

- **Search Dependency Tree** — package/task dependency ကို DFS နဲ့ ဆင်းပြီး "ဘယ် package တွေ မှီခိုလဲ" အကုန်ရှာ၊ install order ရှာ။
- **Find Nearest Item** — map/grid မှာ "အနီးဆုံး hospital / ATM" ရှာတာ — **BFS** (အနီးဆုံး အရင် ရောက်လို့)။
- **Crawl Linked Pages** — web crawler — page တစ်ခုကစ link တွေကို BFS/DFS နဲ့ လိုက်ဖွင့်၊ `visited` URL set နဲ့ ထပ်မဖွင့်အောင်။
- **Permission Inheritance** — role → permission graph ကို DFS နဲ့ ဆင်းပြီး "user ဒီ permission ရှိလား" inherited အကုန်စစ်။
- **Social Network** — "ဒီ user နဲ့ ဘယ်နှ ဆင့်ကွာလဲ (degrees of separation)" — BFS နဲ့ အဆင့် ရေတွက်။

Questions

Graph/grid ပြဿနာ အများစုက — DFS/BFS template ၂ ခုထဲက တစ်ခုကို လိုက်ဖော်ပြီး `visited` မှတ်ရုံပါပဲ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Number of Islands

`'1'` (ကုန်း) နဲ့ `'0'` (ရေ) ပါတဲ့ 2D grid ပေးထားသည်။ **island** (ဘေးချင်းကပ် ကုန်းအုပ်စု — အပေါ်/အောက်/ဘယ်/ညာ ဆက်နေ) ဘယ်နှ ခု ရှိလဲ ပြန်ပါ။

Example 1:

```
Input: grid = [
  ["1","1","0","0"],
  ["1","1","0","0"],
  ["0","0","1","0"],
  ["0","0","0","1"]
]
Output: 3
   (ဘယ်အပေါ် အုပ်စု၊ အလယ် 1၊ အောက်ညာ 1 — ၃ ကျွန်း)
```

ရှင်းလင်းချက်

cell အကုန်ကို loop — `'1'` တစ်ခု တွေ့ရင် → **island အသစ် တစ်ခု** စပြီ → count တိုး၊ ပြီးတော့ DFS နဲ့ အဲ့ island နဲ့ ဆက်နေတဲ့ `'1'` အကုန်ကို `'0'` ပြောင်းပစ် ("visit ပြီး" မှတ်ခြင်း)။ ဒီနည်းနဲ့ island တစ်ခုစီကို တစ်ခါပဲ ရေတွက်တယ်။ (connected component ရေတွက်တာ — grid version ပါ။)

- cell `(r,c)` loop — `'1'` ဆို `count++` + DFS။
- DFS — `'1'` ကို `'0'` ပြောင်း၊ ဘေး ၄ ဖက် ဆက်ဆင်း (grid အပြင် မထွက်အောင် စစ်)။

> **Time Complexity:** $O(m \times n)$ - cell တစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(m \times n)$ - worst case recursion stack (grid အကုန် `'1'`)။

Java Solution

```java
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == '1') {       // island အသစ်
                    count++;
                    dfs(grid, r, c);           // ဆက်နေတဲ့ ကုန်း အကုန် နှစ်ပစ်
                }
            }
        }
        return count;
    }

    private void dfs(char[][] grid, int r, int c) {
        if (r < 0 || r >= grid.length || c < 0 || c >= grid[0].length
            || grid[r][c] != '1') return;      // အပြင်ထွက် / ရေ / visit ပြီး
        grid[r][c] = '0';                      // visit မှတ် (ရေ ပြောင်း)
        dfs(grid, r + 1, c);
        dfs(grid, r - 1, c);
        dfs(grid, r, c + 1);
        dfs(grid, r, c - 1);
    }
}
```

၂။ Flood Fill

image ကို 2D grid `image` (cell တစ်ခုစီ = အရောင်) နဲ့ ပေးထားသည်။ start cell `(sr, sc)` ကစ — **တူညီတဲ့ အရောင်** ဘေးချင်းကပ် cell အကုန်ကို အရောင်အသစ် `color` နဲ့ ပြောင်းပါ (paint bucket tool)။

Example 1:

```
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
   ((1,1) ကစ ဆက်နေတဲ့ 1 အကုန် → 2)
```

ရှင်းလင်းချက်

start cell ရဲ့ မူရင်းအရောင်ကို မှတ်၊ DFS နဲ့ ဆက်နေတဲ့ **တူညီအရောင်** cell အကုန်ကို `color` ပြောင်း။ Number of Islands နဲ့ တူ — "ဆက်နေတဲ့ region" ကို DFS နဲ့ စစ်တာ။ **သတိ:** အရောင်အသစ်က မူရင်းအရောင်နဲ့ တူရင် — ပြောင်းစရာ မလို (infinite loop ရှောင်)။

- `start` cell ရဲ့ မူရင်းအရောင် `old` မှတ်၊ `old == color` ဆို ချက်ချင်း ပြန်။
- DFS — `old` အရောင် cell ကို `color` ပြောင်း၊ ဘေး ၄ ဖက် ဆက်။

> **Time Complexity:** $O(m \times n)$။
> **Space Complexity:** $O(m \times n)$ - recursion stack။

Java Solution

```java
class Solution {
    public int[][] floodFill(int[][] image, int sr, int sc, int color) {
        int old = image[sr][sc];
        if (old != color) dfs(image, sr, sc, old, color);
        return image;
    }

    private void dfs(int[][] image, int r, int c, int old, int color) {
        if (r < 0 || r >= image.length || c < 0 || c >= image[0].length
            || image[r][c] != old) return;      // အပြင် / အရောင်မတူ
        image[r][c] = color;                    // ပြောင်း (visit မှတ် ပါ)
        dfs(image, r + 1, c, old, color);
        dfs(image, r - 1, c, old, color);
        dfs(image, r, c + 1, old, color);
        dfs(image, r, c - 1, old, color);
    }
}
```

၃။ Rotting Oranges

grid မှာ `0` = ဗလာ၊ `1` = လတ်ဆတ်တဲ့ လိမ္မော်သီး၊ `2` = ပုပ်နေတဲ့ သီး။ မိနစ်တိုင်း — ပုပ်သီး ဘေးချင်းကပ် (၄ ဖက်) **လတ်ဆတ်သီးတွေ ပုပ်ကုန်**တယ်။ သီးအားလုံး ပုပ်ဖို့ **အနည်းဆုံး ဘယ်နှ မိနစ်** လိုလဲ ပြန်ပါ။ မဖြစ်နိုင်ရင် `-1` ။

Example 1:

```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

ရှင်းလင်းချက်

ဒါက **multi-source BFS** ပါ — **ပုပ်သီး အကုန်လုံး** ကို start (queue ထဲ အတူတူ ထည့်) ပြီး — အဆင့်တစ်ဆင့်ချင်း (မိနစ်တစ်ခုချင်း) ပတ်လည် ပျံ့သွားတာ။ BFS က "အဆင့်လိုက် ကျယ်" တာ ဖြစ်လို့ — အဆင့် အရေအတွက် = မိနစ်။ နောက်ဆုံး လတ်ဆတ်သီး ကျန်ရင် `-1` ။

- ပုပ်သီး အကုန် queue ထဲ ထည့်၊ လတ်ဆတ်သီး အရေအတွက် `fresh` ရေ။
- BFS — အဆင့်တစ်ခုစီ (`minutes++`) မှာ — queue ထဲက ပုပ်သီးတွေရဲ့ ဘေး လတ်ဆတ်သီးကို ပုပ်စေ + queue ထဲ ထည့် + `fresh--` ။
- ပြီးတော့ `fresh == 0` ဆို `minutes` ၊ မဟုတ်ရင် `-1` ။

> **Time Complexity:** $O(m \times n)$။
> **Space Complexity:** $O(m \times n)$ - queue။

Java Solution

```
class Solution {
    public int orangesRotting(int[][] grid) {
        int m = grid.length, n = grid[0].length, fresh = 0;
        Queue<int[]> queue = new LinkedList<>();
        for (int r = 0; r < m; r++)
            for (int c = 0; c < n; c++) {
                if (grid[r][c] == 2) queue.offer(new int[]{r, c});  // source အကုန်
                else if (grid[r][c] == 1) fresh++;
            }
        if (fresh == 0) return 0;

        int minutes = 0;
        int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
        while (!queue.isEmpty() && fresh > 0) {
            minutes++;
            for (int i = queue.size(); i > 0; i--) {   // အဆင့်တစ်ခုလုံး (၁ မိနစ်)
                int[] cell = queue.poll();
                for (int[] d : dirs) {
                    int nr = cell[0] + d[0], nc = cell[1] + d[1];
                    if (nr < 0 || nr >= m || nc < 0 || nc >= n || grid[nr][nc] != 1)
continue;
                    grid[nr][nc] = 2;              // ပုပ်စေ
                    fresh--;
                    queue.offer(new int[]{nr, nc});
                }
            }
        }
        return fresh == 0 ? minutes : -1;
    }
}
```

၄။ Shortest Path in Binary Matrix

`n × n` binary grid မှာ `0` = သွားလို့ ရ၊ `1` = ပိတ်။ ဘယ်အပေါ်ထောင့် `(0,0)` ကနေ ညာအောက်ထောင့် `(n-1,n-1)` သို့ — **၈ ဖက် (ထောင့်ဖြတ် ပါ)** သွားလို့ ရတဲ့ **အတိုဆုံး လမ်းကြောင်း** ရဲ့ cell အရေအတွက် ပြန်ပါ။ မရှိရင် `-1` ။

Example 1:

```
Input: grid = [[0,0,0],[1,1,0],[1,1,0]]
Output: 4
```

```
((0,0)→(0,1)→(1,2)→(2,2) လမ်းကြောင်း — cell ၄ ခု)
```

ရှင်းလင်းချက်

Shortest path (unweighted) ဆို → **BFS**။ start cell ကစ — အဆင့်လိုက် ပတ်လည် ကျယ်သွားတယ်၊ destination ကို **ပထမဆုံး ရောက်တဲ့ အဆင့်** က အတိုဆုံး။ ဒီမှာ ၈ ဖက် (ထောင့်ဖြတ် ပါ) ရွှေ့လို့ရတယ်။ DFS မသုံးရတဲ့ အကြောင်း — DFS က "ပထမတွေ့တဲ့ လမ်း" ကို ပြန်တာ၊ အတိုဆုံး မဟုတ်နိုင်။

- start `(0,0)` ပိတ် နေရင် `-1` ။
- BFS — cell `(r,c,dist)` queue ထဲ ထည့်၊ `visited` မှတ်။
- destination ရောက်ရင် `dist` ပြန်၊ queue ကုန်လည်း မရောက်ရင် `-1` ။

> **Time Complexity:** $O(n^2)$ - cell တစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(n^2)$ - queue + visited။

Java Solution

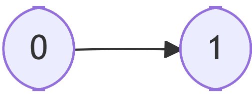
```
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        int n = grid.length;
        if (grid[0][0] == 1 || grid[n-1][n-1] == 1) return -1;

        int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1},{1,1},{1,-1},{-1,1},{-1,-1}};
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{0, 0, 1});       // (r, c, dist)
        grid[0][0] = 1;                         // visit မှတ်

        while (!queue.isEmpty()) {
            int[] cur = queue.poll();
            int r = cur[0], c = cur[1], dist = cur[2];
            if (r == n-1 && c == n-1) return dist;   // ပထမဆုံး ရောက် = အတိုဆုံး
            for (int[] d : dirs) {
                int nr = r + d[0], nc = c + d[1];
                if (nr < 0 || nr >= n || nc < 0 || nc >= n || grid[nr][nc] != 0)
continue;
                grid[nr][nc] = 1;               // visit မှတ် (ထည့်တုန်းက)
                queue.offer(new int[]{nr, nc, dist + 1});
            }
        }
        return -1;
    }
}
```

၅။ Course Schedule

course `numCourses` ခု (`0..numCourses-1`) နဲ့ prerequisite list `prerequisites` ပေးထားသည် — `[a, b]` ဆို "course `a` တက်ဖို့ `b` အရင်ပြီးရမယ်"။ course အကုန် တက်လို့ **ရ/မရ** (cycle မရှိ/ရှိ) ပြန်ပါ။

Example 1:

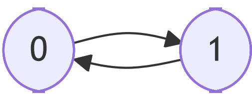
```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
   (course 0 အရင်၊ ပြီးမှ 1 — ဖြစ်နိုင်)
```



prerequisite မရှိ → 1 ကို တက်လို့ ရ။ Cycle မရှိ။

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
   (0 ← 1 ← 0  ဝိုင်းပြန် — မဖြစ်နိုင်)
```



0 → 1 → 0 ဝိုင်းပြန် — နှစ်ခုလုံး အချင်းချင်း မှီခို → Cycle ရှိ။

ရှင်းလင်းချက်

ဒါက **directed graph မှာ cycle ရှိ/မရှိ** စစ်တာ — cycle ရှိရင် course အကုန် ဘယ်တော့မှ မပြီးနိုင်။ ဒီမှာ **topological thinking** ကို မိတ်ဆက်ပါမယ် — **BFS topological sort (Kahn's algorithm)** သုံးမယ်။ prerequisite မရှိတဲ့ (in-degree 0) course ကစ တက်၊ တက်ပြီးတိုင်း သူ့ပေါ် မှီခိုတဲ့ course တွေရဲ့ in-degree လျှော့၊ 0 ဖြစ်ရင် queue ထဲ ထည့်။ နောက်ဆုံး တက်ပြီး course အရေအတွက် = `numCourses` ဆို — cycle မရှိ (`true`)။

- adjacency list + in-degree array ဆောက်။
- in-degree 0 course အကုန် queue ထဲ။
- BFS — course တစ်ခု ထုတ်တိုင်း `taken++` ၊ neighbor in-degree လျှော့၊ 0 ဆို ထည့်။
- `taken == numCourses` ⇒ `true` ။

> **Time Complexity:** $O(V + E)$ - course + prerequisite။
> **Space Complexity:** $O(V + E)$。

Java Solution

```java
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());
        int[] indegree = new int[numCourses];
```

```
        for (int[] p : prerequisites) {        // b → a  (b အရင်)
            graph.get(p[1]).add(p[0]);
            indegree[p[0]]++;
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++)
            if (indegree[i] == 0) queue.offer(i);   // prerequisite မရှိ — အရင်တက်

        int taken = 0;
        while (!queue.isEmpty()) {
            int course = queue.poll();
            taken++;
            for (int next : graph.get(course)) {
                if (--indegree[next] == 0) queue.offer(next);
            }
        }
        return taken == numCourses;            // အကုန် တက်နိုင် = cycle မရှိ
    }
}
```

> **Topological Sort** (dependency order ရှာတာ) နဲ့ Course Schedule II (တက်ရမယ့် အစဉ် ပြန်တာ) ကို အခန်း ၁၉ — **Advanced Graph Algorithms** မှာ အသေးစိတ် ဆက်လေ့လာပါမယ်။

အခန်း ၁၉ - Advanced Graph Algorithms

အခန်း ၁၈ မှာ graph တစ်ခုအတွင်းမှာ ရှိတဲ့ vertex တွေကို စနစ်တကျ လှည့်ပတ်ရှာဖွေတဲ့ (traversal) နည်းလမ်းတွေဖြစ်တဲ့ **BFS (Breadth-First Search)** နဲ့ **DFS (Depth-First Search)** အကြောင်းကို အသေးစိတ် လေ့လာခဲ့ကြပြီး ဖြစ်ပါတယ်။ ဒါပေမယ့် တကယ့် လက်တွေ့ကမ္ဘာ (real-world) ပြဿနာတွေမှာ "vertex တွေ တစ်ခုနဲ့တစ်ခု ဆက်နေသလား" ဆိုတာထက် ပိုမို ရှုပ်ထွေးပြီး နက်နဲတဲ့ မေးခွန်းတွေကို ဖြေရှင်းဖို့ လိုအပ်လာပါတယ်။

ဥပမာအားဖြင့် —

- **Dependency Ordering:** "Software package တွေ၊ သို့မဟုတ် task တွေကို install လုပ်တဲ့အခါ ဘယ်အရာကို အရင်လုပ်ပြီး ဘယ်အရာကို နောက်မှ လုပ်ရမလဲ"
- **Weighted Shortest Path:** "မြို့ A ကနေ မြို့ B ကို သွားတဲ့အခါ ခရီးအကွာအဝေး သို့မဟုတ် သွားလာစရိတ် အသက်သာဆုံး လမ်းကြောင်းကို ဘယ်လို ရှာမလဲ"
- **Minimum Spanning Tree:** "မြို့ နှစ်မြို့ကို လျှပ်စစ်ဓါတ်အားလိုင်း သို့မဟုတ် ဖုန်းလိုင်းတွေ ချိတ်ဆက်တဲ့အခါ ကုန်ကျစရိတ် အနည်းဆုံးဖြစ်အောင် ဘယ်လို ချိတ်ဆက်မလဲ"

ဒီအခန်းမှာတော့ real-world developer တစ်ယောက်အနေနဲ့ နေ့စဉ် ကြုံတွေ့ရလေ့ရှိတဲ့ အရေးပါဆုံး Graph Algorithm တွေကို အခြေခံမှစ၍ လက်တွေ့ကျကျ ရှင်းပြသွားပါမယ်။ Algorithm တစ်ခုစီအတွက် ၎င်းတို့ ဖြေရှင်းပေးသည့် ပြဿနာ၊ အလုပ်လုပ်ပုံ သဘောတရား (Intuition)၊ တစ်ဆင့်ချင်း လုပ်ဆောင်ပုံ (Step trace)၊ Code implementations နှင့် Time/Space complexity တွေကို အသေးစိတ် လေ့လာသွားကြပါမယ်။

ဒီအခန်းမှာ လေ့လာသွားမယ့် အဓိက Algorithm များနှင့် ၎င်းတို့၏ အဓိက သဘောတရားများမှာ အောက်ပါအတိုင်း ဖြစ်ပါတယ် —

Algorithm	ဖြေရှင်းပေးသည့် ပြဿနာ	အဓိက သဘောတရား / ကိရိယာ
Topological Sort	Dependency order (DAG)	In-degree + Queue (Kahn's Algorithm)
Dijkstra	Weighted shortest path (Edge weight ≥ 0)	Min-Heap (Greedy approach)
A* (A-star)	Goal-directed shortest path	Min-Heap + Heuristic function
Bellman-Ford	Shortest path with negative weights	Edge relaxation (Dynamic Programming)
Floyd-Warshall	All-pairs shortest path	3-nested loops (Dynamic Programming)
Union Find	Disjoint sets grouping / Cycle detection	Parent array with Path Compression

MST (Kruskal / Prim)	Minimum cost connect-all tree	Union Find / Min-Heap

---

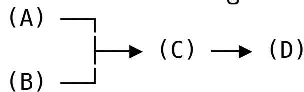
Topological Sort — Dependency Ordering

အခြေခံ သဘောတရားနှင့် အသုံးဝင်ပုံ

Topological Sort ဆိုတာ **Directed Acyclic Graph (DAG)** — ဆိုလိုတာက ဦးတည်ချက်ရှိပြီး cycle (ဝိုင်းပတ်မှု) မရှိတဲ့ graph တစ်ခုမှာ vertex တွေရဲ့ "ဘယ်အရာက အရင်၊ ဘယ်အရာက နောက်" ဆိုတဲ့ တရားဝင် လုပ်ဆောင်မှု အစီအစဉ် (valid ordering) ကို ရှာဖွေပေးတဲ့ algorithm ဖြစ်ပါတယ်။

graph ထဲမှာ directed edge `A → B` ရှိနေတယ်ဆိုရင် "B ကို မလုပ်ခင် A ကို အရင်ပြီးအောင် လုပ်ရမယ်" (A ပြီးမှ B) လို့ အဓိပ္ပါယ်ရပါတယ်။ ဒါကြောင့် Topological Sort ဖြင့် ထွက်လာတဲ့ အစီအစဉ်မှာ node `A` ဟာ node `B` ရဲ့ အရှေ့မှာ အမြဲတမ်း ရောက်ရှိနေရပါမယ်။

```
package dependency ဥပမာ:
```




```
- C ကို သုံးဖို့ A နဲ့ B အရင်လိုအပ်တယ်။
- D ကို သုံးဖို့ C အရင်လိုအပ်တယ်။

တရားဝင်သော အစီအစဉ် (Valid Order):   A, B, C, D  (သို့မဟုတ်)  B, A, C, D
မတရားဝင်သော အစီအစဉ် (Invalid Order): C, A, B, D  (C က A/B ရဲ့ အရှေ့ ရောက်နေ၍ မှားယွင်းသည်)
```

လက်တွေ့ကမ္ဘာမှာ Software Package Management (npm, pip, maven) တွေမှာ package တွေ install လုပ်မယ့် အစီအစဉ် ရှာတာ၊ Build tools (Make, Bazel, Webpack) တွေမှာ source file တွေ compile လုပ်မယ့် အစီအစဉ် သတ်မှတ်တာ၊ တက္ကသိုလ် ဘာသာရပ် prerequisites တွေနဲ့ Task Scheduling ပြဿနာတွေမှာ Topological Sort ကို အဓိက သုံးကြပါတယ်။

> **အရေးကြီးသော အချက်:** Topological Order ရရှိဖို့အတွက် Graph မှာ **Cycle (ဝိုင်းပတ်မှု) လုံးဝ မရှိရပါ** (DAG ဖြစ်ရပါမည်)။ အကယ်၍ "A ပြီးမှ B၊ B ပြီးမှ A" ဆိုပြီး Cycle ဖြစ်နေရင် circular dependency ဖြစ်ပေါ်နေတာ ကြောင့် ဘယ်ဟာကိုမှ စတင် လုပ်ဆောင်လို့ ရတော့မည် မဟုတ်ပါ။

Kahn's Algorithm (In-degree နည်းလမ်း)

Topological Sort ကို ရေးသားဖို့ အသုံးအများဆုံး နည်းလမ်းမှာ **Kahn's Algorithm** ဖြစ်ပြီး ၎င်းသည် **In-degree** သဘောတရားကို အခြေခံထားပါတယ်။

- **In-degree:** Vertex တစ်ခုဆီသို့ ဝင်ရောက်လာသော edge အရေအတွက် ဖြစ်ပါတယ်။ ဒါဟာ "ဒီ task ကို မလုပ်ခင် အရင်ပြီးအောင် လုပ်ရမယ့် prerequisite အရေအတွက်" ကို ဆိုလိုတာဖြစ်ပါတယ်။

- **In-degree = 0:** Prerequisite တစ်ခုမှ မရှိတော့တဲ့အတွက် ဒီ vertex ကို **အခုချက်ချင်း စတင် လုပ်ဆောင်လို့ ရပြီ** လို့ ဆိုလိုပါတယ်။

အလုပ်လုပ်ပုံ နည်းလမ်း:

1. Graph အတွင်းရှိ vertex အားလုံး၏ in-degree ကို စတင် တွက်ချက်ပါတယ်။
2. In-degree `0` ဖြစ်သော vertex တွေကို Queue ထဲသို့ ထည့်သွင်းပါတယ်။
3. Queue ထဲမှ vertex များကို တစ်ခုချင်းစီ ထုတ်ယူကာ topological order စာရင်းထဲသို့ ထည့်ပါတယ်။
4. ထုတ်ယူလိုက်သော vertex နှင့် ချိတ်ဆက်ထားသည့် အိမ်နီးချင်း vertex များ၏ in-degree ကို `1` လျှော့ပေးပါတယ်။ အကယ်၍ အိမ်နီးချင်း vertex ၏ in-degree သည် `0` ဖြစ်သွားပါက ၎င်းကို Queue ထဲသို့ ထပ်မံ ထည့်သွင်းပါတယ်။
5. Queue ဗလာ ဖြစ်သွားသည်အထိ ဒီလုပ်ငန်းစဉ်ကို ထပ်ခါထပ်ခါ လုပ်ဆောင်ပါတယ်။

အကယ်၍ Graph ထဲမှာ Cycle ရှိနေပါက Cycle ထဲတွင် ပါဝင်သော vertex များ၏ in-degree သည် ဘယ်တော့မှ `0` ဖြစ်လာမည် မဟုတ်ပါ။ ဒါကြောင့် ထွက်လာသော topological order ၏ ရလဒ် အရေအတွက်သည် စုစုပေါင်း vertex အရေအတွက် `n` ထက် နည်းနေပါလိမ့်မည်။ ဒါကိုကြည့်ပြီး Graph ထဲမှာ Cycle ရှိမရှိပါ တစ်ပါတည်း စစ်ဆေးနိုင်ပါတယ်။

Step-by-step Trace

အထက်ပါ ဥပမာ (`A→C, B→C, C→D`) ကို Kahn's Algorithm ဖြင့် စနစ်တကျ လိုက်ပါ ကြည့်ရအောင် —

```
၁။ In-degree များ စတင်တွက်ချက်ခြင်း:
   in-degree: A = 0, B = 0, C = 2, D = 1
   Queue: [A, B] (in-degree 0 ရှိသော A နှင့် B အား ထည့်ထားသည်)

၂။ Queue မှ A ကို ထုတ်ယူသည် (Order = [A]):
   - C ရဲ့ in-degree ကို 1 လျှော့သည် (C in-degree: 2 -> 1)

၃။ Queue မှ B ကို ထုတ်ယူသည် (Order = [A, B]):
   - C ရဲ့ in-degree ကို 1 ထပ်မံလျှော့သည် (C in-degree: 1 -> 0)
   - C ရဲ့ in-degree 0 ဖြစ်သွားပြီဖြစ်၍ C အား Queue ထဲသို့ ထည့်သည် (Queue = [C])

၄။ Queue မှ C ကို ထုတ်ယူသည် (Order = [A, B, C]):
   - D ရဲ့ in-degree ကို 1 လျှော့သည် (D in-degree: 1 -> 0)
   - D ရဲ့ in-degree 0 ဖြစ်သွားပြီဖြစ်၍ D အား Queue ထဲသို့ ထည့်သည် (Queue = [D])

၅။ Queue မှ D ကို ထုတ်ယူသည် (Order = [A, B, C, D]):
   - ဆက်လက် လုပ်ဆောင်စရာ Edge မရှိတော့ပါ။

ရလဒ် အစီအစဉ်: [A, B, C, D] ( size = 4 == n ဖြစ်၍ Cycle မရှိပါ ✓ )
```

Java Implementation

```java
import java.util.*;

public class TopologicalSort {
    public List<Integer> topoSort(int n, List<List<Integer>> graph) {
```

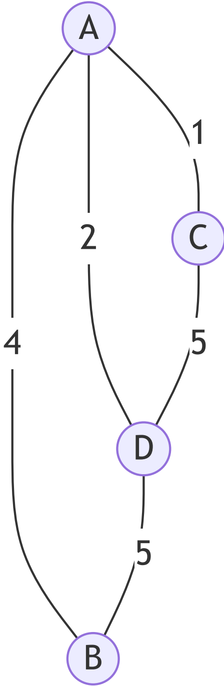
```java
        int[] indegree = new int[n];
        // Vertex တစ်ခုစီ၏ in-degree ကို စတင်တွက်ချက်ခြင်း
        for (int u = 0; u < n; u++) {
            for (int v : graph.get(u)) {
                indegree[v]++;
            }
        }

        // In-degree 0 ရှိသော vertex များကို Queue ထဲသို့ ထည့်ခြင်း
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }

        List<Integer> order = new ArrayList<>();
        while (!queue.isEmpty()) {
            int node = queue.poll();
            order.add(node);

            // အိမ်နီးချင်း vertex များ၏ in-degree ကို လျှော့ချခြင်း
            for (int next : graph.get(node)) {
                indegree[next]--;
                if (indegree[next] == 0) {
                    queue.offer(next);
                }
            }
        }

        // ရလဒ်အရွယ်အစား n နှင့် တူညီပါက valid order ဖြစ်သည်၊ မတူပါက cycle ရှိနေသည်
        return order.size() == n ? order : new ArrayList<>();
    }
}
```

> **Complexity Analysis:**
> - **Time Complexity:** $O(V + E)$ — Vertex နှင့် Edge တစ်ခုစီကို တစ်ကြိမ်စီသာ ဖြတ်သန်းကြည့်ရှု သောကြောင့် ဖြစ်ပါတယ်။
> - **Space Complexity:** $O(V + E)$ — Adjacency list၊ In-degree array နှင့် Queue အတွက် နေရာယူ သောကြောင့် ဖြစ်ပါတယ်။

Dijkstra's Algorithm — Weighted Shortest Path

အခြေခံ သဘောတရားနှင့် Greedy Logic

အခန်း ၁၈ ၌ လေ့လာခဲ့သော **BFS** သည် edge တွေမှာ အလေးချိန် (weight) မရှိသည့် unweighted graph များတွင် အတိုဆုံး လမ်းကြောင်း (shortest path) ကို ရှာဖွေပေးနိုင်ခဲ့ပါတယ်။ သို့သော်လည်း တကယ့် လက်တွေ့ မြေပုံများတွင် လမ်းကြောင်း တစ်ခုစီ၏ အကွာအဝေး၊ သို့မဟုတ် တန်ဖိုး (edge weight) များသည် မတူညီကြပါ။ ဒီလို **weighted graph** များတွင် Single-Source Shortest Path (စတင်မှတ် တစ်ခုမှ ကျန် vertex အားလုံးသို့ အတိုဆုံး လမ်းကြောင်း) ရှာဖွေဖို့ရာ **Dijkstra's Algorithm** ကို သုံးစွဲကြပါတယ်။



- A မှ C သို့ တိုက်ရိုက်: `1`
- A မှ B သို့ တိုက်ရိုက်: `4` (သို့သော် `A -> C -> D -> B` ကဲ့သို့ ပိုမို သက်သာသော လမ်းကြောင်း ရှိနိုင်ပါသလား)

Dijkstra ၏ အဓိက Greedy သဘောတရား:
"စတင်မှတ် (Start node) မှ လက်ရှိ အကွာအဝေး အနည်းဆုံး (အသက်သာဆုံး) ရှိနေသည့် vertex ကို အမြဲတမ်း ပထမဦးစားပေး ရွေးချယ်ပြီး finalize လုပ်သွားခြင်း" ဖြစ်ပါတယ်။

ဒီ Greedy logic မှန်ကန်ဖို့အတွက် အရေးကြီးသော ကန့်သတ်ချက် တစ်ခုရှိပါတယ် — Graph အတွင်းရှိ **Edge weight များသည် အနုတ်ကိန်း မဟုတ်ရပါ (Non-negative: edge weight ≥ 0)**။ Edge weight များ အပေါင်းကိန်း သို့မဟုတ် ၀ ဖြစ်နေသမျှ ကာလပတ်လုံး၊ လက်ရှိ အတိုဆုံး ရောက်ရှိနေသော vertex ၏ distance ကို အခြား မည်သည့် လမ်းကြောင်းကမှ ထပ်မံ လျှော့ချပေးနိုင်တော့မည် မဟုတ်ပါ။ (အကြောင်းမှာ မည်သည့် edge ကိုမဆို ထပ်မံ ကူးဖြတ်ပါက weight သည် တိုးလာရုံသာ ရှိပြီး လျော့ကျသွားမည် မဟုတ်သောကြောင့် ဖြစ်ပါသည်)။

အကွာအဝေး အနည်းဆုံး vertex ကို အမြန်ဆုံး ဆွဲထုတ်နိုင်ရန် **Min-Heap (PriorityQueue)** ကို သုံးစွဲကြပါတယ်။

Relaxation သဘောတရား

Shortest-path algorithm တွေရဲ့ အဓိက အနှစ်သာရမှာ **Edge Relaxation** ဖြစ်ပါတယ်။

```
if (dist[u] + weight(u, v) < dist[v]) {
    dist[v] = dist[u] + weight(u, v);
}
```

ဒါဟာ "Start မှ `v` သို့ လက်ရှိ သိထားသော အတိုဆုံး လမ်းကြောင်းထက်၊ `u` ကို ဖြတ်သန်းပြီးမှ `v` သို့ သွားသော လမ်းကြောင်းက ပိုမို တိုတောင်းပါက `v` ၏ အကွာအဝေး တန်ဖိုး (dist[v]) ကို အသစ် ပြင်ဆင် (update) မယ်" ဆိုတဲ့ သဘောတရား ဖြစ်ပါတယ်။

Step-by-step Trace

အထက်ပါ Graph တွင် `A` မှ စတင်၍ Dijkstra algorithm ဖြင့် အတိုဆုံး လမ်းကြောင်း ရှာပုံကို လိုက်ပါ ကြည့်ရအောင် —

```
စတင်ချိန်: dist = {A: 0, B: ∞, C: ∞, D: ∞}, PriorityQueue = [(0, A)]

၁။ PQ မှ (0, A) ကို ထုတ်ယူသည် ( A အား Finalize လုပ်သည် ):
   - Edge A->B (weight 4): dist[B] = min(∞, 0 + 4) = 4  -> PQ ထဲသို့ (4, B) ထည့်သည်
   - Edge A->C (weight 1): dist[C] = min(∞, 0 + 1) = 1  -> PQ ထဲသို့ (1, C) ထည့်သည်
   - Edge A->D (weight 2): dist[D] = min(∞, 0 + 2) = 2  -> PQ ထဲသို့ (2, D) ထည့်သည်

၂။ PQ မှ အသေးဆုံး (1, C) ကို ထုတ်ယူသည် ( C အား Finalize လုပ်သည် ):
   - Edge C->D (weight 5): 1 + 5 = 6 ( သို့သော် dist[D] = 2 ဖြစ်ပြီးသား မို့ update မလုပ်ပါ )

၃။ PQ မှ အသေးဆုံး (2, D) ကို ထုတ်ယူသည် ( D အား Finalize လုပ်သည် ):
   - Edge D->B (weight 5): 2 + 5 = 7 ( သို့သော် dist[B] = 4 ဖြစ်ပြီးသား မို့ update မလုပ်ပါ )

၄။ PQ မှ (4, B) ကို ထုတ်ယူသည် ( B အား Finalize လုပ်သည် ):
   - ဆက်လက် လုပ်ဆောင်စရာ edge မရှိပါ။

ရလဒ် အတိုဆုံး အကွာအဝေးများ: dist = {A: 0, B: 4, C: 1, D: 2}
```

Practical Problem: Network Delay Time

Node `n` ခု (`1` မှ `n` အထိ) နှင့် ရောက်ရှိရန် ကြာမြင့်ချိန်များ ပါဝင်သော directed weighted edges `times` (`[u, v, w]` = u မှ v သို့ ရောက်ရန် w စက္ကန့်) ကို ပေးထားသည်။ စတင်မှတ် Node `k` မှ စတင်၍ ကွန်ရက်အတွင်းရှိ Node **အားလုံး** သို့ သတင်းအချက်အလက် (signal) ရောက်ရှိရန် **စုစုပေါင်း ဘယ်နှစ် စက္ကန့်** ကြာမြင့်မည်နည်း။ Node အားလုံးသို့ မရောက်ရှိနိုင်ပါက `-1` ကို ပြန်ပေးပါ။

```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2
( Explanation: 2->1 သို့ 1s ကြာသည်၊ 2->3->4 သို့ 2s ကြာသည်။ signal သည် တစ်ပြိုင်နက် ပျံ့နှံ့သဖြင့် Node
အားလုံး ရောက်ရန် အဝေးဆုံး Node ၏ အချိန်ဖြစ်သော 2s ကြာမြင့်မည် ဖြစ်သည် )
```

Java Solution

```
import java.util.*;
```

```java
class Solution {
    public int networkDelayTime(int[][] times, int n, int k) {
        // Adjacency list ဆောက်လုပ်ခြင်း: u -> List of (v, w)
        List<int[]>[] graph = new List[n + 1];
        for (int i = 1; i <= n; i++) graph[i] = new ArrayList<>();
        for (int[] t : times) {
            graph[t[0]].add(new int[]{t[1], t[2]});
        }

        int[] dist = new int[n + 1];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[k] = 0;

        // Min-Heap: (distance, node) အလိုက် စီစဉ်ထားသည်
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> Integer.compare(a[0],
b[0]));
        pq.offer(new int[]{0, k});

        while (!pq.isEmpty()) {
            int[] cur = pq.poll();
            int d = cur[0], u = cur[1];

            // ဟောင်းနွမ်းနေသော (ပိုမိုကြီးမားသော distance ရှိသော) entry အား ကျော်လွှန်ခြင်း
            if (d > dist[u]) continue;

            for (int[] edge : graph[u]) {
                int v = edge[0], weight = edge[1];
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.offer(new int[]{dist[v], v});
                }
            }
        }

        // Node အားလုံးထဲတွင် အဝေးဆုံး ရောက်ရှိသည့် အချိန်အား ရှာဖွေခြင်း
        int maxDelay = 0;
        for (int i = 1; i <= n; i++) {
            if (dist[i] == Integer.MAX_VALUE) return -1; // မရောက်နိုင်သော node ရှိပါက -1
            maxDelay = Math.max(maxDelay, dist[i]);
        }
        return maxDelay;
    }
}
```

> **Complexity Analysis:**
> - **Time Complexity:** $O(E \log V)$ — Edge တစ်ခုစီသည် Min-Heap ထဲသို့ ဝင်/ထွက် လုပ်ဆောင်ပြီး Min-Heap operation တိုင်းသည် $O(\log V)$ ကြာမြင့်သောကြောင့် ဖြစ်ပါတယ်။
> - **Space Complexity:** $O(V + E)$ — Graph ကို Adjacency List ဖြင့် သိမ်းဆည်းခြင်းနှင့် Heap memory တို့အတွက် ဖြစ်ပါတယ်။

A* (A-star) Search — Goal-directed Pathfinding

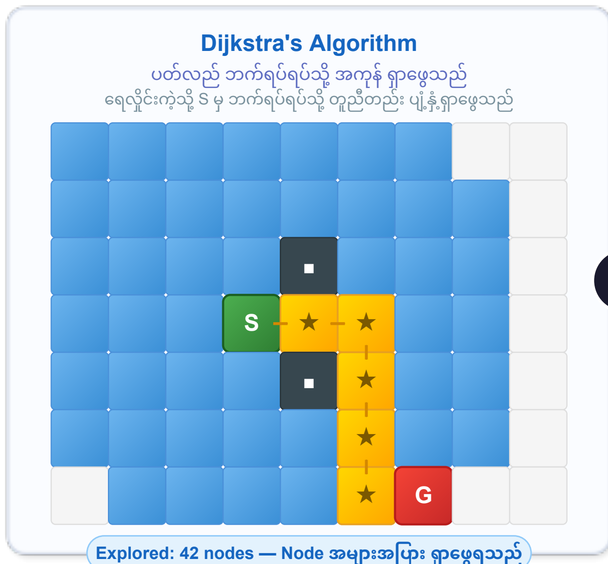
Dijkstra နှင့် A* Search တို့၏ ကွာခြားချက်

Dijkstra's algorithm သည် စတင်မှတ်မှ ပတ်ဝန်းကျင် အရပ်ရပ်သို့ **ရေလှိုင်းများ ပျံ့နှံ့သွားသကဲ့သို့** တညီတည်း ကျယ်ပြန့်စွာ ရှာဖွေသွားလေ့ ရှိပါတယ်။ ဒါဟာ "ပန်းတိုင် (Goal) က ဘယ်နေရာမှာ ရှိမှန်း မသိသေးတဲ့" ပြဿနာတွေမှာ သင့်တော်ပေမယ့်၊ မြေပုံပေါ်မှာ Goal ရဲ့ တည်နေရာကို **ကြိုတင်သိရှိထားသည့် Grid သို့မဟုတ် Game Pathfinding** ပြဿနာများတွင် အလဟဿ ရှာဖွေမှုများ ပိုမို များပြားစေပါတယ်။

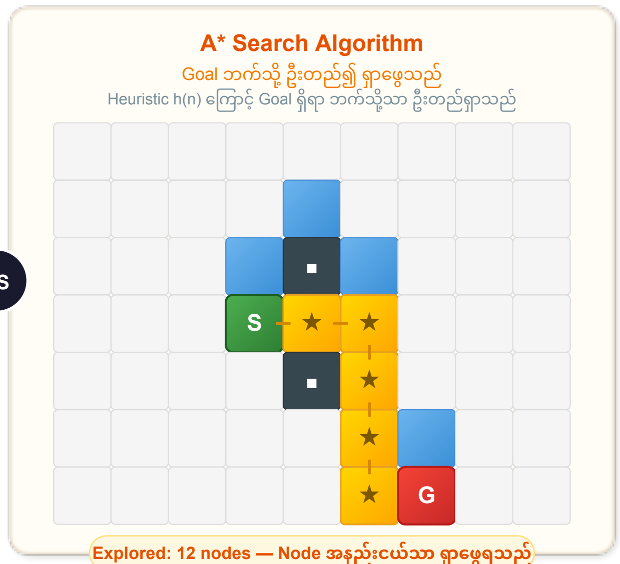
A* Search Algorithm သည် Dijkstra ကို ပိုမို အဆင့်မြှင့်တင်ထားခြင်း ဖြစ်ပြီး၊ Node တစ်ခုကို ရွေးချယ်သည့်အခါ "စတင်မှတ်မှ လက်ရှိနေရာသို့ ကုန်ကျစရိတ်" တစ်ခုတည်းကို မကြည့်ဘဲ၊ "**ဒီနေရာမှ ပန်းတိုင်ဆီသို့ ဘယ်လောက် ကျန်ရှိဦးမည်နည်း**" ဆိုသည့် **ခန့်မှန်းချက် (Heuristic function)** ကိုပါ ထည့်သွင်း တွက်ချက်ကာ Goal ရှိရာ ဘက်သို့ ဦးတည်၍ အမြန်ဆုံး ရှာဖွေပေးပါတယ်။

Dijkstra vs A* ရှာဖွေပုံ နှိုင်းယှဉ်ချက်

S — Start (စတင်မှတ်) | G — Goal (ပန်းတိုင်) | ■ — Wall (အတားအဆီး)
Explored (ရှာဖွေပြီး) | ★ Shortest Path (အတိုဆုံး) | · — Unexplored (မရှာဖွေရသေး)

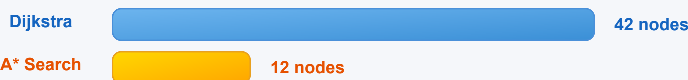


VS



★ အတိုဆုံး လမ်းကြောင်း (Shortest Path) နှစ်ခုလုံး အတူတူပင် ဖြစ်သော်လည်း

A* သည် ရှာဖွေရသည့် Node အရေအတွက် သိသိသာသာ နည်းပါးသောကြောင့် Dijkstra ထက် ပိုမို မြန်ဆန်ပါသည်။



💡 **အဓိက Insight**

- Dijkstra သည် g(n) တစ်ခုတည်းဖြင့် priority ထားပြီး ဘက်ရပ်ရပ်သို့ တူညီတည်း ပျံ့နှံ့ရှာဖွေသည်။
- A* သည် f(n) = g(n) + h(n) ဖြင့် Goal ဘက်သို့ ဦးတည်ရှာဖွေသဖြင့် မလိုအပ်သော Node များကို ကျော်လွှန်နိုင်သည်။
- h(n) = 0 ထားလိုက်ပါက A* သည် Dijkstra အဖြစ် ပြန်ပြောင်းသွားပါသည်။ ∴ Dijkstra သည် A* ၏ special case ဖြစ်သည်။

နှစ်ခုလုံးက Min-Heap + Relaxation အခြေခံပုံ တူပေမယ့်၊ Heap ထဲက Node တစ်ခုကို ရွေးတဲ့ **priority value** နဲ့ **ရှာဖွေတဲ့ ဦးတည်ချက်** ကသာ ကွာခြားပါတယ်။

အချက်	Dijkstra	A* Search

Priority value	$g(n)$ — Start မှ လက်ရှိ Node အထိ တကယ်ကုန်တဲ့ စရိတ်	$f(n) = g(n) + h(n)$ — စရိတ် + Goal အထိ ခန့်မှန်းခြေ
ရှာဖွေဦးတည်ချက်	ဘက်မရွေး၊ စက်ဝိုင်းပြန့် အကုန် တူညီ တည်း ပျံ့နှံ့	Goal ဘက်သို့သာ ဦးတည်၊ အချိုးကျ ဦးစားပေး
လိုအပ်ချက်	Edge weight ≥ 0	Edge weight ≥ 0 နဲ့ Goal တည်နေရာ ကြိုတင်သိရမယ်
Heuristic $h(n)$	မသုံး (သို့မဟုတ် $h = 0$)	Admissible $h(n)$ မဖြစ်မနေ ထည့်ရ
Explore လုပ်တဲ့ Node အရေအတွက်	များပါတယ် (ရှိသမျှ လမ်းကြောင်း အနှံ့)	နည်းပါးပါတယ် (Goal ဆီ တည့်တည့်)
Optimal (အတိုဆုံး) အဖြေ သေချာလား	အမြဲ သေချာ	$h(n)$ Admissible ဖြစ်ရင် သေချာ
သင့်တော်တဲ့ ကိစ္စ	Goal နေရာ မသိရင်၊ Multiple targets အားလုံး	Goal နေရာ သိရင်၊ တစ်ဦး တစ်ယောက်တည်း

> **အဓိက Insight:** $h(n) = 0$ ထားလိုက်ရင် A *ဟာ Dijkstra အဖြစ်ပြန်ပြောင်းသွားပါတယ်။ ဒါ ကြောင့် Dijkstra က A* ၏ အထူးပြု (special case) တစ်ခုဟု မှတ်ယူနိုင်ပါတယ်။

$f(n) = g(n) + h(n)$ အဓိက ညီမျှခြင်း

A* Search တွင် Node တစ်ခုစီအတွက် တန်ဖိုး ၃ ခုကို စက်ဝိုင်းလည် တွက်ချက်ပါတယ်။

- $g(n)$**:** Start Node မှ လက်ရှိ Node `n` သို့ ရောက်ရှိရန် **တကယ် ကုန်ကျခဲ့သည့် စရိတ်** (Dijkstra ၏ `dist` တန်ဖိုး ဖြစ်သည်)။
- $h(n)$ **(Heuristic):** လက်ရှိ Node `n` မှ Goal Node သို့ ရောက်ရှိရန် **ခန့်မှန်းခြေ ကုန်ကျမည့် စရိတ်**။
- $f(n) = g(n) + h(n)$**:** လက်ရှိ Node `n` ကို ဖြတ်သန်းသွားပါက စုစုပေါင်း ကုန်ကျနိုင်မည့် ခန့်မှန်းစရိတ်။

Min-Heap ထဲတွင် $g(n)$ အစား $f(n)$ **တန်ဖိုး အသေးဆုံး** ရှိသော Node ကို ဦးစားပေး ဆွဲထုတ် လုပ်ဆောင်သောကြောင့် ပန်းတိုင်သို့ ထိရောက်စွာ အမြန်ဆုံး ရောက်ရှိစေပါတယ်။

Admissible Heuristic ဆိုတာဘာလဲ

A* Search မှ ပေးသော အဖြေသည် အတိုဆုံးနှင့် အမှန်ကန်ဆုံး (optimal) ဖြစ်စေရန်အတွက် Heuristic function $h(n)$ သည် **Admissible** ဖြစ်ရပါမည်။ Admissible ဖြစ်တယ်ဆိုသည်မှာ ခန့်မှန်းချက် $h(n)$ သည် တကယ့် အကွာအဝေးထက် **မည်သည့်အခါမျှ ပိုမို မကြီးမားရပါ (Never overestimate)**။

2D Grid Pathfinding များတွင် အသုံးများသော Admissible Heuristics များမှာ

- **Manhattan Distance:** $|r_1 - r_2| + |c_1 - c_2|$ (၄ ဖက် ဓာတ်ဆန့်ကျင် ရွေ့လျားနိုင်ပြီး move တိုင်း cost = 1 ဖြစ်လျှင်)

- **Chebyshev Distance:** $\max(|r_1 - r_2|, |c_1 - c_2|)$ (၈ ဖက် ထောင့်ဖြတ်ပါ ရွေ့လျားနိုင်ပြီး move တိုင်း cost = 1 ဖြစ်လျှင်)
- **Euclidean Distance:** $\sqrt{(r_1 - r_2)^2 + (c_1 - c_2)^2}$ (ပကတိ ထောင့်ဖြတ် အကွာအဝေး cost = $\sqrt{2}$ ဖြစ်လျှင်)

Practical Problem: Shortest Path in Binary Matrix (A* Implementation)

`m × n` grid တစ်ခုတွင် `0` သည် သွားလာနိုင်သော လမ်းကြောင်းဖြစ်ပြီး `1` သည် အတားအဆီး ဖြစ်သည်။ ဘယ်အပေါ်ထောင့် `(0,0)` မှ ညာအောက်ထောင့် `(m-1,n-1)` သို့ ၄ ဖက် ရွေ့လျား၍ ရောက်ရှိနိုင်သော **အတိုဆုံး အလှမ်း အရေအတွက်** ကို A* Search သုံး၍ ရှာဖွေပါ။

Java Solution

```
import java.util.*;

public class AStarGrid {
    public int shortestPathAStar(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        if (grid[0][0] == 1 || grid[m - 1][n - 1] == 1) return -1;

        int targetR = m - 1, targetC = n - 1;

        // gScore[r][c]: Start မှ (r,c) သို့ တကယ် ရောက်ရှိခဲ့သော အလှမ်း အရေအတွက်
        int[][] gScore = new int[m][n];
        for (int[] row : gScore) Arrays.fill(row, Integer.MAX_VALUE);
        gScore[0][0] = 0;

        // Min-Heap: (fScore, r, c)
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> Integer.compare(a[0],
b[0]));

        // f = g + h -> 0 + Manhattan Distance
        int startH = Math.abs(0 - targetR) + Math.abs(0 - targetC);
        pq.offer(new int[]{startH, 0, 0});

        int[][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

        while (!pq.isEmpty()) {
            int[] cur = pq.poll();
            int r = cur[1], c = cur[2];

            if (r == targetR && c == targetC) return gScore[r][c];

            for (int[] d : dirs) {
                int nr = r + d[0], nc = c + d[1];
                if (nr >= 0 && nr < m && nc >= 0 && nc < n && grid[nr][nc] == 0) {
                    int tentG = gScore[r][c] + 1;
                    if (tentG < gScore[nr][nc]) {
                        gScore[nr][nc] = tentG;
                        int h = Math.abs(nr - targetR) + Math.abs(nc - targetC);
                        pq.offer(new int[]{tentG + h, nr, nc});
                    }
                }
            }
        }
        return -1;
    }
```

```
}
```

> **Complexity Analysis:**
> - **Time Complexity:** Worst-case တွင် $O(V \log V)$ (Dijkstra နှင့် တူညီသည်) ဖြစ်သော်လည်း လက်တွေ့တွင် ကောင်းမွန်သော Heuristic ကြောင့် ရှာဖွေရသည့် Node အရေအတွက်ကို မြောက်များစွာ လျှော့ချပေးပါသည်။
> - **Space Complexity:** $O(V)$ — Grid matrix တန်ဖိုးများ နှင့် Heap memory အတွက် ဖြစ်ပါသည်။

Bellman-Ford Algorithm — Negative Edge Weights

Dijkstra ၏ အားနည်းချက်နှင့် Bellman-Ford

Dijkstra's Algorithm သည် Greedy ချဉ်းကပ်မှုကို သုံးထားသဖြင့် Edge weight များတွင် **အနုတ်ကိန်း (Negative weight)** ပါဝင်လာပါက အဖြေမှားယွင်းသွားတတ်ပါတယ်။ အကြောင်းမှာ Dijkstra သည် Node တစ်ခုကို Finalize လုပ်ပြီးပါက ပြန်လည် စစ်ဆေးခြင်း မရှိတော့ပေ။ သို့သော် Negative weight ရှိနေပါက Finalize လုပ်ပြီးသား Node သို့ နောက်မှ ပိုမို သက်သာသော လမ်းကြောင်း ရောက်ရှိလာနိုင်သောကြောင့် ဖြစ်ပါတယ်။

ဒီလို Negative weight များ ပါဝင်သော Graph များအတွက် **Dynamic Programming** သဘောတရားကို အခြေခံထားသည့် **Bellman-Ford Algorithm** ကို သုံးရပါတယ်။

$V - 1$ ကြိမ် Relaxation လုပ်ဆောင်ခြင်း

Vertex အရေအတွက် `V` ခု ရှိသော Graph တစ်ခုတွင် Cycle မပါဝင်သော အတိုဆုံး လမ်းကြောင်း တစ်ခု၌ အများဆုံး ပါဝင်နိုင်သည့် Edge အရေအတွက်သည် **`V - 1` ခု** ဖြစ်ပါတယ်။

Bellman-Ford Algorithm သည် Graph အတွင်းရှိ **Edge အားလုံးကို `V - 1` ကြိမ်တိုင်တိုင် ထပ်ခါထပ်ခါ Relax လုပ်ဆောင်ပေးခြင်း** ဖြင့် အတိုဆုံး လမ်းကြောင်းကို ရှာဖွေပေးပါတယ်။

```
for i = 1 to V - 1:
    for each edge (u, v, weight):
        if dist[u] + weight < dist[v]:
            dist[v] = dist[u] + weight
```

Negative Cycle Detection

အကယ်၍ `V - 1` ကြိမ်မကဘဲ `V` **ကြိမ်မြောက် ထပ်မံ Relax လုပ်သည့်အခါတွင်ပင် တန်ဖိုးများ ထပ်မံ လျော့နည်းသွားသေးပါက** ထို Graph ၌ **Negative Cycle (လည်ပတ်လေ အကွာအဝေး လျော့နည်းလေဖြစ်သော Cycle)** ရှိနေကြောင်း အတည်ပြုနိုင်ပါတယ်။ Negative Cycle ရှိနေပါက အတိုဆုံး လမ်းကြောင်း တန်ဖိုးသည် $-\infty$ သို့ ဦးတည်သွားမည် ဖြစ်၍ အဖြေရှာရန် မဖြစ်နိုင်တော့ပါ။

> **Dijkstra vs Bellman-Ford:**
> - **Dijkstra:** Edge weights ≥ 0 သာ လက်ခံသည်၊ မြန်ဆန်သည် $O(E \log V)$။
> - **Bellman-Ford:** Negative edge weight များ ပါဝင်နိုင်သည်၊ Negative Cycle ကို စစ်ဆေးပေးနိုင်သည်၊ ပိုမိုနှေးကွေးသည် $O(V \times E)$။

Floyd-Warshall Algorithm — All-Pairs Shortest Path

အခြေခံ သဘောတရား

Dijkstra နှင့် Bellman-Ford တို့သည် Node တစ်ခုတည်းမှ ကျန် Node များသို့ အတိုဆုံး လမ်းကြောင်း (Single-Source) ကို ရှာဖွေပေးခြင်း ဖြစ်ပါတယ်။ သို့သော် **Node တိုင်းမှ ကျန် Node တိုင်းသို့ (All-Pairs)** အတိုဆုံး လမ်းကြောင်း ဇယားတစ်ခုလုံး သိရှိလိုပါက **Floyd-Warshall Algorithm** ကို သုံးရပါတယ်။

Floyd-Warshall ၏ အဓိက သဘောတရားမှာ **Intermediate Node DP** ဖြစ်ပါတယ်။
"Node `i` မှ `j` သို့ သွားရာတွင် ကြားခံ Node `k` ကို ဖြတ်သန်း ခွင့်ပြုလိုက်ပါက ပိုမို သက်သာသော လမ်းကြောင်း ရှိလာမည်လား" ဆိုသည်ကို ကြားခံ Node `k` တစ်ခုစီအတွက် တွက်ချက်သွားခြင်း ဖြစ်ပါတယ်။

```
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

Java Implementation

```java
public class FloydWarshall {
    public void floydWarshall(int V, int[][] graph) {
        int[][] dist = new int[V][V];

        // ဇယားအား စတင် ပြင်ဆင်ခြင်း
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                dist[i][j] = graph[i][j];
            }
        }

        // Intermediate node k အား ကြားခံအဖြစ် ဖြတ်သန်းကြည့်ခြင်း
        for (int k = 0; k < V; k++) {
            for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                    if (dist[i][k] != Integer.MAX_VALUE && dist[k][j] !=
```

```
Integer.MAX_VALUE) {
                            dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                        }
                    }
                }
            }
        }
    }
```

> **Complexity Analysis:**
> - **Time Complexity:** $O(V^3)$ — 3-nested loops သုံးထားသောကြောင့် ဖြစ်ပြီး Vertex အရေအတွက် အနည်းငယ် (ဥပမာ V ≤ 400) ရှိသော Graph များအတွက် အလွန် သင့်တော်ပါသည်။
> - **Space Complexity:** $O(V^2)$ — 2D distance matrix အတွက် ဖြစ်ပါသည်။

Union-Find (Disjoint-Set) Data Structure

အခြေခံ သဘောတရားနှင့် Operations

Union-Find (သို့မဟုတ် Disjoint-Set Union - DSU) ဆိုသည်မှာ အစိတ်အပိုင်း (elements) အများအပြားကို ထပ်တူမကျသော အုပ်စု (disjoint sets) များအဖြစ် ခွဲခြား သိုလှောင်ထားပြီး၊ အောက်ပါ အဓိက Operations နှစ်ခုကို အလွန် မြန်ဆန်စွာ လုပ်ဆောင်ပေးနိုင်သည့် Data Structure ဖြစ်ပါတယ် —

1. `find(x)` : Element `x` ပါဝင်သော အုပ်စု၏ ပင်မ ကိုယ်စားလှယ် **Root** ကို ရှာဖွေပေးသည်။
2. `union(x, y)` : Element `x` နှင့် `y` တို့ ပါဝင်သော အုပ်စု နှစ်ခုကို အုပ်စု တစ်ခုတည်းအဖြစ် **ပေါင်းစည်း** ပေးသည်။

```
union(1,2), union(3,4), union(2,4) လုပ်ဆောင်ပြီးနောက် သစ်ပင် ပုံသဏ္ဌာန်:

      1                 ( Root 1 သည် အုပ်စုတစ်ခုလုံး၏ ကိုယ်စားလှယ် ဖြစ်သည် )
     / \
    2   3
        |
        4
find(4) -> 4 -> 3 -> 1 (Root = 1)
find(2) -> 2 -> 1      (Root = 1)  ⇒  2 နှင့် 4 သည် အုပ်စုတူညီကြသည် ✓
```

Optimizations: Path Compression နှင့် Union by Rank

ရိုးရိုး Union-Find သည် သစ်ပင် ပုံသဏ္ဌာန် အောက်သို့ ရှည်လျားသွားပါက $O(n)$ အထိ နှေးကွေးသွားနိုင်ပါတယ်။ ဒါကို ကာကွယ်ဖို့ အရေးကြီးသော Optimization နည်းလမ်း နှစ်ခုကို တွဲဖက် သုံးရပါတယ် —

- **Path Compression (`find` တွင် သုံးသည်):** Root ကို ရှာဖွေသည့် လမ်းကြောင်းပေါ်ရှိ Node အားလုံးကို Root သို့ တိုက်ရိုက် ချိတ်ဆက်ပေးလိုက်ခြင်းဖြင့် သစ်ပင်၏ အမြင့်ကို ပြားသွားစေပါသည်။
- **Union by Rank (`union` တွင် သုံးသည်):** အုပ်စု နှစ်ခု ပေါင်းစပ်သည့်အခါ အမြင့်နိမ့်သော (Rank နည်းသော) သစ်ပင်ကို အမြင့်မြင့်သော သစ်ပင်၏ အောက်သို့သာ သွားရောက် ချိတ်ဆက်စေပါသည်။

ဒီ Optimization နှစ်ခုကို သုံးလိုက်ပါက Operation တစ်ခု၏ အချိန်ကြာမြင့်ချိန်သည် $O(\alpha(n))$ (Inverse Ackermann function) သို့ ရောက်ရှိသွားပြီး လက်တွေ့တွင် $O(1)$ နီးပါး အလွန် မြန်ဆန်သွားပါတယ်။

Practical Problem: Redundant Connection (Cycle Detection)

Node `n` ခု ပါဝင်သော undirected tree တွင် edge တစ်ခု ပိုမို ထည့်သွင်းလိုက်သဖြင့် cycle ဖြစ်ပေါ်သွားသည်။ ပေးထားသော edge များထဲမှ ဖယ်ရှားလိုက်ပါက မူလ tree ပြန်လည် ဖြစ်ပေါ်စေမည့် redundant edge ကို Union-Find သုံး၍ ရှာဖွေပါ။

Java Solution

```
class UnionFind {
    int[] parent, rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    public int find(int i) {
        if (parent[i] != i) {
            parent[i] = find(parent[i]); // Path Compression
        }
        return parent[i];
    }

    public boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) return false; // အုပ်စု တူညီပြီးသားဖြစ်၍ cycle ဖြစ်ပေါ်စေသည်

        // Union by Rank
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        return true;
```

```
        }
    }

    class SolutionRedundant {
        public int[] findRedundantConnection(int[][] edges) {
            int n = edges.length;
            UnionFind uf = new UnionFind(n + 1);

            for (int[] edge : edges) {
                if (!uf.union(edge[0], edge[1])) {
                    return edge; // Union မဖြစ်ဘဲ cycle ဖြစ်စေသော redundant edge အား ပြန်ပေးခြင်း
                }
            }
            return new int[0];
        }
    }
```

> **Complexity Analysis:**
> - **Time Complexity:** $O(N \cdot \alpha(N)) \approx O(N)$ — Edge တစ်ခုစီကို Union-Find လုပ်ဆောင်ခြင်း ဖြစ်ပါတယ်။
> - **Space Complexity:** $O(N)$ — Parent နှင့် Rank array များအတွက် ဖြစ်ပါတယ်။

Minimum Spanning Tree (MST)

Spanning Tree နှင့် Cut Property

Weighted undirected graph တစ်ခုတွင် Vertex အားလုံးကို ဆက်စပ်မှု ရှိစေရန် (connected ဖြစ်စေရန်) Edge များကို ရွေးချယ်ရာ၌ **Edge weight စုစုပေါင်း အနည်းဆုံး ဖြစ်အောင်** ရွေးချယ်ထားသော သစ်ပင် structure ကို **Minimum Spanning Tree (MST)** ဟု ခေါ်ပါတယ်။ Node `n` ခု ပါဝင်သော Graph တစ်ခု၏ MST တွင် Cycle မပါဝင်ဘဲ Edge အတိအကျ **`n - 1` ခု** ပါဝင်ရပါမည်။

MST ကို ရှာဖွေရာတွင် Greedy logic မှန်ကန်ကြောင်း သက်သေပြသည့် အဓိက သီအိုရီမှာ **Cut Property** ဖြစ်ပါတယ် — "Graph အား အုပ်စု နှစ်ခု ခွဲခြားကြည့်ပါက ထိုအုပ်စု နှစ်ခုကြားကို ချိတ်ဆက်ပေးသော Edge များထဲမှ **အကွာအဝေး အနည်းဆုံး (အသက်သာဆုံး) Edge** သည် MST ၏ အစိတ်အပိုင်း ဖြစ်ရမည်"။

MST ရှာဖွေရန် နာမည်ကျော် Algorithm နှစ်ခု ရှိပါတယ် — **Kruskal's Algorithm** နှင့် **Prim's Algorithm**။

Kruskal's Algorithm vs Prim's Algorithm

1. **Kruskal's Algorithm (Edge-centric):**
   - Graph အတွင်းရှိ Edge အားလုံးကို Edge weight အလိုက် ငယ်စဉ်မှ ကြီးစဉ် စီစဉ်လိုက်ပါတယ်။
   - Weight အနည်းဆုံး Edge မှ စတင်၍ Cycle မဖြစ်ပေါ်ပါက (Union-Find ဖြင့် စစ်ဆေး၍) MST ထဲသို့ ထည့်သွင်းပါတယ်။ Edge အရေအတွက် `n - 1` ခု ပြည့်ပါက ရပ်တန့်ပါတယ်။
   - Sparse Graph (Edge အရေအတွက် နည်းသော Graph) တွင် အသုံးပြုရန် ပိုမို သင့်တော်ပါသည်။

1. **Prim's Algorithm (Node-centric):**
   - Node တစ်ခုမှ စတင်၍ လက်ရှိ MST သစ်ပင်နှင့် ချိတ်ဆက်နိုင်သော Edge များထဲမှ အသက်သာဆုံး Edge ကို Min-Heap သုံး၍ ရွေးချယ် ပေါင်းစည်းသွားခြင်း ဖြစ်ပါတယ်။
   - Dense Graph (Edge အရေအတွက် များပြားသော Graph) တွင် အသုံးပြုရန် ပိုမို သင့်တော်ပါသည်။

Practical Problem: Min Cost to Connect All Points (Kruskal's MST)

2D plane ပေါ်ရှိ coordinates `points` ကို ပေးထားသည်။ Point နှစ်ခုကြား ချိတ်ဆက်မှု စရိတ်သည် Manhattan distance (`|x1 - x2| + |x1 - x2|`) ဖြစ်သည်။ Point **အားလုံး** ချိတ်ဆက်မိစေရန် ကုန်ကျမည့် **အနည်းဆုံး စရိတ်** ကို Kruskal's Algorithm ဖြင့် ရှာဖွေပါ။

Java Solution

```
import java.util.*;

class SolutionMST {
    public int minCostConnectPoints(int[][] points) {
        int n = points.length;
        List<int[]> edges = new ArrayList<>();

        // Point အားလုံးကြားရှိ Edge များနှင့် Manhattan Distance များကို တွက်ချက်ခြင်း
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int dist = Math.abs(points[i][0] - points[j][0]) + Math.abs(points[i][1]
- points[j][1]);
                edges.add(new int[]{dist, i, j});
            }
        }

        // Edge များကို Weight အလိုက် ငယ်စဉ်မှ ကြီးစဉ် စီစဉ်ခြင်း
        edges.sort((a, b) -> Integer.compare(a[0], b[0]));

        UnionFind uf = new UnionFind(n);
        int mstCost = 0;
        int edgesUsed = 0;

        for (int[] edge : edges) {
            int weight = edge[0];
            int u = edge[1];
            int v = edge[2];
```

```
                if (uf.union(u, v)) {
                    mstCost += weight;
                    edgesUsed++;
                    if (edgesUsed == n - 1) break; // Edge n - 1 ခု ပြည့်ပါက MST ရရှိပြီ ဖြစ်သည်
                }
            }
            return mstCost;
        }
    }
```

> **Complexity Analysis:**
> - **Time Complexity:** $O(N^2 \log(N^2)) = O(N^2 \log N)$ — Point အားလုံးကြား Edge စုစုပေါင်း N^2 ခုအား Sorting လုပ်ဆောင်ရသောကြောင့် ဖြစ်ပါတယ်။
> - **Space Complexity:** $O(N^2)$ — Edge List သိမ်းဆည်းရန် ဖြစ်ပါတယ်။

Real-world Applications Summary

ယခုအခန်း၌ လေ့လာခဲ့သော Advanced Graph Algorithm များသည် သီအိုရီ သက်သက် မဟုတ်ဘဲ Software Engineering နယ်ပယ် အသီးသီးတွင် အဓိက အုတ်မြစ်အဖြစ် ပါဝင်နေပါတယ် —

1. **Package Management & Build Systems (Topological Sort):** npm, pip သို့မဟုတ် Gradle တို့တွင် Dependency အစဉ်လိုက် Install / Compile လုပ်ဆောင်နိုင်ရန် သုံးစွဲသည်။
2. **GPS Routing & Navigation (Dijkstra & A* Search):** Google Maps, Apple Maps သို့မဟုတ် Game AI Pathfinding တွင် အတိုဆုံး သွားလာနိုင်မည့် လမ်းကြောင်း ရှာဖွေရာ၌ သုံးစွဲသည်။
3. **Network Routing Protocols (Dijkstra & Bellman-Ford):** Internet Router များအကြား Packet ဒေတာများ အမြန်ဆုံးနှင့် အမှန်ကန်ဆုံး ကူးပြောင်းနိုင်ရန် သုံးစွဲသည်။
4. **Infrastructure Grid Design (Minimum Spanning Tree):** လျှပ်စစ်ဓါတ်အားလိုင်းများ၊ ရေပိုက်လိုင်းများနှင့် ကွန်ရက် Cable လိုင်းများကို စရိတ် အနည်းဆုံးဖြင့် လွှမ်းခြုံ ချိတ်ဆက်နိုင်ရန် သုံးစွဲသည်။
5. **Social Network & Image Processing (Union-Find):** Social media ပလက်ဖောင်းများတွင် သူငယ်ချင်း အုပ်စုများ ခွဲခြားခြင်းနှင့် Image Segmentation တွင် အရာဝတ္ထုများ အုပ်စုဖွဲ့ခြင်းတို့၌ သုံးစွဲသည်။

အခန်း ၂၀ - Greedy

နေ့စဉ်ဘဝမှာ ကျွန်တော်တို့ ဆုံးဖြတ်ချက် အများစုကို "ရှေ့ ဘယ်လောက် ဖြစ်လာမလဲ" အကုန် မတွက်ဘဲ — **အခု အချိန်မှာ အကောင်းဆုံး ထင်ရတာ** ကို ရွေးပြီး ဆက်သွားကြတာပါ။ ဈေးဝယ်ထွက်ရင် အသက်သာဆုံး ဆိုင်ကို ဝင်တယ်၊ အကြွေ ပြန်အမ်းရင် အကြီးဆုံး အရွက်ကို ပြန် ထုတ်ပေးတယ်။ ဒီလို "လောလောဆယ် အကောင်းဆုံးကို ရွေး" တဲ့ ချဉ်းကပ်နည်းကို algorithm ဘာသာစကားနဲ့ **Greedy** လို့ ခေါ်ပါတယ်။

အခန်း ၁၉ က Kruskal, Prim တွေဟာ "အခု **အသက်သာဆုံး edge**" ကို ရွေး၊ Dijkstra ကတော့ "start ကနေ **distance အသေးဆုံး vertex**" ကို ရွေးပြီး သူ့ edge တွေကို relax — အကုန်လုံး greedy algorithm တွေပါ။ တစ်ဆင့်ချင်း "လောလောဆယ် အကောင်းဆုံး" ကို ရွေးသွားတာ။ ဒီအခန်းမှာတော့ greedy ကို pattern တစ်ခု အဖြစ် သီးခြား လေ့လာပါမယ်။

Greedy ရဲ့ idea က ရိုးရှင်းပေမယ့် — **အန္တရာယ်** တစ်ခု ရှိတယ်။ "အခု အကောင်းဆုံး ရွေးတိုင်း — အဆုံးမှာ အကောင်းဆုံး ရတယ်" လို့ **အာမ မခံနိုင်**ပါ။ တစ်ခါတစ်ရံ အခုလောလောဆယ် နည်းနည်း နစ်နာပြီး ရွေးလိုက်တာက — အဆုံးမှာ ပိုကောင်းတဲ့ အဖြေ ပေးတတ်တယ်။ ဒါကြောင့် greedy ကို သင်တဲ့အခါ — code ရေးနည်းထက် **"ဒီ ပြဿနာမှာ greedy တကယ် မှန်ရဲ့လား"** ဆိုတဲ့ မေးခွန်းက ပို အရေးကြီးပါတယ်။ ဒီအခန်းမှာ — greedy ဘယ်လို အလုပ်လုပ်လဲ၊ ဘယ်တော့ မှန်ပြီး ဘယ်တော့ ကျိုးလဲ၊ pattern ၂ မျိုးနဲ့ ဘယ်လို သုံးလဲ၊ ပြီးတော့ classic ပြဿနာ ၅ ခုကို လေ့လာသွားပါမယ်။

Greedy ဆိုတာ ဘာလဲ — Local vs Global

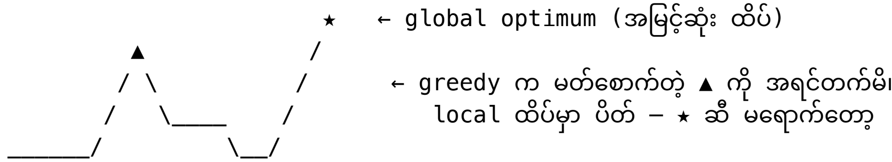
Greedy algorithm ရဲ့ အဓိက လုပ်ဆောင်ချက်ကို **greedy choice** လို့ ခေါ်တယ် — ဆုံးဖြတ်ချက် တစ်ဆင့်စီမှာ၊ ရှေ့မှာ ဘာဆက်ဖြစ်မလဲ မတွက်ဘဲ၊ **အခု အကောင်းဆုံး ထင်ရတဲ့ option** ကို ရွေးပြီး ဆက်သွားတာ။ ရွေးပြီးသား ဆုံးဖြတ်ချက်ကို နောက်ပြန် မပြင်တော့ဘူး (ဒါက back-tracking နဲ့ ကွာတဲ့ အချက်ပါ)။

ဒီနေရာမှာ နားလည်ထားရမယ့် အရေးကြီး ၂ ခု က

- **Local optimum** — ဆုံးဖြတ်ချက် **တစ်ခုတည်း** အတွက် အကောင်းဆုံး ရွေးချယ်မှု (ဥပမာ — "အခု ခြေလှမ်းမှာ အမြင့်ဆုံး ဘက်ကို တက်")။
- **Global optimum** — ပြဿနာ **တစ်ခုလုံး** အတွက် အကောင်းဆုံး အဖြေ (ဥပမာ — "တောင်ထဲက အမြင့်ဆုံး ထိပ်ကို ရောက်")။

Greedy ရဲ့ အဓိက မေးခွန်းက — **"local optimum တွေ ဆက်တိုက် ရွေးသွားရင်၊ global optimum ဆီ ရောက်ရဲ့လား?"** ဆိုတာပါ။ တောင်တက်တဲ့ ဥပမာနဲ့ ကြည့်ရအောင် —

```
ပန်းတိုင်: အမြင့်ဆုံး ထိပ် ★ ကို ရောက်ချင်
greedy စည်းမျဉ်း: "အခု ခြေလှမ်းမှာ အတက်ဆုံး ဘက်ကို သွား"
```



ဒီဥပမာမှာ — "အခု အတက်ဆုံး ဘက် သွား" တဲ့ local choice က — ပိုနိမ့်တဲ့ တောင်ထိပ် (local optimum) မှာ ပိတ်စေပြီး၊ တကယ့် အမြင့်ဆုံး (global optimum) ★ ဆီ မရောက်ဖြစ်စေတယ်။ ဒါကြောင့် greedy ကို သုံးတိုင်း — **"ဒီ ပြဿနာမှာ local ရွေးတာ global ဆီ တကယ် ရောက်ရဲ့လား"** ကို သေချာ ဆုံးဖြတ် ဖို့ လိုပါတယ်။

ဘယ်တော့ Greedy အလုပ်ဖြစ်သလဲ

တောင်တက်တဲ့ ဥပမာက greedy က ရလဒ် မှန်ကန် မှု မရှိတာကို ပြတယ်။ ဒါဆို greedy ဘယ်တော့ **မှန်**သလဲ? ပြဿနာတစ်ခုမှာ အောက်က အချက် ၂ ခု ရှိမှ greedy က အဖြေ မှန်ပါတယ် —

1. **Greedy choice property** — အခု ရွေးလိုက်တဲ့ local optimum က — global optimal solution ထဲမှာ **အမြဲ ပါဝင်နေတယ်** ဆိုတာ သက်သေပြနိုင်ရမယ်။ တစ်နည်းပြောရင် — "အကောင်းဆုံး ရွေးချယ်မှု" ကြောင့် နောက်ပိုင်း နောင်တရစရာ မရှိ၊ ပြန်ပြင်စရာ မလို။
2. **Optimal substructure** — ရွေးချယ်မှု တစ်ခု လုပ်ပြီးတဲ့နောက် — ကျန်တဲ့ ပြဿနာက **ပိုငယ်တဲ့ ပြဿနာ တစ်ခု** ဖြစ်သွားပြီး၊ အဲ့ဒါကိုလည်း တူညီတဲ့ greedy နည်းနဲ့ ဆက်ဖြေလို့ ရရမယ်။

ဒီ ၂ ခု ရှိရင် greedy က Dynamic Programming (အခန်း ၂၂) လို possibility အကုန် မစစ်ဘဲ၊ တစ်ဆင့်ချင်း တန်းရွေးသွားလို့ **ပိုမြန်** (များသောအားဖြင့် $O(n)$ ဒါမှမဟုတ် $O(n \log n)$) နိုင် ပြီး memory လည်း သက်သာတယ်။ ဒါပေမယ့် အဲ့ ၂ ချက် မရှိရင် — greedy က မှားတဲ့ အဖြေ ပေးတတ်လို့၊ DP လို ပိုစေ့စပ်တဲ့ နည်းကို သုံးရတယ်။

When Greedy Fails — Coin Change ဥပမာ

Greedy မမှန်တာကို အရှင်းဆုံး ပြတဲ့ ဥပမာက အကြွေစေ့ ပြဿနာပါ။ အကြွေစေ့ အမျိုးအစား `[1, 3, 4]` (ကျပ်) ရှိတယ် ဆိုပါစို့ — `6` ကျပ်ကို **အကြွေစေ့ အရေအတွက် အနည်းဆုံး** နဲ့ ဖွဲ့ချင်တယ်။

```
Greedy ("အကြီးဆုံး စေ့ အရင် ရွေး"):
     6 → 4 ယူ → ကျန် 2 → 1 ယူ → ကျန် 1 → 1 ယူ → ကျန် 0
     ရလဒ်:  4 + 1 + 1  =  ၃ စေ့

Optimal (အမှန်):
     6 → 3 + 3  =  ၂ စေ့   ✓  (ပိုနည်း!)
```

Greedy က "အကြီးဆုံး 4 ကို အရင်ရွေး" တဲ့ local choice ကြောင့် — `3 + 3` ဆိုတဲ့ ပိုကောင်းတဲ့ အဖြေကို လွတ်သွားတယ်။ ဒီ denomination `[1,3,4]` မှာ greedy choice property မရှိလို့ — greedy က မမှန် တာပါ (ဒီလို ပြဿနာကို **DP** နဲ့ ဖြေရတယ် — အခန်း ၂၂)။

သိထားသင့်တာက — `[1, 5, 10, 25]` ဆိုတဲ့ ဒီ denomination set မှာတော့ greedy က မှန်ပါတယ် (ဒီ denomination တွေက greedy choice property ရှိနေလို့)။ ဒါပေမယ့် denomination set တိုင်းမှာ မှန်တာ မဟုတ်ဘူး — `[1,3,4]` လို set မျိုးမှာ မမှန် ပါဘူး။

> **greedy က ပြဿနာ အပေါ် မူတည်တယ်၊ သုံးခင် "မှန်မမှန်" အရင် စစ်ရမယ်။**

Greedy ရဲ့ Pattern ၂ မျိုး

Greedy choice က မှန်တယ် ဆုံးဖြတ်ပြီးရင် — နောက်တစ်ဆင့်က "အဲ့ choice ကို code ထဲ ဘယ်လို အမြန် ရွေးထုတ်မလဲ" ပါ။ Real-world greedy ပြဿနာ အများစုက အောက်က pattern ၂ ခုထဲက တစ်ခုနဲ့ ဖြေလို့ ရတယ်။

၁။ Sorting + Greedy

greedy choice က **အစကတည်းက ပုံသေ** (ဥပမာ — "အသေးဆုံး cookie အရင်"၊ "အစောဆုံး ပြီးတဲ့ meeting အရင်") ဆိုရင် — data ကို **သင့်တော်တဲ့ key နဲ့ တစ်ခါ sort** လုပ်လိုက်ရုံနဲ့ — အစဉ်လိုက် တစ်ခုချင်း ရွေးသွားလို့ ရတယ်။

ဒီ pattern ရဲ့ အဓိက အလုပ်က — **"ဘယ် key နဲ့ sort လုပ်ရင် greedy choice ရှင်းသွားလဲ"** ကို ရှာတာပါ။ ဥပမာ — meeting scheduling မှာ start အလိုက် sort လုပ်တာထက် **end အလိုက်** sort လုပ်တာက ပိုအလုပ်ဖြစ်တယ် ("အစောဆုံး ပြီးတာ အရင်ရွေးရင် နောက်အတွက် နေရာ အများဆုံး ကျန်" လို့)။ Time complexity က sort ကြောင့် $O(n \log n)$ ဖြစ်လေ့ ရှိတယ်။

၂။ Priority Queue + Greedy

greedy choice က **အမြဲ ပြောင်းနေ**တယ် ဆိုရင် (ဥပမာ — "အခု လက်ကျန် အများဆုံး task ကို run" — task တစ်ခု run ပြီးတိုင်း လက်ကျန် ပြောင်းသွား) — တစ်ခါတည်း sort လုပ်ထားလို့ မရတော့ဘူး။ အဲ့ဒီအခါ **min/max heap (priority queue)** ကို သုံးပြီး — အဆင့်တိုင်းမှာ "လက်ရှိ အကောင်းဆုံး" ကို ထိပ်ကနေ ဆွဲထုတ်၊ update လုပ်ပြီး ပြန်ထည့်တယ်။ heap operation တစ်ခုက $O(\log n)$ ဖြစ်လို့ — overall $O(n \log n)$ ဖြစ်လေ့ ရှိတယ်။

> ဒီ pattern ၂ ခုဟာ — အခန်း ၁၁ (Sorting) နဲ့ အခန်း ၁၂ (Heap) မှာ သင်ခဲ့တဲ့ tool တွေကို greedy logic နဲ့ တွဲသုံးတာ ဖြစ်ပါတယ်။ greedy ရဲ့ "ခက်တဲ့ အပိုင်း" က algorithm မဟုတ်ဘဲ — **"ဘယ် greedy choice က မှန်လဲ"** ဆိုတာ မြင်တာပါ။

Real-world Examples

- **Schedule Maximum Meetings** — အခန်းတစ်ခုတည်းမှာ meeting အများဆုံး ထည့်ဖို့ — "အစောဆုံး ပြီးတဲ့ meeting အရင် ရွေး" (activity selection)။
- **Assign Resources** — server / worker အကန့်အသတ်နဲ့ — request တွေကို size အလိုက် ခွဲဝေ (greedy fit)။
- **Minimize Waiting Time** — queue မှာ "အချိန်တိုဆုံး job အရင် လုပ်" ဆိုရင် customer အားလုံးရဲ့ စုစုပေါင်း စောင့်ချိန် အနည်းဆုံး (shortest job first)။

- **Choose Cheapest Option** — flight / hotel booking မှာ "အခု အသက်သာဆုံး ရွေး"၊ retry/backoff မှာ "အရင်ဆုံး ရနိုင်တဲ့ slot ယူ"။
- **Data Compression (Huffman)** — character frequency အလိုက် "အကြိမ်အနည်းဆုံး ၂ ခု အရင် ပေါင်း" — priority queue + greedy (ZIP/DEFLATE နဲ့ baseline JPEG မှာ Huffman coding component အဖြစ် သုံး)။

Questions

Greedy ပြဿနာ ဖြေတဲ့အခါ — အရင်ဆုံး **"ဘယ် greedy choice က မှန်မလဲ"** ကို ရှာ (sort key ဒါမှမဟုတ် heap)၊ ပြီးတော့ အဲ့ choice ကို တစ်ခုချင်း လိုက်ရွေးတာပါ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Assign Cookies

ကလေး `n` ယောက် နဲ့ cookie `m` ခု ပေးထားသည် — ကလေး `i` မှာ **ဆာလောင်မှု** `g[i]` ၊ cookie `j` မှာ **အရွယ်အစား** `s[j]` ရှိတယ်။ `s[j] >= g[i]` ဖြစ်မှ ကလေး `i` ကျေနပ်တယ်၊ cookie တစ်ခုကို ကလေး တစ်ယောက်ပဲ ပေးလို့ ရတယ်။ **အများဆုံး ဘယ်နှယောက်** ကျေနပ်စေနိုင်မလဲ။

```
Input: g = [1,2,3], s = [1,1]
Output: 1
   (cookie 2 ခုစလုံး size 1 — ဆာလောင်မှု 1 ကလေး တစ်ယောက်ပဲ ကျေနပ်)
```

ရှင်းလင်းချက်

Greedy choice: "အသေးဆုံး cookie ကို — အဲ့ဒါနဲ့ ကျေနပ်နိုင်တဲ့ ဆာလောင်မှု အနည်းဆုံး ကလေးကို ပေး"။ ဘာကြောင့် မှန်သလဲ — cookie အသေးကို ဆာလောင်မှု အများ ကလေးဆီ ဖြုန်းပစ်ရင် နစ်နာတယ်။ ဒါကြောင့် ၂ ခုစလုံး **sort** (ငယ်→ကြီး) လုပ်ပြီး — pointer ၂ ခုနဲ့ "ကိုက်ရင် ၂ ဖက်တိုး၊ မကိုက်ရင် cookie ဘက်ပဲ တိုး" (two pointers)။

`g = [1,2,3]` , `s = [1,1]` **ကို လိုက်ကြည့်ရအောင်** — (sort ပြီးသား)

i (ကလေး)	g[i]	j (cookie)	s[j]	s[j] >= g[i]?	လုပ်ဆောင်ချက်
0	1	0	1	`1 >= 1` ဟုတ်	ကလေး ကျေနပ် → `i=1, j=1`
1	2	1	1	`1 >= 2` မဟုတ်	cookie ကုန် → `j=2`

`j` က cookie အကုန် ကုန်ပြီ → ရပ်။ ကျေနပ်တဲ့ ကလေး = `i = 1` ✓

> **Time Complexity:** $O(n \log n + m \log m)$ - sort ၂ ခု။
> **Space Complexity:** Java in-place `Arrays.sort` → $O(1)$ extra (sort stack အပြင်)။

Java Solution

```java
class Solution {
```

```
        public int findContentChildren(int[] g, int[] s) {
            Arrays.sort(g);                      // ဆာလောင်မှု ငယ်→ကြီး
            Arrays.sort(s);                      // cookie size ငယ်→ကြီး
            int i = 0, j = 0;                    // i = ကလေး, j = cookie
            while (i < g.length && j < s.length) {
                if (s[j] >= g[i]) i++;           // ကိုက် — ကလေး ကျေနပ်
                j++;                             // cookie က ဘယ်လိုပဲဖြစ် ရှေ့တိုး
            }
            return i;                            // ကျေနပ်တဲ့ ကလေး အရေအတွက်
        }
    }
```

၂။ Jump Game

array `nums` ပေးထားသည် — `nums[i]` က index `i` ကနေ **ရှေ့သို့ အများဆုံး ခုန်နိုင်တဲ့ အကွာအဝေး**။ index `0` ကစ — **နောက်ဆုံး index** ကို ရောက်နိုင်/မရောက်နိုင် ပြန်ပါ။

```
Input: nums = [2,3,1,1,4]
Output: true
   (0 →(2 ခုန်)→ 2 →... ဒါမှမဟုတ် 0→1→4  အဆုံး ရောက်)

Input: nums = [3,2,1,0,4]
Output: false
   (index 3 မှာ value 0 — ဘယ်လိုမှ ကျော်လို့ မရ၊ index 4 မရောက်)
```

ရှင်းလင်းချက်

Greedy choice: "အခုထိ ရောက်နိုင်တဲ့ **အဝေးဆုံး index (reach)** ကို မှတ်ထား၊ index တစ်ခုစီ ဖြတ်ရင်း update"။ ဘာကြောင့် မှန်သလဲ — index `i` ကို ရောက်နိုင်ရင် (`i <= reach`) — အဲ့ဒီကနေ `i + nums[i]` ထိ ဆက်ရောက်နိုင်တယ်။ `reach` က နောက်ဆုံး index ကို မီရင် `true` ။ index `i` က `reach` ထက် ကျော်သွားရင် (`i > reach`) — အဲ့ဒီ index ကို ဘယ်လိုမှ မရောက်နိုင်တော့လို့ `false` ။

`nums = [2,3,1,1,4]` ကို လိုက်ကြည့်ရအောင် —

i	nums[i]	reach (မဖြတ်ခင်)	i > reach?	reach (ဖြတ်ပြီး = max(reach, i+nums[i]))
0	2	0	မဟုတ်	`max(0, 0+2) = 2`
1	3	2	မဟုတ်	`max(2, 1+3) = 4` ← index 4 မီပြီ
2	1	4	မဟုတ်	`max(4, 2+1) = 4`
3	1	4	မဟုတ်	`max(4, 3+1) = 4`
4	4	4	မဟုတ်	အဆုံး ရောက် → `true` ✓

`[3,2,1,0,4]` ဆိုရင် — index 1,2,3 မှာ reach က `3` ပဲ မတိုးတော့ဘဲ၊ `i = 4` မှာ `4 > 3` ဖြစ်လို့ `false` ။

> **Early termination:** `reach` က နောက်ဆုံး index (`n-1`) ကို မီတာနဲ့ — ကျန်တဲ့ array ဆက်ဖြတ်စရာ မလိုတော့ဘဲ ချက်ချင်း `true` ပြန်လို့ ရတယ်။ worst case complexity မပြောင်းပေမယ့် — အဖြေ စောစော တွေ့ရင် loop အစောဆုံး ရပ်လို့ လက်တွေ့မှာ ပိုမြန်တယ်။

> **Time Complexity:** $O(n)$ - တစ်ခေါက်ပဲ ဖြတ်။
> **Space Complexity:** $O(1)$ - `reach` တစ်ခုပဲ။

Java Solution

```
class Solution {
    public boolean canJump(int[] nums) {
        int reach = 0;                          // အခုထိ ရောက်နိုင်တဲ့ အဝေးဆုံး
        for (int i = 0; i < nums.length; i++) {
            if (i > reach) return false;        // ဒီ index မရောက်နိုင် — ပိတ်
            reach = Math.max(reach, i + nums[i]);  // reach ကို ချဲ့
            if (reach >= nums.length - 1) return true;  // နောက်ဆုံး မီပြီ — စောစီးစွာ ရပ်
        }
        return true;                            // loop ပြီး = နောက်ဆုံး ရောက်နိုင်
    }
}
```

၃။ Gas Station

ဝိုင်းပတ် လမ်းကြောင်းပေါ်မှာ gas station `n` ခု ရှိ — station `i` မှာ ဆီ `gas[i]` ဖြည့်နိုင်၊ station `i` ကနေ `i+1` သို့ သွားဖို့ ဆီ `cost[i]` ကုန်။ ဆီ ဗလာ tank နဲ့ စတင်ပြီး — ဝိုင်းတစ်ပတ် ပြန်လည်ပတ်နိုင်မယ့် **စမှတ် station index** ပြန်ပါ။ မဖြစ်နိုင်ရင် `-1` ။ (အဖြေ ရှိရင် တစ်ခုတည်း ရှိမယ်လို့ အာမခံ)။

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
   (station 3 ကစ → tank: +4-1=3 → +5-2=6 → +1-3=4 → +2-4=2 → +3-5=0  ပတ်ပြီး)
```

ရှင်းလင်းချက်

အဓိက ၂ ချက် —

1. ဆီ စုစုပေါင်း (`sum(gas)`) က ကုန်ကျ စုစုပေါင်း (`sum(cost)`) ထက် နည်းရင် — ဘယ်ကစစ ပတ်လို့ မရ → `-1` ။
2. **Greedy:** station တစ်ခုစီ ဖြတ်ရင်း `tank` (လက်ကျန်ဆီ) ပေါင်း — `tank < 0` ဖြစ်တဲ့ နေရာ ရောက်ရင် — အဲ့ start ကနေ ဒီထိ **ဘယ်နေရာကမှ** start လို့ မရဘူး (ကြားထဲ ဆီ ပြတ်)။ ဒါကြောင့် start ကို **နောက် station (** `i+1` **)** ကို ရွှေ့ပြီး `tank` reset။ ဒီနည်းနဲ့ စမှတ်ကို တစ်ခေါက်တည်း ($O(n)$) နဲ့ ရှာတယ်။

`gas = [1,2,3,4,5]` , `cost = [3,4,5,1,2]` ကို လိုက်ကြည့်ရအောင် —

```
i   diff=gas-cost   tank (ပေါင်းပြီး)   tank<0?         start
0     1-3 = -2          -2            ဟုတ် → reset      start=1, tank=0
```

```
1     2−4 = −2          −2           ဟုတ် → reset     start=2, tank=0
2     3−5 = −2          −2           ဟုတ် → reset     start=3, tank=0
3     4−1 =  3           3           မဟုတ်            start=3
4     5−2 =  3           6           မဟုတ်            start=3

total = −2−2−2+3+3 = 0 ≥ 0  →  answer = start = 3 ✓
```

`tank` က index 0,1,2 မှာ နုတ်ဖြစ်လို့ start ကို တဖြည်းဖြည်း ရွှေ့သွားပြီး — index 3 ကစတော့ tank မနုတ်တော့လို့ — start `3` က အဖြေ။ `total ≥ 0` ဖြစ်လို့ တကယ် ပတ်လို့ ရတယ်။

> **Time Complexity:** $O(n)$ - တစ်ခေါက်ပဲ ဖြတ်။
> **Space Complexity:** $O(1)$။

Java Solution

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int total = 0, tank = 0, start = 0;
        for (int i = 0; i < gas.length; i++) {
            int diff = gas[i] - cost[i];
            total += diff;                    // စုစုပေါင်း လက်ကျန် (ဖြစ်နိုင်/မဖြစ်)
            tank += diff;                     // လက်ရှိ start ကနေ လက်ကျန်
            if (tank < 0) {                   // ဒီထိ ပတ်လို့ မရ
                start = i + 1;                // start ကို နောက် station ရွှေ့
                tank = 0;                     // reset
            }
        }
        return total >= 0 ? start : -1;       // total < 0 ဆို ဘယ်လိုမှ မဖြစ်
    }
}
```

၄။ Minimum Number of Arrows to Burst Balloons

balloon တစ်ခုစီကို 2D axis ပေါ်မှာ `[start, end]` (အလျားလိုက် အကျယ်) နဲ့ ပေးထားသည်။ မြှား တစ်စင်းကို `x` နေရာမှာ မတ်မတ် ပစ်ရင် — `start <= x <= end` ဖြစ်တဲ့ balloon အကုန် ပေါက်တယ်။ balloon **အကုန် ပေါက်ဖို့ မြှား အနည်းဆုံး** ဘယ်နှ စင်း လိုလဲ။

```
Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output: 2
   (x=6 မှာ [1,6],[2,8] ပေါက် ;  x=12 မှာ [7,12],[10,16] ပေါက်  → ၂ စင်း)
```

ရှင်းလင်းချက်

ဒါက **activity selection** pattern ပါ — "overlap ဖြစ်တဲ့ balloon တွေကို မြှားတစ်စင်းတည်းနဲ့ ပေါက်"။ **Greedy choice:** balloon တွေကို `end` **အလိုက် sort** ပြီး — ပထမ balloon ရဲ့ `end` မှာ မြှား ပစ်။ နောက် balloon ရဲ့ `start` က အဲ့ မြှား `x` ထက် ကြီးနေမှ (overlap မဖြစ်တော့မှ) — မြှားအသစ် ထပ်ပစ်။ ဘာကြောင့် `end` နဲ့ sort လဲ — "အစောဆုံး ဆုံးတဲ့ balloon ရဲ့ အဆုံးမှာ ပစ်" ရင် — နောက်က overlap ဖြစ်နိုင်သမျှ အများဆုံး ပေါက်လို့။

`points = [[10,16],[2,8],[1,6],[7,12]]` ကို လိုက်ကြည့်ရအောင် —

```
end အလိုက် sort:  [1,6]  [2,8]  [7,12]  [10,16]

balloon     start > x ?            လုပ်ဆောင်ချက်
[1,6]       (ပထမ)                   arrows=1,  x=6
[2,8]       2 > 6 ? မဟုတ်             မြှား x=6 နဲ့ ပေါက်ပြီးသား (skip)
[7,12]      7 > 6 ? ဟုတ်              arrows=2,  x=12
[10,16]     10 > 12 ? မဟုတ်           မြှား x=12 နဲ့ ပေါက်ပြီးသား (skip)

→ မြှား 2 စင်း (x=6 နဲ့ x=12) ✓
```

> **Time Complexity:** $O(n \log n)$ - sort။
> **Space Complexity:** $O(n)$ - `points` က object array (`int[][]`) ဖြစ်လို့ Java `Arrays.sort(..., comparator)` (TimSort) $O(n)$ သုံး။

Java Solution

```java
class Solution {
    public int findMinArrowShots(int[][] points) {
        if (points.length == 0) return 0;
        // end အလိုက် sort — overflow ရှောင်ဖို့ Integer.compare
        Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

        int arrows = 1;
        int x = points[0][1];                // ပထမ balloon ရဲ့ end မှာ ပစ်
        for (int i = 1; i < points.length; i++) {
            if (points[i][0] > x) {          // overlap မဖြစ်တော့
                arrows++;                    // မြှား အသစ်
                x = points[i][1];
            }
        }
        return arrows;
    }
}
```

၅။ Task Scheduler

task တွေ (`tasks` — **စာလုံးကြီး** `'A'` – `'Z'` character) နဲ့ cooldown `n` ပေးထားသည်။ CPU က unit time တစ်ခုမှာ task တစ်ခု run ၊ ဒါမှမဟုတ် idle နေနိုင်။ **တူညီတဲ့ task ၂ ခုကြား အနည်းဆုံး `n` unit ခြားရမယ်**။ task အကုန် ပြီးဖို့ **အနည်းဆုံး unit time** ဘယ်လောက် လိုလဲ။

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
   (A B idle  A B idle  A B  → 8 unit ;  A,A ကြား ≥2 ခြား)
```

ရှင်းလင်းချက်

ဒါက **Priority Queue + Greedy** pattern ပါ။ **Greedy choice:** "လက်ကျန် **အများဆုံး** task ကို အရင် run" — ဒါမှ frequent task ကို ဖြန့်ပြီး idle အနည်းဆုံး ဖြစ်တယ်။ "အများဆုံး" က run ပြီးတိုင်း ပြောင်းနေလို့ — **max-heap** နဲ့ ဆွဲထုတ်ရတယ်။

အလုပ်လုပ်ပုံ — round တစ်ခုစီမှာ slot `n+1` ခု ရှိ (task + cooldown)။ round တစ်ခုစီ — heap ထဲက အများဆုံး `n+1` ခုကို ဆွဲထုတ်၊ count လျှော့၊ `0` မဟုတ်သေးရင် ပြန်ထည့်။ heap ကုန်ရင် ပြီး — မကုန်သေးရင် round တစ်ခုက `n+1` unit (idle ပါ)။

> **အရေးကြီး — round ထဲမှာ heap ကုန်ရင် `break` :** cooldown `n` ကြီးပြီး task နည်းတဲ့အခါ (ဥပမာ `["A","A"]` , `n=100`) — round ထဲ slot `n+1` ခု အကုန် iterate ရင် — heap ဗလာ slot တွေအတွက် အလဟဿ loop ဖြစ်ပြီး $O(N \times n)$ ဖြစ်သွားမယ်။ ဒါကြောင့် — heap ကုန်တာနဲ့ `break` လုပ်ပြီး round ကို ရပ်ရတယ် (idle slot တွေ မဖြုန်း)။ ဒါမှ pop အရေအတွက် စုစုပေါင်း = task `N` ခု ဖြစ်ပြီး $O(N)$ ရတယ်။

`tasks = ["A","A","A","B","B","B"]` , `n = 2` ကို လိုက်ကြည့်ရအောင် — (round တစ်ခု = `n+1` = ၃ slot)

```
freq:  A=3, B=3        heap (max): [3,3]

round 1: A run(3→2), B run(3→2), idle     → time += 3  (heap မကုန် → n+1)
         heap ပြန်: [2,2]
round 2: A run(2→1), B run(2→1), idle     → time += 3
         heap ပြန်: [1,1]
round 3: A run(1→0), B run(1→0)           → time += 2  (heap ကုန် → done=2)

စီစဉ်ပုံ:  A B _ | A B _ | A B      →  total time = 3+3+2 = 8 ✓
```

နောက်ဆုံး round မှာ A, B ၂ ခုပဲ ကျန်တော့ — idle မလိုဘဲ `done = 2` ပဲ ပေါင်းလို့ — `9` မဟုတ်ဘဲ `8` ရတယ်။

> **Time Complexity:** $O(N)$ - task `N` ခု၊ heap size ≤ 26 (constant) ဖြစ်လို့ heap op က $O(1)$။
> **Space Complexity:** $O(1)$ - count array / heap ≤ 26။

> **သတိ — နောက်ဆုံး round:** task ကုန်ခါနီး နောက်ဆုံး round မှာ idle မလို။ ဒါကြောင့် round တစ်ခုစီ — heap ကုန်သွားရင် **တကယ် run တဲ့ task အရေအတွက် (`done`)** ပဲ ပေါင်း၊ မကုန်သေးရင်မှ idle ပါတဲ့ `n+1` ပေါင်းတယ်။

Java Solution

```java
class Solution {
    public int leastInterval(char[] tasks, int n) {
        int[] freq = new int[26];
        for (char t : tasks) freq[t - 'A']++;
```

```
        // max-heap — လက်ကျန် အများဆုံး task အပေါ် (overflow ရှောင် Integer.compare)
        PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> Integer.compare(b,
a));
        for (int f : freq) if (f > 0) heap.offer(f);

        int time = 0;
        while (!heap.isEmpty()) {
            List<Integer> temp = new ArrayList<>();
            int done = 0;
            for (int i = 0; i <= n; i++) {     // round = n+1 slot
                if (heap.isEmpty()) break;     // heap ကုန်ရင် ဒီ round စောစီးစွာ ရပ် (idle
iteration မဖြုန်း)
                int cur = heap.poll() - 1;     // run — လက်ကျန် ၁ လျှော့
                if (cur > 0) temp.add(cur);
                done++;                        // တကယ် run တဲ့ task
            }
            for (int c : temp) heap.offer(c);  // လက်ကျန် ပြန်ထည့်
            time += heap.isEmpty() ? done : n + 1;  // နောက်ဆုံး round — idle မပါ
        }
        return time;
    }
}
```

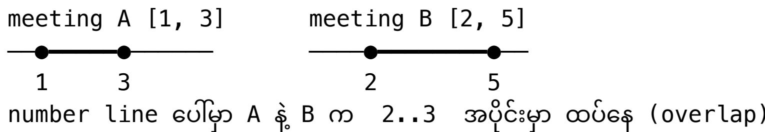
အခန်း ၂၁ - Intervals

အခန်း ၂၀ မှာ greedy ကို သင်တုန်း — "balloon ကို `end` အလိုက် sort ပြီး overlap မဖြစ်တော့မှ မြှား အသစ်" ဆိုတဲ့ ပြဿနာကို ဖြေခဲ့ပါတယ်။ အဲ့ဒါက **interval (အပိုင်းအခြား)** ပြဿနာ တစ်မျိုးပါ။ Interval ပြဿနာတွေက real-world developer အတွက် အလွန် အသုံးဝင်တယ်။ calendar booking, hotel room, subscription period, promotion date, shift schedule စတဲ့ "**အချိန် / အပိုင်းအခြား**" နဲ့ ဆိုင်တဲ့ feature တိုင်းမှာ တွေ့ရတယ်။

Interval ဆိုတာ — `[start, end]` ဆိုတဲ့ "အစ–အဆုံး အတွဲ" တစ်ခုပါ (ဥပမာ — meeting `[9, 10]` က ၉ နာရီကစ ၁၀ နာရီ ဆုံး)။ ဒီအခန်းမှာ — interval တွေ **overlap (ထပ်) ဖြစ်/မဖြစ်** ဘယ်လို စစ်လဲ၊ ဘယ်အချက်နဲ့ **sort** လုပ်ရင် ပြဿနာ ရှင်းသွားလဲ၊ ပြီးတော့ **sweep line** ဆိုတဲ့ နည်းကို မိတ်ဆက်ပြီး classic ပြဿနာ ၅ ခု ဖြေပါမယ်။

Interval ဆိုတာ ဘာလဲ — Start, End, Overlap

Interval = `[start, end]` — number line ပေါ်က အပိုင်းတစ်ခု။



Overlap Detection — အရေးကြီးဆုံး အခြေခံ

Interval ပြဿနာ အကုန်လုံးရဲ့ အနှစ်ချုပ်က — "interval ၂ ခု **ထပ်လား မထပ်ဘူးလား**" ကို စစ်တာပါ။ စည်းမျဉ်းက ရိုးရှင်းတယ် —

```
A = [a1, a2],  B = [b1, b2]   (a1 ≤ a2,  b1 ≤ b2)

overlap ဖြစ်ဖို့:   a1 ≤ b2   AND   b1 ≤ a2
(A ရဲ့အစ က B ရဲ့အဆုံးထက် မကျော်  +  B ရဲ့အစ က A ရဲ့အဆုံးထက် မကျော်)
```

တစ်နည်းပြောရင် — "**ထပ်မဖြစ်** ဖို့က — တစ်ခုက တစ်ခုရဲ့ ရှေ့မှာ လုံးဝ ပြီးသွားရမယ်" (`a2 < b1` ဒါမှမဟုတ် `b2 < a1`)။ ဒါမဟုတ်ရင် ထပ်တယ်။

```
ထပ်တယ်:     [1,4] နဲ့ [3,6]   → 1≤6 AND 3≤4 ✓
မထပ်ဘူး:    [1,2] နဲ့ [5,7]   → 2 < 5  (A ပြီးမှ B စ)
```

Sort by Start vs Sort by End

Interval ပြဿနာ အများစုကို ဖြေဖို့ — အရင်ဆုံး **sort** လုပ်ရတယ်။ ဘယ်အချက်နဲ့ sort လုပ်မလဲ ဆိုတာက ပြဿနာ အလိုက် ကွဲတယ် — ဒါက အရေးအကြီးဆုံး ဆုံးဖြတ်ချက်ပါ။

- **Sort by start** — "interval တွေကို ဘယ်ကစ ဘယ်လို တန်းစီနေလဲ" ကြည့်ချင်တဲ့အခါ — **merge, insert** ပြဿနာတွေအတွက်။ အစ အလိုက် စီထားရင် — ဘေးချင်း interval တွေပဲ overlap ဖြစ်နိုင်လို့ စစ်ရ လွယ်တယ်။
- **Sort by end** — "အများဆုံး ဘယ်နှ ခု ထပ်မဖြစ်အောင် ရွေးနိုင်လဲ" (greedy activity selection) — **non-overlapping** ပြဿနာတွေအတွက်။ "အစောဆုံး ပြီးတာ အရင်ရွေး" ရင် နောက်အတွက် နေရာ အများဆုံး ကျန်လို့။

> **မှတ်ရန်:** merge / insert → **start** အလိုက်။ count / remove (greedy) → **end** အလိုက်။ ဒီ ၂ ခု ကို ခွဲမှတ်ထားရင် interval ပြဿနာ အများစု ရှင်းသွားတယ်။

Sweep Line — အချိန်တန်း တစ်လျှောက် လှည့်ကြည့်ခြင်း

Sweep line ဆိုတာ — interval တွေကို တစ်ခုလုံး မကြည့်ဘဲ — **number line (အချိန်တန်း) တစ်လျှောက် ဘယ်ကညာ လှည့်သွား**ပြီး၊ event (start / end) တစ်ခုစီမှာ "အခု တစ်ပြိုင်နက် ဘယ်နှ ခု active ဖြစ်နေလဲ" ရေတွက်တဲ့ နည်းပါ။

အဓိက idea — interval `[s, e]` တစ်ခုစီကို event ၂ ခု အဖြစ် ခွဲ —

- `s` မှာ **+1** (interval တစ်ခု စ — active တိုး)
- `e` မှာ **−1** (interval တစ်ခု ဆုံး — active လျော့)

event တွေကို အချိန်အလိုက် sort ပြီး — running count ကို ကြည့်ရင် — "တစ်ချိန်တည်းမှာ အများဆုံး ဘယ်နှ ခု ထပ်နေလဲ" ကို သိတယ်။ ဒါက **meeting room အရေအတွက်, minimum platforms** လို ပြဿနာတွေရဲ့ အခြေခံပါ။

```
meetings: [1,4] [2,5] [7,9]

events (sort အလိုက်):  1:+1  2:+1  4:-1  5:-1  7:+1  9:-1
running count:          1     2     1     0     1     0
                              ↑ အများဆုံး = 2  → အခန်း ၂ ခု လို
```

Real-world Examples

- **Calendar Booking** — Google Calendar မှာ event အသစ် ထည့်ရင် "ရှိပြီးသား event နဲ့ ထပ်လား" (overlap) စစ်တာ၊ free slot ရှာတာ။
- **Hotel / Room Booking** — date range `[check-in, check-out]` တွေ ထပ်မဖြစ်အောင်၊ ဒါမှမဟုတ် "တစ်ချိန်တည်း အခန်း ဘယ်နှ ခု လို" (sweep line) တွက်တာ။
- **Subscription Active Period** — user ရဲ့ subscription `[start, expiry]` တွေ merge လုပ်ပြီး "စုစုပေါင်း active ရက်" တွက်တာ။

- **Promotion / Discount Date Range** — promotion `[from, to]` တွေ ထပ်နေရင် merge၊ conflict ရှိမရှိ စစ်တာ။
- **Employee Shift Schedule** — shift `[in, out]` တွေ — တစ်ချိန်တည်း ဝန်ထမ်း ဘယ်နှ ယောက် လို၊ shift ထပ်နေလား စစ်တာ။

Questions

Interval ပြဿနာ ဖြေတဲ့အခါ — အရင်ဆုံး **"start အလိုက် sort လား end အလိုက် sort လား"** ဆုံးဖြတ်၊ ပြီးတော့ overlap rule ကို သုံးတာပါ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Merge Intervals

interval list `intervals` (`[start, end]`) ပေးထားသည်။ **ထပ်နေတဲ့ (overlap) interval တွေ ပေါင်းစည်း** ပြီး — ထပ်မဖြစ်တော့တဲ့ interval list ပြန်ပါ။

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
   ([1,3] နဲ့ [2,6] ထပ်နေ → [1,6] ပေါင်း)
```

ရှင်းလင်းချက်

Sort by start — interval တွေကို အစ အလိုက် စီလိုက်ရင် — ပေါင်းစရာ interval တွေ **ဘေးချင်းကပ်** လာတယ်။ ပြီးတော့ result list ထဲ တစ်ခုချင်း ထည့်ရင်း — "လက်ရှိ interval ရဲ့ start က result ရဲ့ နောက်ဆုံး interval ရဲ့ end ထက် ≤ ဆို (ထပ်)" → end ကို `max` နဲ့ ချဲ့၊ မထပ်ရင် interval အသစ် ထည့်။

`[[1,3],[2,6],[8,10],[15,18]]` **ကို လိုက်ကြည့်ရအောင်** — (start အလိုက် sort ပြီးသား)

cur	result နောက်ဆုံး	cur.start > last.end?	လုပ်ဆောင်ချက်
`[1,3]`	(ဗလာ)	—	`result = [[1,3]]`
`[2,6]`	`[1,3]`	`2 > 3` ? မဟုတ် (ထပ်)	end ချဲ့ → `[[1,6]]`
`[8,10]`	`[1,6]`	`8 > 6` ? ဟုတ် (မထပ်)	အသစ် → `[[1,6],[8,10]]`
`[15,18]`	`[8,10]`	`15 > 10` ? ဟုတ် (မထပ်)	အသစ် → `[[1,6],[8,10],[15,18]]`

→ `[[1,6],[8,10],[15,18]]` ✓

> **Time Complexity:** $O(n \log n)$ - sort က dominant။
> **Space Complexity:** $O(n)$ - result list (sort အတွက် $O(\log n)$)။

Java Solution

```
class Solution {
```

```
    public int[][] merge(int[][] intervals) {
        // start အလိုက် sort — overflow ရှောင် Integer.compare
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        List<int[]> result = new ArrayList<>();
        for (int[] cur : intervals) {
            // result ဗလာ ဒါမှမဟုတ် ထပ်မဖြစ် → interval အသစ်
            if (result.isEmpty() || cur[0] > result.get(result.size() - 1)[1]) {
                result.add(cur);
            } else {                        // ထပ် → end ချဲ့
                int[] last = result.get(result.size() - 1);
                last[1] = Math.max(last[1], cur[1]);
            }
        }
        return result.toArray(new int[0][]);
    }
}
```

၂။ Insert Interval

ထပ်မဖြစ်အောင် sort ထားပြီးသား interval list `intervals` နဲ့ — interval အသစ် `newInterval` ပေးထားသည်။ `newInterval` ကို ထည့်ပြီး — ထပ်ရင် merge လုပ်၊ ထပ်မဖြစ်တော့တဲ့ list ပြန်ပါ။

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
   ([2,5] က [1,3] နဲ့ ထပ် → [1,5] ပေါင်း)
```

ရှင်းလင်းချက်

list က sort ပြီးသားမို့ — အပိုင်း ၃ ပိုင်း အဖြစ် ဖြတ်တယ်။ (၁) `newInterval` ရဲ့ **အရှေ့မှာ လုံးဝ ပြီးသွားတဲ့** interval တွေ — တိုက်ရိုက် ထည့်။ (၂) `newInterval` နဲ့ **ထပ်တဲ့** interval တွေ — start `min` , end `max` နဲ့ပေါင်းသွား။ (၃) `newInterval` ရဲ့ **အနောက်က** interval တွေ — တိုက်ရိုက် ထည့်။ sort ပြီးသားမို့ — sort ပြန်မလို၊ $O(n)$ နဲ့ ပြီးတယ်။

`intervals = [[1,3],[6,9]]` , `newInterval = [2,5]` ကို လိုက်ကြည့်ရအောင် —

```
အပိုင်း (၁) — newInterval [2,5] ရှေ့မှာ လုံးဝ ပြီးတာ (end < 2):
    [1,3] → end 3 < 2 ? မဟုတ် → ဒီအပိုင်း ဘာမှ မထည့်

အပိုင်း (၂) — [2,5] နဲ့ ထပ်တာ (start ≤ 5) ကို ပေါင်း:
    [1,3] → start 1 ≤ 5 ? ဟုတ် → merge:  [min(2,1), max(5,3)] = [1,5]
    [6,9] → start 6 ≤ 5 ? မဟုတ် → ရပ်
    result = [[1,5]]

အပိုင်း (၃) — ကျန်တာ တိုက်ရိုက်ထည့်:
    [6,9] → result = [[1,5],[6,9]]

→ [[1,5],[6,9]] ✓
```

> **Time Complexity:** $O(n)$ - တစ်ခေါက်ပဲ ဖြတ် (sort ပြီးသား)။
> **Space Complexity:** $O(n)$ - result list။

Java Solution

```
class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int i = 0, n = intervals.length;

        // (၁) newInterval အရှေ့မှာ လုံးဝ ပြီးတဲ့ interval — တိုက်ရိုက်ထည့်
        while (i < n && intervals[i][1] < newInterval[0]) result.add(intervals[i++]);

        // (၂) ထပ်တဲ့ interval — start min, end max နဲ့ ပေါင်း
        while (i < n && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
            newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
            i++;
        }
        result.add(newInterval);

        // (၃) ကျန်တဲ့ (အနောက်က) interval — တိုက်ရိုက်ထည့်
        while (i < n) result.add(intervals[i++]);

        return result.toArray(new int[0][]);
    }
}
```

၃။ Meeting Rooms (Can Attend All)

meeting interval list `intervals` (`[start, end]`) ပေးထားသည်။ လူတစ်ယောက်က meeting **အကုန်လုံး တက်နိုင်/မတက်နိုင်** ပြန်ပါ (meeting ၂ ခု ထပ်နေရင် မတက်နိုင်)။

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: false
   ([0,30] နဲ့ [5,10] ထပ်နေ → အကုန် မတက်နိုင်)
```

ရှင်းလင်းချက်

Sort by start — meeting တွေကို အစ အလိုက် စီပြီး — ဘေးချင်း meeting ၂ ခု ထပ်လား စစ်ရုံပါ။ "လက်ရှိ meeting ရဲ့ start က ရှေ့ meeting ရဲ့ end ထက် ငယ်ရင် (`cur.start < prev.end`)" → ထပ်နေ → `false` ။ တစ်ခုမှ မထပ်ရင် `true` ။

> **Time Complexity:** $O(n \log n)$ - sort။
> **Space Complexity:** $O(1)$ (sort အပြင်)။

Java Solution

```
class Solution {
    public boolean canAttendMeetings(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));  // start အလိုက်
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] < intervals[i - 1][1]) return false;    // ထပ်နေ
        }
```

```
            return true;
        }
    }
```

၄။ Meeting Rooms II (Minimum Rooms)

meeting interval list ပေးထားသည်။ meeting **အကုန် ကျင်းပဖို့ အနည်းဆုံး အခန်း ဘယ်နှ ခု** လိုလဲ ပြန်ပါ (တစ်ချိန်တည်း ထပ်နေတဲ့ meeting တွေက အခန်း သီးသန့်စီ လို)။

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
   ([0,30] တစ်ခန်း ;  [5,10] နဲ့ [15,20] က နောက်တစ်ခန်းမှာ ဆက်တိုက်  → ၂ ခန်း)
```

ရှင်းလင်းချက်

ဒါက **sweep line** ပြဿနာ ပါ — "တစ်ချိန်တည်း အများဆုံး ဘယ်နှ meeting ထပ်နေလဲ" = လိုအပ်တဲ့ အခန်း အရေအတွက်။ နည်း ၂ မျိုး —

1. **Sweep line (start/end ခွဲ sort):** start array နဲ့ end array ကို သီးသန့် sort၊ pointer ၂ ခုနဲ့ လိုက်ဖြတ် — meeting အသစ် စတိုင်း (`start < end`) အခန်းတိုး၊ ဟောင်းတစ်ခု ဆုံးရင် အခန်းပြန်လျှော့။ running max = answer။
2. **Min-heap:** start အလိုက် sort၊ heap မှာ active meeting တွေရဲ့ end သိမ်း — meeting အသစ်ရဲ့ start က heap ထိပ် (အစောဆုံး end) ≥ ဆို → အခန်းဟောင်း ပြန်သုံး (pop)၊ မဟုတ်ရင် အခန်းသစ်။ heap size = active room။

အောက်မှာ **sweep line** version ပြထားတယ် (heap မလို၊ ပိုမြန်)။

`[[0,30],[5,10],[15,20]]` ကို လိုက်ကြည့်ရအောင် —

`starts` sort: `[0, 5, 15]` | `ends` sort: `[10, 20, 30]` | `i` = start pointer, `j` = end pointer

i	starts[i]	ends[j]	starts[i] < ends[j]?	rooms	maxRooms
0	0	10	`0 < 10` ဟုတ် → meeting စ	1	1
1	5	10	`5 < 10` ဟုတ် → meeting စ	2	2
2	15	10	`15 < 10` မဟုတ် → ဟောင်းဆုံး 1 (`j→1`)	1	2

→ `maxRooms = 2` (တစ်ချိန်တည်း အများဆုံး ၂ ခု ထပ်) ✓

> **Time Complexity:** $O(n \log n)$ - start/end sort။
> **Space Complexity:** $O(n)$ - start/end array။

Java Solution

```
class Solution {
    public int minMeetingRooms(int[][] intervals) {
        int n = intervals.length;
        int[] starts = new int[n], ends = new int[n];
        for (int i = 0; i < n; i++) {
            starts[i] = intervals[i][0];
            ends[i] = intervals[i][1];
        }
        Arrays.sort(starts);                    // start တွေ သီးသန့် sort
        Arrays.sort(ends);                      // end တွေ သီးသန့် sort

        int rooms = 0, maxRooms = 0, j = 0;
        for (int i = 0; i < n; i++) {
            // meeting အသစ်ရဲ့ start က အစောဆုံး end ထက် ငယ် → ထပ် → အခန်းတိုး
            if (starts[i] < ends[j]) {
                rooms++;
                maxRooms = Math.max(maxRooms, rooms);
            } else {                            // ဟောင်းတစ်ခု ဆုံးပြီ → အခန်းပြန်သုံး
                j++;
            }
        }
        return maxRooms;
    }
}
```

၅။ Non-overlapping Intervals

interval list ပေးထားသည်။ ကျန်တဲ့ interval တွေ **ထပ်မဖြစ်တော့အောင်** ဖို့ — **အနည်းဆုံး ဘယ်နှ ခု ဖယ်ရှား** ရမလဲ ပြန်ပါ။

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
   ([1,3] ကို ဖယ်ရင် ကျန်တာ ထပ်မဖြစ်တော့  → ၁ ခု ဖယ်)
```

ရှင်းလင်းချက်

ဒါက greedy **activity selection** ပါ — "ထပ်မဖြစ်အောင် **အများဆုံး** ဘယ်နှ ခု ထားနိုင်လဲ" ရှာပြီး — ကျန်တာ ဖယ်တာ။ **Sort by end** — "အစောဆုံး ပြီးတဲ့ interval အရင် ထား" ရင် — နောက်အတွက် နေရာ အများဆုံး ကျန်လို့ ထပ်မဖြစ်တာ အများဆုံး ထားနိုင်တယ်။ နောက် interval ရဲ့ start က ထား ထားတဲ့ end ထက် ငယ်ရင် (ထပ်) → ဖယ် (count++)၊ မဟုတ်ရင် ထား (end update)။

- `end` အလိုက် sort။
- `prevEnd` ထား၊ interval တစ်ခုစီ — `cur.start < prevEnd` ဆို → ဖယ် (`count++`)၊ မဟုတ်ရင် `prevEnd = cur.end` ။

`[[1,2],[2,3],[3,4],[1,3]]` ကို လိုက်ကြည့်ရအောင် —

```
end အလိုက် sort:  [1,2]  [2,3]  [1,3]  [3,4]
                                ↑ end 3 တူပေမယ့် ... [2,3] အရင်

cur      prevEnd    cur.start < prevEnd ?        လုပ်ဆောင်ချက်
[1,2]    (ပထမ)       —                            prevEnd = 2
[2,3]    2          2 < 2 ? မဟုတ်                 ထား → prevEnd = 3
```

```
[1,3]    3         1 < 3 ? ဟုတ် (ထပ်)          ဖယ် → count = 1
[3,4]    3         3 < 3 ? မဟုတ်                ထား → prevEnd = 4

→ ဖယ်ရတဲ့ interval = 1 ([1,3] ကို ဖယ်) ✓
```

> **Time Complexity:** $O(n \log n)$ - sort။
> **Space Complexity:** $O(1)$ (sort အပြင်)။

Java Solution

```java
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals.length == 0) return 0;
        // end အလိုက် sort — overflow ရှောင် Integer.compare
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));

        int count = 0;
        int prevEnd = intervals[0][1];
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] < prevEnd) {   // ထပ်နေ → ဖယ်
                count++;
            } else {                           // မထပ် → ထား
                prevEnd = intervals[i][1];
            }
        }
        return count;
    }
}
```

အခန်း ၂၂ - 1-D Dynamic Programming

အခန်း ၂၀ (Greedy) မှာ — အကြွေစေ့ `[1,3,4]` နဲ့ `6` ဖွဲ့တဲ့အခါ greedy က `4+1+1` (၃ စေ့) ဆိုပြီး မှား သွားတာ တွေ့ခဲ့ပါတယ် (အမှန် `3+3` = ၂ စေ့)။ "အခု အကောင်းဆုံး" ရွေးတာ မလုံလောက်တဲ့ — ဒီလို ပြဿနာတွေကို ဖြေဖို့ — ဖြစ်နိုင်ခြေ **အကုန် စစ်**ဖို့ လိုတယ်။ ဒါပေမယ့် အကုန် brute-force စစ်ရင် — တူညီတဲ့ အလုပ်ကို **ထပ်ခါထပ်ခါ** လုပ်မိပြီး အလွန် နှေးတယ်။

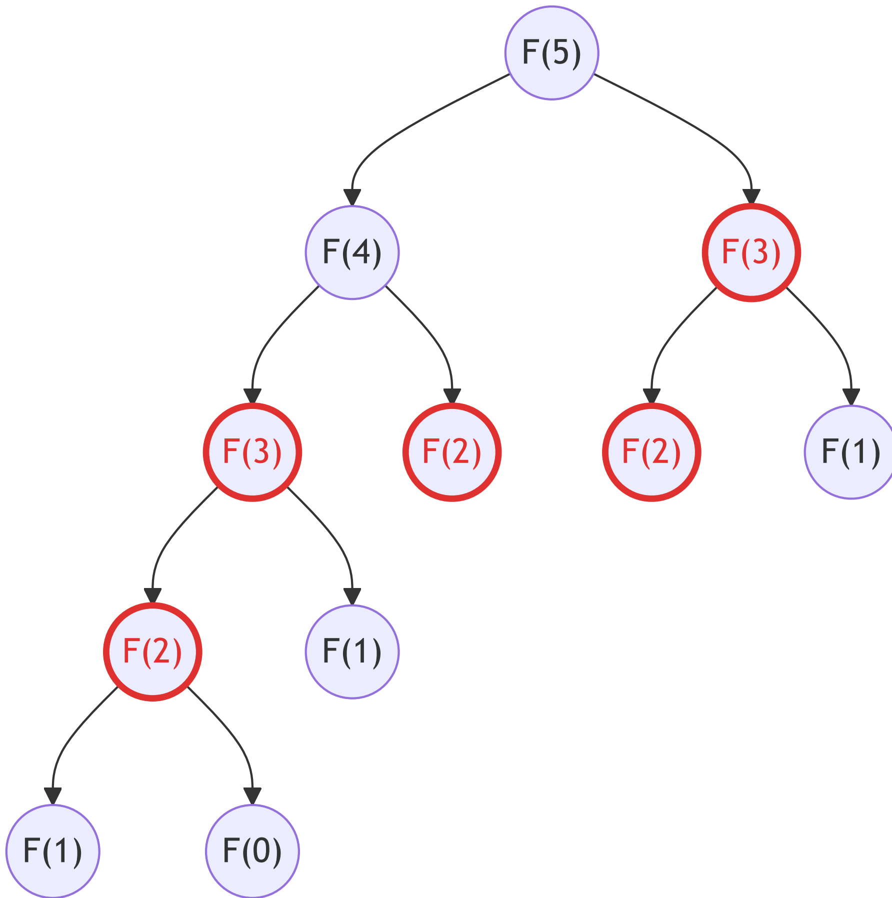
Dynamic Programming (DP) က ဒီပြဿနာကို ဖြေတယ် — "**ထပ်တူ ပြဿနာငယ်တွေရဲ့ အဖြေကို တစ်ခါတည်း တွက်ပြီး သိမ်းထား၊ ပြန်လိုရင် ပြန်သုံး**" ဆိုတဲ့ idea ပါ။ DP ဟာ beginner အများစု အတွက် အခက်ဆုံး topic လို့ ဆိုကြပေမယ့် — တကယ်တော့ **recursion** → **memoization** → **tabulation** ဆိုတဲ့ flow အတိုင်း တစ်ဆင့်ချင်း လိုက်သွားရင် — အလွန် စနစ်ကျတဲ့ pattern တစ်ခုသာ ဖြစ်ပါတယ်။ ဒီအခန်းမှာ အဲ့ဒီ flow ကို Fibonacci နဲ့ စတင်ပြီး — classic ပြဿနာ ၅ ခုနဲ့ လေ့လာသွားပါမယ်။

DP ဆိုတာ ဘာလဲ — ဘယ်တော့ သုံးလဲ

ပြဿနာတစ်ခုကို DP နဲ့ ဖြေလို့ ရဖို့ — အင်္ဂါရပ် ၂ ခု ရှိရတယ် —

၁။ Overlapping Subproblems (ထပ်တူ ပြဿနာငယ်)

ပြဿနာကြီးကို ပြဿနာငယ်တွေ ခွဲဖြေတဲ့အခါ — **တူညီတဲ့ ပြဿနာငယ်** ကို အကြိမ်ကြိမ် ပြန်ဖြေနေရ တယ်။ Fibonacci ($F(n) = F(n-1) + F(n-2)$) ကို ကြည့်ရအောင် —



→ `F(3)` ၂ ခါ၊ `F(2)` ၃ ခါ ... ထပ်တွက်နေ (ပြောင်းရောင် = ထပ်တူ ပြဿနာငယ်)

`F(2)` တို့ `F(3)` တို့ကို **ထပ်ခါထပ်ခါ** တွက်နေတာ — ဒါ overlapping subproblem ပါ။ ဒီအဖြေတွေ သိမ်းထားရင် တစ်ခါပဲ တွက်ရတော့မယ်။

၂။ Optimal Substructure (အကောင်းဆုံး ဖွဲ့စည်းပုံ)

ပြဿနာကြီးရဲ့ **အကောင်းဆုံး အဖြေ**ဟာ — ပြဿနာငယ်တွေရဲ့ **အကောင်းဆုံး အဖြေ**တွေကနေ တည်ဆောက်လို့ ရတယ်။ ဥပမာ — "n ထစ် တက်ဖို့ နည်းလမ်း အရေအတွက်" က — "n-1 ထစ်" နဲ့ "n-2 ထစ်" ရဲ့ အဖြေတွေ ပေါင်းတာပဲ။

ဒီ ၂ ခု ရှိရင် — DP သုံးလို့ ရတယ်။ (greedy နဲ့ ကွာတာက — greedy မှာ optimal substructure ရှိပေမယ့် "တစ်ခုတည်း ရွေး"တာ၊ DP မှာ "ဖြစ်နိုင်ခြေ အကုန် စစ်ပြီး အကောင်းဆုံး ယူ"တာ)။

DP ၃ ဆင့် — Recursion → Memoization → Tabulation

DP ပြဿနာ တစ်ခုကို ဖြေတဲ့အခါ — အဆင့် ၃ ဆင့်နဲ့ တိုးတက်အောင် လုပ်သွားတာက အကောင်းဆုံး လမ်းကြောင်းပါ။ Fibonacci နဲ့ ကြည့်ရအောင်။

အဆင့် ၁ — Plain Recursion (နှေး)

ပြဿနာကို တိုက်ရိုက် recursion နဲ့ ရေးတာ — မှန်ပေမယ့် overlapping subproblem ကြောင့် **exponential** $O(2^n)$ — အလွန် နှေး။

```
int fib(int n) {
    if (n <= 1) return n;                    // base case
    return fib(n - 1) + fib(n - 2);          // ထပ်တူ ပြဿနာငယ် ထပ်တွက်
}
```

အဆင့် ၂ — Memoization (Top-down + Cache)

recursion ကို ထားပြီး — တွက်ပြီးသား အဖြေကို **cache (memo)** ထဲ သိမ်း၊ ပြန်လိုရင် ပြန်သုံး။ ဒါနဲ့ ပြဿနာငယ် တစ်ခုကို **တစ်ခါပဲ** တွက်ရတော့လို့ — $O(n)$ ဖြစ်သွားတယ်။ ("top-down" — ပြဿနာကြီးကစ ဆင်းဖြေတာ)။

```
int fib(int n, int[] memo) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];        // တွက်ပြီးသား → ပြန်သုံး
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);  // တွက်ပြီး သိမ်း
    return memo[n];
}
```

အဆင့် ၃ — Tabulation (Bottom-up + Array)

recursion ဖျောက်ပြီး — **ပြဿနာငယ်ကစ** (base case ကစ) array `dp[]` ကို တဖြည်းဖြည်း ဖြည့်တက်တာ။ "bottom-up" — အသေးကစ အကြီးဆီ။ recursion stack မလိုလို့ stack overflow ရှောင်နိုင်ပြီး၊ ပိုမြန်တယ်။

```
int fib(int n) {
    if (n <= 1) return n;
    int[] dp = new int[n + 1];
    dp[0] = 0; dp[1] = 1;                     // base case
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];        // transition
    }
    return dp[n];
}
```

```
dp ဖြည့်ပုံ (n=6):
dp[0]=0  dp[1]=1  dp[2]=1  dp[3]=2  dp[4]=3  dp[5]=5  dp[6]=8
         └──────── dp[i] = dp[i-1] + dp[i-2] ────────┘
```

အဆင့် ၄ (bonus) — Space Optimization

`dp[i]` က `dp[i-1]` နဲ့ `dp[i-2]` ၂ ခုပဲ လိုတာမို့ — array တစ်ခုလုံး မလိုဘဲ **variable ၂ ခု**နဲ့ ရတယ် — space $O(n) \to O(1)$။

```
int fib(int n) {
    if (n <= 1) return n;
    int prev2 = 0, prev1 = 1;
```

```
    for (int i = 2; i <= n; i++) {
        int cur = prev1 + prev2;
        prev2 = prev1;
        prev1 = cur;
    }
    return prev1;
}
```

> **မှတ်ရန်:** Memoization (top-down) နဲ့ Tabulation (bottom-up) ၂ ခုလုံး $O(n)$ ပါ — ရွေးချယ်မှုပဲ။ Recursion နဲ့ စဉ်းစားရ လွယ်ရင် memoization၊ stack overflow / speed အရေးကြီးရင် tabulation။ ဒီအခန်းက ပြဿနာ အများစုကို **tabulation (1-D array)** နဲ့ ပြသွားပါမယ်။

DP ရေးနည်း Framework — State, Transition, Base Case

DP ပြဿနာ တိုင်းကို ဖြေဖို့ — အောက်က မေးခွန်း ၃ ခု ဖြေနိုင်ရင် ပြီးပါပြီ —

1. **State (`dp[i]` ဆိုတာ ဘာကို ကိုယ်စားပြုလဲ)** — ဥပမာ " `dp[i]` = index `i` ထိ အကောင်းဆုံး အဖြေ"။ ဒါက အခက်ဆုံး အပိုင်း — state မှန်အောင် သတ်မှတ်နိုင်ရင် ကျန်တာ လွယ်သွားတယ်။
2. **Transition (`dp[i]` ကို အရင် state တွေကနေ ဘယ်လို တွက်လဲ)** — ဥပမာ `dp[i] = dp[i-1] + dp[i-2]` ။ ဒါက DP ရဲ့ "ဆက်စပ်မှု ပုံသေနည်း"။
3. **Base case (အသေးဆုံး state ရဲ့ တန်ဖိုး)** — ဥပမာ `dp[0] = 0, dp[1] = 1` ။ recursion ရပ်တဲ့ နေရာ။

ဒီ framework ကို ပြဿနာ ၅ ခုမှာ ထပ်ခါ သုံးသွားပါမယ် — state ရှာ၊ transition ရေး၊ base case သတ်မှတ်။

Real-world Examples

- **Best Reward / Points** — အဆင့်ဆင့် ဆုံးဖြတ်ချက် (ဥပမာ — "ဒီ level ကို ယူရင် reward + ၊ ဒါပေမယ့် နောက် level skip") မှာ — အကောင်းဆုံး စုစုပေါင်း reward တွက်တာ (House Robber pattern)။
- **Retry / Cost Optimization** — operation တစ်ခုစီ retry cost မတူရင် — "အနည်းဆုံး ကုန်ကျမှုနဲ့ အဆုံးထိ ရောက်ဖို့" အဆင့်ဆင့် တွက်တာ (Coin Change / Min Cost pattern)။
- **Step-by-step Decision** — "ဒီအဆင့်မှာ ဘယ် option ရွေးရင် နောက်ဆုံး အကောင်းဆုံးလဲ" ဆိုတဲ့ planning — ယခင် ဆုံးဖြတ်ချက်ရဲ့ ရလဒ်ကို သိမ်းပြီး ဆက်တွက်တာ။
- **Text / Message Parsing** — SMS / code ကို decode လုပ်တဲ့အခါ "ဖြစ်နိုင်တဲ့ နည်းလမ်း ဘယ်နှ ခု" တွက်တာ (Decode Ways)။
- **Analytics** — transaction stream မှာ "အမြတ်အများဆုံး ရတဲ့ ဆက်တိုက် ကာလ" ရှာတာ (Maximum Subarray)။

Questions

DP ပြဿနာ ဖြေတဲ့အခါ — အရင်ဆုံး **state** (`dp[i]` ဘာလဲ) သတ်မှတ်၊ ပြီးမှ **transition** နဲ့ **base case** ရှာတာပါ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Climbing Stairs

လှေကားမှာ `n` ထစ် ရှိ — တစ်ခါ **၁ ထစ်** ဒါမှမဟုတ် **၂ ထစ်** တက်နိုင်။ ထိပ်ဆုံး (`n`) ထိ တက်ဖို့ **ကွဲပြားတဲ့ နည်းလမ်း ဘယ်နှ မျိုး** ရှိလဲ။

```
Input: n = 3
Output: 3
  (1+1+1 ;  1+2 ;  2+1  → ၃ နည်း)
```

ရှင်းလင်းချက်

- **State:** `dp[i]` = ထစ် `i` ထိ တက်နိုင်တဲ့ နည်းလမ်း အရေအတွက်။
- **Transition:** ထစ် `i` ကို — ထစ် `i-1` ကနေ (၁ ထစ်) ဒါမှမဟုတ် ထစ် `i-2` ကနေ (၂ ထစ်) ရောက်နိုင်လို့ → `dp[i] = dp[i-1] + dp[i-2]` ။ (Fibonacci နဲ့ တူ!)
- **Base case:** `dp[0] = 1` (ဘာမှ မတက်ဘဲ ၁ နည်း)၊ `dp[1] = 1` ။

`n = 3` ကို လိုက်ကြည့်ရအောင် —

```
dp[0]=1   dp[1]=1
dp[2] = dp[1]+dp[0] = 1+1 = 2
dp[3] = dp[2]+dp[1] = 2+1 = 3   ✓
```

> **Time Complexity:** $O(n)$ - ထစ်တစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(1)$ - variable ၂ ခုပဲ (space optimized)။

Java Solution

```java
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int prev2 = 1, prev1 = 2;            // dp[1]=1, dp[2]=2
        for (int i = 3; i <= n; i++) {
            int cur = prev1 + prev2;         // dp[i] = dp[i-1] + dp[i-2]
            prev2 = prev1;
            prev1 = cur;
        }
        return prev1;
    }
}
```

၂။ House Robber

အိမ်တန်း `nums` — အိမ်တစ်လုံးစီမှာ ပိုက်ဆံ `nums[i]` ရှိ။ ဒါပေမယ့် **ကပ်လျက် အိမ် ၂ လုံး** ဖောက်ရင် alarm မြည်တယ်။ alarm မမြည်စေဘဲ — **ပိုက်ဆံ အများဆုံး** ဘယ်လောက် ရနိုင်လဲ။

```
Input: nums = [2,7,9,3,1]
Output: 12
   (အိမ် 0 (2) + အိမ် 2 (9) + အိမ် 4 (1) = 12 ;  ကပ်လျက် မဖောက်)
```

ရှင်းလင်းချက်

- **State:** `dp[i]` = အိမ် `0..i` ထဲက ဖောက်လို့ ရတဲ့ ပိုက်ဆံ အများဆုံး။
- **Transition:** အိမ် `i` အတွက် ရွေးချယ်စရာ ၂ ခု — (၁) **မဖောက်** → `dp[i-1]` (အရင်အထိ အကောင်းဆုံး)၊ (၂) **ဖောက်** → `dp[i-2] + nums[i]` (ကပ်လျက် မဖြစ်အောင် `i-1` ကျော်)။ ၂ ခုထဲက **max** ယူ → `dp[i] = max(dp[i-1], dp[i-2] + nums[i])` ။
- **Base case:** `dp[0] = nums[0]` ၊ `dp[1] = max(nums[0], nums[1])` ။

`nums = [2,7,9,3,1]` ကို လိုက်ကြည့်ရအောင် —

```
 i   nums[i]   dp[i-1]   dp[i-2]+nums[i]    dp[i] = max
 0     2         —          —               2
 1     7         2          —               max(2,7)        = 7
 2     9         7          2+9=11          max(7,11)       = 11
 3     3         11         7+3=10          max(11,10)      = 11
 4     1         11         11+1=12         max(11,12)      = 12  ✓
```

> **Time Complexity:** $O(n)$။
> **Space Complexity:** $O(1)$ - variable ၂ ခု (space optimized)။

Java Solution

```java
class Solution {
    public int rob(int[] nums) {
        int prev2 = 0, prev1 = 0;            // dp[i-2], dp[i-1]
        for (int num : nums) {
            int cur = Math.max(prev1, prev2 + num);  // မဖောက် vs ဖောက်
            prev2 = prev1;
            prev1 = cur;
        }
        return prev1;
    }
}
```

၃။ Coin Change

အကြွေစေ့ အမျိုးအစား `coins` (အကန့်အသတ်မဲ့ အရေအတွက်) နဲ့ ပမာဏ `amount` ပေးထားသည်။ `amount` ဖွဲ့ဖို့ **အကြွေစေ့ အရေအတွက် အနည်းဆုံး** ပြန်ပါ။ မဖွဲ့နိုင်ရင် `-1` ။

```
Input: coins = [1,3,4], amount = 6
Output: 2
```

```
    (3 + 3 = 6  → ၂ စေ့ ;  greedy ရဲ့ 4+1+1=၃ စေ့ ထက် ပိုကောင်း)
```

ရှင်းလင်းချက်

ဒါက အခန်း ၂၀ မှာ greedy ကျိုးခဲ့တဲ့ ပြဿနာပါ — DP နဲ့ ဖြေရင် မှန်တယ် (ဖြစ်နိုင်ခြေ အကုန် စစ်လို့)။

- **State:** `dp[a]` = ပမာဏ `a` ဖွဲ့ဖို့ အကြွေစေ့ အနည်းဆုံး။
- **Transition:** coin `c` တစ်ခုစီအတွက် — `a` ကို ဖွဲ့ဖို့ "coin `c` ယူပြီး ကျန် `a-c` ဖွဲ့" → `dp[a] = min(dp[a], dp[a-c] + 1)` ။
- **Base case:** `dp[0] = 0` (ပမာဏ ၀ ဖွဲ့ဖို့ ၀ စေ့)။ ကျန်တာ `∞` (မဖွဲ့နိုင်သေး) နဲ့ စ။

`coins = [1,3,4]` , `amount = 6` ကို dp ဖြည့်ကြည့်ရအောင် —

```
a:     0   1   2   3   4   5   6
dp:    0   1   2   1   1   2   2
                       ↑       ↑
dp[3] = dp[0]+1 = 1  (coin 3)
dp[6] = min(dp[5]+1, dp[3]+1, dp[2]+1) = min(3, 2, 3) = 2  (coin 3)  ✓
```

> **Time Complexity:** $O(amount \times coins)$ - ပမာဏ တစ်ခုစီ × coin တစ်ခုစီ။
> **Space Complexity:** $O(amount)$ - dp array။

Java Solution

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);           // "∞" အစား — ဘယ်တော့မှ မရောက်နိုင်တဲ့ ကြီးတန်
        dp[0] = 0;                             // base case
        for (int a = 1; a <= amount; a++) {
            for (int c : coins) {
                if (c <= a) {                  // coin က ပမာဏထက် မကြီးမှ
                    dp[a] = Math.min(dp[a], dp[a - c] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];  // မရောက်ရင် -1
    }
}
```

၄။ Maximum Subarray (Kadane's Algorithm)

integer array `nums` (အနုတ် ပါ) ပေးထားသည်။ **ဆက်တိုက် subarray** (အနည်းဆုံး ၁ ခု) ထဲက — **ပေါင်းလဒ် အများဆုံး** ပြန်ပါ။

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
   (subarray [4,-1,2,1] ရဲ့ ပေါင်းလဒ် = 6)
```

ရှင်းလင်းချက်

- **State:** `dp[i]` = index `i` မှာ **ဆုံးတဲ့** subarray ရဲ့ ပေါင်းလဒ် အများဆုံး။
- **Transition:** index `i` မှာ ဆုံးတဲ့ အကောင်းဆုံး subarray က — (၁) `nums[i]` တစ်ခုတည်း အသစ်စ၊ (၂) အရင် subarray (`dp[i-1]`) ကို ဆက်ပြီး `nums[i]` ပေါင်း — ၂ ခုထဲက max → `dp[i] = max(nums[i], dp[i-1] + nums[i])` ။ (`dp[i-1]` က နုတ်ဖြစ်နေရင် — ဆက်တာ နစ်နာလို့ အသစ်စ ပိုကောင်း)။
- **Base case:** `dp[0] = nums[0]` ။ answer က `dp[]` အကုန်ရဲ့ **max** (နောက်ဆုံး dp မဟုတ်)။

`[-2,1,-3,4,-1,2,1,-5,4]` ကို လိုက်ကြည့်ရအောင် —

```
nums:   -2   1   -3   4   -1   2   1   -5   4
dp:     -2   1   -2   4    3   5   6    1   5
                          └─────────┘

dp[i] = max(nums[i], dp[i-1]+nums[i])
global max = 6  (dp[6])  ✓
```

> **Time Complexity:** $O(n)$ - တစ်ခေါက်ပဲ ဖြတ်။
> **Space Complexity:** $O(1)$ - `cur` နဲ့ `best` ၂ ခုပဲ (space optimized)။

Java Solution

```java
class Solution {
    public int maxSubArray(int[] nums) {
        int cur = nums[0], best = nums[0];      // dp[0] = nums[0]
        for (int i = 1; i < nums.length; i++) {
            cur = Math.max(nums[i], cur + nums[i]);  // အသစ်စ vs ဆက်
            best = Math.max(best, cur);         // global max
        }
        return best;
    }
}
```

၅။ Decode Ways

`'A'→"1"` , `'B'→"2"` , …, `'Z'→"26"` ဆိုတဲ့ mapping နဲ့ — ဂဏန်း string `s` ကို ပေးထားသည်။ `s` ကို စာလုံး အဖြစ် **decode လုပ်နိုင်တဲ့ နည်းလမ်း ဘယ်နှ မျိုး** ရှိလဲ ပြန်ပါ။ (`'0'` က သီးသန့် စာလုံး မရှိ — `10` , `20` အဖြစ်သာ)။

```
Input: s = "226"
Output: 3
   ("2 2 6"=BBF ;  "22 6"=VF ;  "2 26"=BZ  → ၃ နည်း)
```

ရှင်းလင်းချက်

- **State:** `dp[i]` = string ရဲ့ ပထမ `i` လုံး (`s[0..i-1]`) ကို decode လုပ်နိုင်တဲ့ နည်းလမ်း အရေအတွက်။

- **Transition:** လုံး `i` အတွက် — (၁) `s[i-1]` က `'0'` မဟုတ်ရင် (၁ လုံး တစ်ခုတည်း `1..9`) → `dp[i] += dp[i-1]` ၊ (၂) `s[i-2..i-1]` က `10..26` ကြားဆို (၂ လုံး တွဲ) → `dp[i] += dp[i-2]` ။
- **Base case:** `dp[0] = 1` (string ဗလာ — ၁ နည်း)၊ `dp[1] = (s[0] != '0') ? 1 : 0` ။

`s = "226"` ကို လိုက်ကြည့်ရအောင် —

```
dp[0] = 1  (ဗလာ)
dp[1] = 1  ("2" → B)
dp[2]:  "2" (≠0) → += dp[1]=1 ;  "22" (10..26) → += dp[0]=1  → dp[2]=2
dp[3]:  "6" (≠0) → += dp[2]=2 ;  "26" (10..26) → += dp[1]=1  → dp[3]=3  ✓
```

> **Time Complexity:** $O(n)$ - လုံးတစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(1)$ - variable ၂ ခု (space optimized)။

Java Solution

```
class Solution {
    public int numDecodings(String s) {
        if (s.charAt(0) == '0') return 0;      // '0' နဲ့ စ → decode မရ
        int prev2 = 1, prev1 = 1;              // dp[0]=1, dp[1]=1
        for (int i = 2; i <= s.length(); i++) {
            int cur = 0;
            if (s.charAt(i - 1) != '0') cur += prev1;          // ၁ လုံး (1..9)
            int two = Integer.parseInt(s.substring(i - 2, i)); // ၂ လုံး တွဲ
            if (two >= 10 && two <= 26) cur += prev2;          // (10..26)
            prev2 = prev1;
            prev1 = cur;
        }
        return prev1;
    }
}
```

အခန်း ၂၃ - 2-D Dynamic Programming

အခန်း ၂၂ မှာ — `dp[i]` ဆိုတဲ့ **၁ တန်း (1-D)** state နဲ့ ပြဿနာတွေ ဖြေခဲ့ပါတယ် ("index `i` ထိ အကောင်းဆုံး")။ ဒါပေမယ့် ပြဿနာ အများကြီးမှာ — အဖြေ တစ်ခုကို သတ်မှတ်ဖို့ **အညွှန်း ၂ ခု** လိုတယ် —

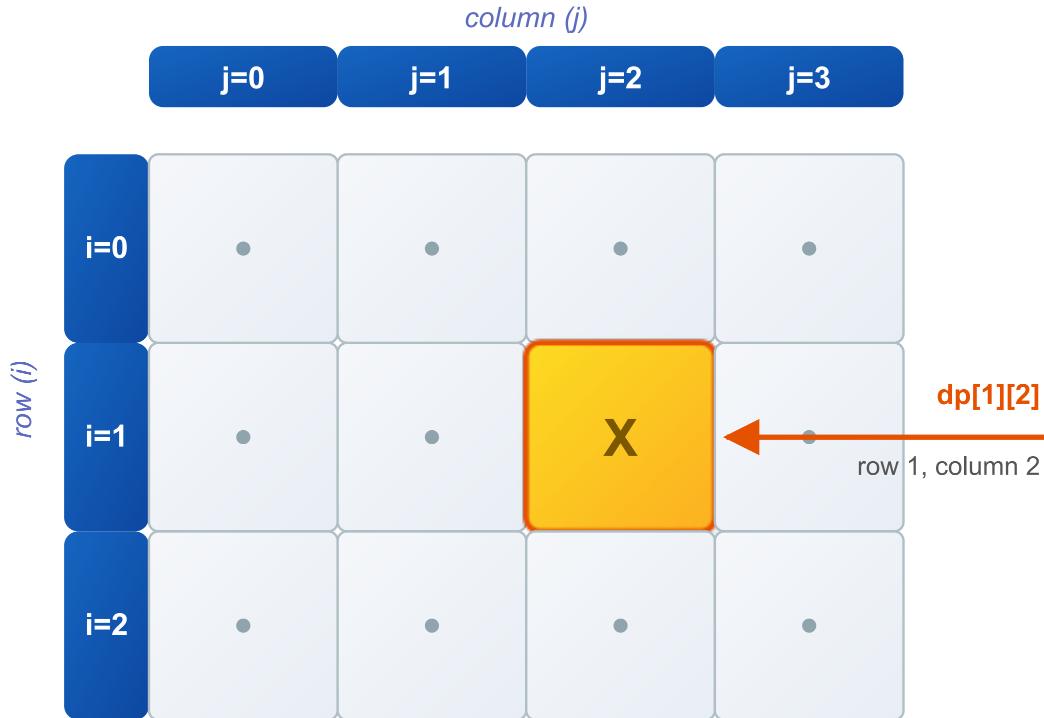
- **Grid** ပေါ်မှာ လမ်းကြောင်း ရှာတာ — "row `i` , column `j` ထိ" (အညွှန်း ၂ ခု)။
- **String ၂ ခု** နှိုင်းတာ — "string A ရဲ့ ပထမ `i` လုံး၊ string B ရဲ့ ပထမ `j` လုံး" (အညွှန်း ၂ ခု)။

ဒီလို ပြဿနာတွေကို **2-D DP** နဲ့ ဖြေတယ် — state က `dp[i][j]` ဆိုတဲ့ **ဇယား (table)** ဖြစ်လာတယ်။ 2-D DP က 1-D ထက် ခက်တယ် ထင်ရပေမယ့် — တကယ်တော့ **table ကို ပုံဆွဲပြီး** စဉ်းစားရင် အလွန် မြင်သာတယ်။ ဒီအခန်းမှာ — table ဖွဲ့စည်းပုံ၊ transition ဦးတည်ချက် (အပေါ်/ဘယ်/ထောင့်ဖြတ်)၊ ပြီးတော့ **Grid DP, String DP, Knapsack** ဆိုတဲ့ classic ပြဿနာ ၅ ခု ဖြေသွားပါမယ်။

2-D State နဲ့ DP Table

2-D DP ရဲ့ state က `dp[i][j]` — အညွှန်း ၂ ခုနဲ့ သတ်မှတ်တဲ့ table တစ်ခုပါ။ `i` က **row (အတန်း)**၊ `j` က **column (ကော်လံ)**။ `dp[i][j]` ဆိုတာ "row `i` , column `j` အခြေအနေထိ ရောက်တဲ့ အကောင်းဆုံး အဖြေ" ကို ကိုယ်စားပြုတယ်။

2-D DP Table (m × n) — dp[i][j]

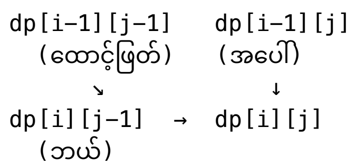


cell တစ်ခုစီ = "row i, column j အခြေအနေထိ အကောင်းဆုံး အဖြေ" — အိမ်နီးချင်း cell တွေကနေ တွက်

1-D DP မှာ "index `i` ထိ" တွက်သလို — 2-D DP မှာ **cell တစ်ခုစီကို — အရင် cell တွေ (အိမ်နီးချင်း) ကနေ တွက်ပြီး table ကို ဖြည့်တက်**တာပါ။

Transition ဦးတည်ချက် — အပေါ်/ဘယ်/ထောင့်ဖြတ်

2-D DP ရဲ့ အနှစ်သာရက — `dp[i][j]` ကို **ဘယ် cell တွေကနေ တွက်လဲ** ဆိုတာပါ။ အများဆုံး အသုံးပြုတဲ့ ၃ ဦးတည်ချက် —



- **အပေါ်** (`dp[i-1][j]`) — အပေါ်တန်းကနေ ဆင်းလာတာ။
- **ဘယ်** (`dp[i][j-1]`) — ဘယ်ဘက်ကနေ လာတာ။
- **ထောင့်ဖြတ်** (`dp[i-1][j-1]`) — ဖြတ်လာတာ။

ပြဿနာ အလိုက် — ဒီ ၃ ခုထဲက ဘယ်ဟာ သုံးမလဲ ကွဲတယ်။ Grid path က **အပေါ် + ဘယ်**၊ string နှိုင်းတာတွေက **ထောင့်ဖြတ်** ပါ များတယ်။ ဒါက 2-D DP ပြဿနာ ဖြေတဲ့အခါ အရင်ဆုံး ရှာရမယ့် အချက်ပါ။

Grid DP vs String DP

2-D DP ပြဿနာ အများစုက အမျိုးအစား ၂ မျိုး —

- **Grid DP** — `dp[i][j]` က **grid cell** **`(i,j)`** ကို တိုက်ရိုက် ကိုယ်စားပြု (Unique Paths, Min Path Sum)။ table size = grid size။
- **String DP** — `dp[i][j]` က **"string A ရဲ့ ပထမ `i` လုံး နဲ့ string B ရဲ့ ပထမ `j` လုံး"** ကို ကိုယ်စားပြု (LCS, Edit Distance)။ `i=0` / `j=0` က "string ဗလာ" ကို ကိုယ်စားပြုလို့ — table size = `(m+1) × (n+1)` ။

> **မှတ်ရန်:** String DP မှာ row/column ကို `0` ကစ ရေတွက်တဲ့အခါ — `dp[i][j]` ရဲ့ `i` က "string A ရဲ့ ပထမ `i` **လုံး**" (index မဟုတ်) ကို ဆိုလိုတယ်။ ဒါကြောင့် character ကို ယူဉ်တဲ့အခါ `A[i-1]` နဲ့ `B[j-1]` (၁ နုတ်) ကို သုံးရတယ်။ ဒီ "off-by-one" က string DP ရဲ့ အမှား အဖြစ်ဆုံး အချက်ပါ။

Space Optimization — Overview

2-D DP table က space $O(m \times n)$ ယူတယ်။ ဒါပေမယ့် `dp[i][j]` ကို တွက်ဖို့ **အပေါ်တန်း (`i-1`) နဲ့ လက်ရှိတန်း (`i`)** ၂ တန်းပဲ လိုလေ့ ရှိတယ်။ ဒါဆို — table တစ်ခုလုံး မသိမ်းဘဲ — **တန်း ၂ ခု** (ဒါမှ မဟုတ် **၁ တန်း** ကို သတိနဲ့ overwrite) ပဲ သိမ်းရင် — space ကို $O(n)$ အထိ လျှော့လို့ ရတယ်။

ဒါက "rolling array" technique ပါ။ Logic ပိုရှုပ်လို့ — ဒီအခန်းမှာ ပြဿနာ အများစုကို **full 2-D table** နဲ့ ပြ (နားလည် လွယ်အောင်)၊ ပြီးမှ optimization ကို မှတ်ချက် အဖြစ် ထည့်ပါမယ်။

Real-world Examples

- **Text Comparison / Diff Tool** — `git diff` , editor ရဲ့ "ဘယ်စာကြောင်း ပြောင်းသွားလဲ" — string ၂ ခုကြား **Longest Common Subsequence / Edit Distance** ပေါ်မှာ အခြေခံတယ်။
- **Spell Check / Autocorrect** — "ရိုက်မိတဲ့ စာလုံး နဲ့ အနီးစပ်ဆုံး မှန်တဲ့ စာလုံး" — **Edit Distance** (ဘယ်နှ လုံး ပြင်ရမလဲ) နဲ့ တိုင်းတာ။
- **Route Cost in Grid** — warehouse / map grid ပေါ်မှာ "အကုန်ကျ အနည်းဆုံး လမ်း" — **Min Path Sum**။
- **Budget / Resource Allocation** — budget အကန့်အသတ်နဲ့ "value အများဆုံး ရအောင် ဘယ် item ရွေးမလဲ" — **0/1 Knapsack**။
- **DNA / Sequence Alignment** — bioinformatics မှာ DNA sequence ၂ ခု နှိုင်းတာ — LCS / Edit Distance ရဲ့ ကြီးထွားပုံ။

Questions

2-D DP ပြဿနာ ဖြေတဲ့အခါ — **table ကို ပုံဆွဲပြီး**၊ (၁) `dp[i][j]` ဘာလဲ (state)၊ (၂) ဘယ် cell ကနေ တွက်လဲ (transition)၊ (၃) ပထမ row/column (base case) — ဆိုတာ ရှာတာပါ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Unique Paths

`m × n` grid မှာ — ဘယ်အပေါ်ထောင့် `(0,0)` ကစ — **ညာ** ဒါမှမဟုတ် **အောက်** ပဲ ရွှေ့နိုင်ပြီး — ညာအောက်ထောင့် `(m-1,n-1)` ထိ ရောက်ဖို့ **ကွဲပြားတဲ့ လမ်း ဘယ်နှ မျိုး** ရှိလဲ။

```
Input: m = 3, n = 3
Output: 6
```

ရှင်းလင်းချက်

- **State:** `dp[i][j]` = `(0,0)` ကနေ cell `(i,j)` ထိ ရောက်တဲ့ လမ်း အရေအတွက်။
- **Transition:** cell `(i,j)` ကို — **အပေါ်** `(i-1,j)` ကနေ (အောက်ဆင်း) ဒါမှမဟုတ် **ဘယ်** `(i,j-1)` ကနေ (ညာသွား) ရောက်နိုင်လို့ → `dp[i][j] = dp[i-1][j] + dp[i][j-1]` ။
- **Base case:** ပထမ row နဲ့ ပထမ column အကုန် `1` (လမ်းကြောင်း တစ်ခုတည်း — တန်းသွား ဒါမှမဟုတ် တန်းဆင်း)။

`m = 3, n = 3` table ဖြည့်ကြည့်ရအောင် —

```
      j=0  j=1  j=2
i=0  [ 1 ][ 1 ][ 1 ]      ← ပထမ row အကုန် 1
i=1  [ 1 ][ 2 ][ 3 ]      dp[1][1] = 1+1 = 2,  dp[1][2] = 1+2 = 3
i=2  [ 1 ][ 3 ][ 6 ]      dp[2][2] = dp[1][2] + dp[2][1] = 3+3 = 6 ✓
```

> **Time Complexity:** $O(m \times n)$ - cell တစ်ခုစီ တစ်ခါ။
> **Space Complexity:** $O(m \times n)$ - table (rolling array နဲ့ $O(n)$ လျှော့နိုင်)။

Java Solution

```java
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i = 0; i < m; i++) dp[i][0] = 1;      // ပထမ column
        for (int j = 0; j < n; j++) dp[0][j] = 1;      // ပထမ row
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]; // အပေါ် + ဘယ်
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

၂။ Minimum Path Sum

`m × n` grid မှာ cell တစ်ခုစီ တန်ဖိုး (cost) ရှိ။ ဘယ်အပေါ်ထောင့်ကစ — **ညာ/အောက်** ပဲ ရွှေ့ပြီး — ညာအောက်ထောင့်ထိ — **ဖြတ်သွားတဲ့ cell တွေရဲ့ ပေါင်းလဒ် အနည်းဆုံး** ပြန်ပါ။

```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
   (လမ်း 1→3→1→1→1 = 7)
```

ရှင်းလင်းချက်

Unique Paths နဲ့ တူ — ကွာတာက "လမ်း ရေတွက်" မဟုတ်ဘဲ "cost အနည်းဆုံး" ။

- **State:** `dp[i][j] = (0,0)` ကနေ `(i,j)` ထိ — ပေါင်းလဒ် အနည်းဆုံး။
- **Transition:** `dp[i][j] = grid[i][j] + min(dp[i−1][j], dp[i][j−1])` — အပေါ်နဲ့ ဘယ် ၂ ခုထဲက **အသက်သာတာ** ရွေး။
- **Base case:** `dp[0][0] = grid[0][0]` ၊ ပထမ row/column က — တစ်ဘက်တည်းကပဲ လာနိုင်လို့ ဆက်ပေါင်း။

`grid = [[1,3,1],[1,5,1],[4,2,1]]` table —

```
grid:              dp (ပေါင်းလဒ် အနည်းဆုံး):
1 3 1              1 4 5
1 5 1     →        2 7 6
4 2 1              6 8 7
dp[2][2] = 1 + min(dp[1][2]=6, dp[2][1]=8) = 1+6 = 7 ✓
```

> **Time Complexity:** $O(m \times n)$။
> **Space Complexity:** $O(m \times n)$ - table (in-place ဆို $O(1)$)။

Java Solution

```java
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        int[][] dp = new int[m][n];
        dp[0][0] = grid[0][0];
        for (int j = 1; j < n; j++) dp[0][j] = dp[0][j - 1] + grid[0][j];  // ပထမ row
        for (int i = 1; i < m; i++) dp[i][0] = dp[i - 1][0] + grid[i][0];  // ပထမ column
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = grid[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

၃။ Longest Common Subsequence (LCS)

string ၂ ခု `text1` , `text2` ပေးထားသည်။ ၂ ခုစလုံးမှာ ပါတဲ့ — **အရှည်ဆုံး common subsequence** (အစဉ်အတိုင်း၊ ဆက်တိုက် မဖြစ်လည်း ရ) ရဲ့ **အရှည်** ပြန်ပါ။

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
   (common subsequence "ace" — အရှည် 3)
```

ရှင်းလင်းချက်

ဒါက **String DP** ပါ — table size `(m+1) × (n+1)` (row/column `0` = string ဗလာ)။

- **State:** `dp[i][j] = text1` ရဲ့ ပထမ `i` လုံး နဲ့ `text2` ရဲ့ ပထမ `j` လုံးကြား LCS အရှည်။
- **Transition:**
  - **စာလုံး တူ** (`text1[i-1] == text2[j-1]`) → **ထောင့်ဖြတ် + 1** → `dp[i][j] = dp[i-1][j-1] + 1` ။
  - **မတူ** → အပေါ်နဲ့ဘယ် ၂ ခုထဲက **max** → `dp[i][j] = max(dp[i-1][j], dp[i][j-1])` ။
- **Base case:** `dp[0][*] = 0` ၊ `dp[*][0] = 0` (string ဗလာနဲ့ LCS = 0)။

`text1 = "abcde"` , `text2 = "ace"` table —

LCS DP Table — text1 = "abcde", text2 = "ace"

စာလုံး တူရင် → ထောင့်ဖြတ် dp[i-1][j-1] + 1

text2 (j) / *text1 (i)*

	""	a	c	e
""	0	0	0	0
a	0	**1**	1	1
b	0	1	1	1
c	0	1	**2**	2
d	0	1	2	2
e	0	1	2	**3**

ထောင့်ဖြတ် Match

a == a → dp[0][0]+1 = 1

c == c → dp[2][1]+1 = 2

e == e → dp[4][2]+1 = 3

common subsequence
"ace" → LCS = 3 ✓

မတူရင် → max(အပေါ်, ဘယ်)
cell တိုင်း = ထိုင်းမှန်နေတဲ့ အကောင်းဆုံး LCS

> **Time Complexity:** $O(m \times n)$။
> **Space Complexity:** $O(m \times n)$ - table။

Java Solution

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length(), n = text2.length();
        int[][] dp = new int[m + 1][n + 1];        // row/col 0 = ဗလာ (base = 0)
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;           // တူ → ထောင့်ဖြတ်+1
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  // မတူ → max
                }
            }
        }
        return dp[m][n];
    }
}
```

၄။ Edit Distance

string `word1` ကို `word2` ဖြစ်အောင် ပြောင်းဖို့ — **operation အနည်းဆုံး** ပြန်ပါ။ operation ၃ မျိုး — **insert** (ထည့်)၊ **delete** (ဖျက်)၊ **replace** (အစားထိုး)။

```
Input: word1 = "horse", word2 = "ros"
Output: 3
   (horse → rorse (replace h→r) → rose (delete r) → ros (delete e))
```

ရှင်းလင်းချက်

String DP — "spell check / diff tool" ရဲ့ အနှစ်သာရ။

- **State:** `dp[i][j]` = `word1` ပထမ `i` လုံးကို `word2` ပထမ `j` လုံး ဖြစ်အောင် — operation အနည်းဆုံး။
- **Transition:**
  - **စာလုံး တူ** → ဘာမှ မလို → `dp[i][j] = dp[i-1][j-1]` (ထောင့်ဖြတ်)။
  - **မတူ** → operation ၃ မျိုးထဲက **min + 1**:
    - **replace** → `dp[i-1][j-1]` (ထောင့်ဖြတ်)
    - **delete** → `dp[i-1][j]` (အပေါ်)
    - **insert** → `dp[i][j-1]` (ဘယ်)
- **Base case:** `dp[i][0] = i` (လုံး `i` ခု ဖျက်)၊ `dp[0][j] = j` (လုံး `j` ခု ထည့်)။

`word1 = "horse"` , `word2 = "ros"` table —

Edit Distance DP Table — word1 = "horse", word2 = "ros"

တူရင် → ထောင့်ဖြတ် · မတူရင် → 1 + min(အပေါ်, ဘယ်, ထောင့်ဖြတ်)

word2 (j) / *word1 (i)*

	""	r	o	s
""	0	1	2	3
h	1	1	2	3
o	2	2	**1**	2
r	3	2	2	2
s	4	3	3	2
e	5	4	4	**3**

Edit Distance Rules

base — dp[i][0]=i (ဖျက်)
dp[0][j]=j (ထည့်)

o == o
→ dp[i-1][j-1] = 1

မတူ → 1 + min(...)
အပေါ် · ဘယ် · ထောင့်ဖြတ်

horse → ros
dp[5][3] = 3 steps ✓

cell တိုင်း = အဲ့ အခြေအနေထိ လိုအပ်တဲ့ operation အနည်းဆုံး

> **Time Complexity:** $O(m \times n)$။
> **Space Complexity:** $O(m \times n)$ - table။

Java Solution

```java
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length(), n = word2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 0; i <= m; i++) dp[i][0] = i;  // base — i လုံး ဖျက်
        for (int j = 0; j <= n; j++) dp[0][j] = j;  // base — j လုံး ထည့်
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];      // တူ → ဘာမှ မလို
                } else {
                    dp[i][j] = 1 + Math.min(dp[i - 1][j - 1],         // replace
                                   Math.min(dp[i - 1][j],            // delete
                                            dp[i][j - 1]));          // insert
                }
```

```
            }
        }
        return dp[m][n];
    }
}
```

၅။ 0/1 Knapsack

item `n` ခု — တစ်ခုစီမှာ အလေးချိန် `weight[i]` နဲ့ တန်ဖိုး `value[i]` ရှိ။ အိတ် (knapsack) က အလေးချိန် `W` ထိ ဆံ့။ item တစ်ခုကို **ယူ ဒါမှမဟုတ် မယူ** (၂ ပိုင်း မဖြတ်) ဆိုရင် — အိတ်ထဲ ထည့်လို့ ရတဲ့ **တန်ဖိုး စုစုပေါင်း အများဆုံး** ပြန်ပါ။

```
Input: weight = [1,3,4,5], value = [1,4,5,7], W = 7
Output: 9
   (item 1 (w3,v4) + item 2 (w4,v5) = အလေး 7, တန်ဖိုး 9)
```

ရှင်းလင်းချက်

ဒါက "budget အကန့်အသတ်နဲ့ value အများဆုံး" ပြဿနာ — `dp[i][w]` table နဲ့ ဖြေတယ်။

- **State:** `dp[i][w]` = ပထမ `i` ခု item ထဲက ရွေးပြီး — အလေးချိန် `w` ထိ — တန်ဖိုး အများဆုံး။
- **Transition:** item `i` အတွက် ၂ ရွေး —
  - **မယူ** → `dp[i-1][w]` (အရင် item တွေ အတိုင်း)။
  - **ယူ** (weight ဆံ့မှ — `weight[i-1] <= w`) → `dp[i-1][w - weight[i-1]] + value[i-1]` ။
  - ၂ ခုထဲက **max** → `dp[i][w] = max(မယူ, ယူ)` ။
- **Base case:** `dp[0][*] = 0` (item မရှိ → တန်ဖိုး ၀)။

`weight=[1,3,4,5]` **,** **`value=[1,4,5,7]`** **,** **`W=7`** **—** `dp[i][w]` table (နောက်ဆုံး `dp[4][7]` = 9) ဖြည့်သွားတယ်။

Time Complexity: $O(n \times W)$ - item × weight။
Space Complexity: $O(n \times W)$ - table (rolling array နဲ့ $O(W)$ လျှော့နိုင်)။

Java Solution

```java
class Solution {
    public int knapsack(int[] weight, int[] value, int W) {
        int n = weight.length;
        int[][] dp = new int[n + 1][W + 1];         // row 0 = item မရှိ (base = 0)
        for (int i = 1; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                dp[i][w] = dp[i - 1][w];             // မယူ
                if (weight[i - 1] <= w) {           // ဆံ့မှ — ယူလို့ ရ
                    dp[i][w] = Math.max(dp[i][w],
                            dp[i - 1][w - weight[i - 1]] + value[i - 1]);  // ယူ
                }
            }
        }
```

```
        return dp[n][W];
    }
}
```

အခန်း ၂၄ - Bit Manipulation

အရင်အခန်းတွေမှာ data ကို array, hash table, tree စတဲ့ structure တွေနဲ့ ကိုင်တွယ်ခဲ့ပါတယ်။ ဒါပေမယ့် computer ထဲမှာ ဒီ data အကုန်လုံးဟာ နောက်ဆုံးမှာ **bit** (`0` နဲ့ `1`) တွေ အဖြစ်သာ ရှိနေတာပါ။ **Bit Manipulation** ဆိုတာ — အဲ့ဒီ bit တွေကို တိုက်ရိုက် ကိုင်တွယ်တဲ့ နည်းပညာပါ။

Bit manipulation ကို နေ့စဉ် app development မှာ အမြဲ မသုံးပေမယ့် — **permission / role flags, feature flag, compact settings, low-level optimization** စတဲ့ နေရာတွေမှာ အလွန် အသုံးဝင်တယ်။ ဥပမာ — user တစ်ယောက်ရဲ့ permission ၈ မျိုးကို — boolean ၈ ခု အစား — **integer တစ်ခုတည်း**နဲ့ သိမ်းလို့ ရတယ် (bit ၈ လုံး)။ ဒီအခန်းမှာ — binary အခြေခံ၊ bitwise operator တွေ၊ bit mask လုပ်ဆောင်ချက်တွေ၊ ပြီးတော့ classic ပြဿနာ ၅ ခု ဖြေသွားပါမယ်။

Binary အခြေခံ

Computer က ဂဏန်းတွေကို **binary (base-2)** — `0` နဲ့ `1` တွေနဲ့ ကိုယ်စားပြုတယ်။ position တစ်ခုစီက `2` ရဲ့ ထပ်ကိန်း တန်ဖိုး —

```
ဒသမ (decimal) 13  →  binary 1101
     1      1      0      1
     ×8     ×4     ×2     ×1      ← 2³ 2² 2¹ 2⁰
     8  +   4  +   0  +   1   =  13
```

bit တစ်လုံးစီကို **position (index)** နဲ့ ရည်ညွှန်းတယ် — အညာဆုံး (least significant) က position `0` ။ ဥပမာ `1101` မှာ — position 0, 2, 3 က `1` ၊ position 1 က `0` ။

Bitwise Operators

bit တွေကို ကိုင်တွယ်တဲ့ အခြေခံ operator ၆ ခု —

AND (`&`), OR (`|`), XOR (`^`)

bit ၂ လုံးကို position အလိုက် နှိုင်းတာ —

```
a = 5 = 0101
b = 3 = 0011

a & b  (AND — ၂ ခုလုံး 1 မှ 1):   0001 = 1
a | b  (OR  — တစ်ခုခု 1 ဆို 1):    0111 = 7
a ^ b  (XOR — မတူမှ 1):            0110 = 6
```

- **AND (`&`)** — bit ၂ လုံးစလုံး `1` မှ `1` ။ "bit စစ်ထုတ် (mask)" ဖို့ သုံး။

- **OR (`|`)** — တစ်ခုခု `1` ဆို `1` ။ "bit ဖွင့် (set)" ဖို့ သုံး။
- **XOR (`^`) — မတူမှ `1`** (တူရင် `0`)။ ဂုဏ်သတ္တိ အရေးကြီး ၂ ခု — `a ^ a = 0` (ကိုယ့်ကိုယ်ကို XOR = 0)၊ `a ^ 0 = a` ။ ဒါက "single number" လို ပြဿနာတွေရဲ့ သော့ချက်ပါ။

NOT (`~`)

bit အကုန်လုံးကို ပြောင်းပြန်လှန် (`0↔1`)။ signed integer မှာ — sign bit ပါ ပြောင်းလို့ ရလဒ်က နှုတ်ကိန်း ဖြစ်တယ် (two's complement)။

```
~5  →  ...11111010  =  −6   (two's complement: ~x = −x − 1)
```

Left Shift (`<<`), Right Shift (`>>`)

bit တွေကို ဘယ်/ညာ ရွှေ့တာ —

```
5 << 1  →  0101 → 1010 = 10   (ဘယ်ရွှေ့ ၁ = × 2)
5 << 2  →  0101 → 10100 = 20  (ဘယ်ရွှေ့ 2 = × 4)
5 >> 1  →  0101 → 0010 = 2    (ညာရွှေ့ ၁ = ÷ 2, အကြွင်း ပစ်)
```

- **Left shift `n << k`** = `n × 2^k` ။ `1 << k` က — position `k` မှာ bit တစ်လုံးတည်း `1` ဖြစ်တဲ့ ဂဏန်း (2^k) — bit mask ဆောက်ဖို့ အရေးကြီး။
- **Right shift `n >> k`** = `n ÷ 2^k` ။

Bit Mask — Bit တစ်လုံးချင်း ကိုင်တွယ်ခြင်း

ဂဏန်းတစ်ခုရဲ့ **position `i` က bit** ကို — `1 << i` ဆိုတဲ့ "mask" နဲ့ ပေါင်းပြီး ကိုင်တွယ်တယ်။ အခြေခံ လုပ်ဆောင်ချက် ၄ ခု —

```
mask = 1 << i      (position i မှာ bit တစ်လုံး 1)

Check  (bit ဖွင့်ထား/မထား):   (n >> i) & 1      → 1 ဆို ဖွင့်ထား
Set    (bit ဖွင့်):            n | (1 << i)      → position i ကို 1
Clear  (bit ပိတ်):             n & ~(1 << i)     → position i ကို 0
Toggle (ပြောင်းပြန်):          n ^ (1 << i)      → 0↔1 လှန်
```

```
n = 1010 (=10),  i = 0:
  Check:  (1010 >> 0) & 1 = 0    → position 0 ပိတ်ထား
  Set:    1010 | 0001 = 1011     → position 0 ဖွင့် (=11)
n = 1010,  i = 1:
  Clear:  1010 & ~0010 = 1000    → position 1 ပိတ် (=8)
  Toggle: 1010 ^ 0010 = 1000     → position 1 လှန် (=8)
```

Permission Flags — အသုံးအများဆုံး Real-world Pattern

bit manipulation ရဲ့ အသုံးအဝင်ဆုံး real-world pattern က **permission / feature flag** ပါ။ permission တစ်ခုစီကို **bit တစ်လုံး** အဖြစ် သတ်မှတ်ပြီး — integer တစ်ခုတည်းနဲ့ permission အများကြီး သိမ်းတယ်။

```
READ  = 1 = 001       (bit 0)
WRITE = 2 = 010       (bit 1)
EXEC  = 4 = 100       (bit 2)

user permission = READ | WRITE = 001 | 010 = 011 (=3)

"WRITE ရှိလား?"    permission & WRITE  != 0  →  ရှိ ✓
"EXEC ထည့်":        permission | EXEC        →  111 (=7)
"READ ဖြုတ်":       permission & ~READ       →  010 (=2)
```

ဒီနည်းက — boolean field အများကြီး အစား integer (`int` = bit 32 လုံး) တစ်ခုနဲ့ flag ၃၂ မျိုး သိမ်းနိုင်လို့ — database column, network packet, config setting တွေမှာ နေရာ အလွန် သက်သာတယ် (Linux file permission `chmod 755` ဟာ ဒီ pattern ပါ)။

Real-world Examples

- **Role / Permission System** — user role ကို `READ|WRITE|DELETE...` bit flag integer တစ်ခုနဲ့ သိမ်း၊ permission စစ်တာ `&` တစ်ချက်နဲ့ ($O(1)$)။
- **Feature Flags** — app ရဲ့ feature ၃၂ မျိုးကို integer တစ်ခုနဲ့ on/off — A/B testing, gradual rollout။
- **Compact Settings** — IoT device, game state, network protocol မှာ — setting အများကြီးကို byte နည်းနည်းနဲ့ pack လုပ်တာ (bandwidth/memory ချွေတာ)။
- **Low-level Optimization** — `× 2` အစား `<< 1` ၊ `% 2` အစား `& 1` ၊ hash function, graphics, cryptography မှာ bit trick။
- **Bitset / Bloom Filter** — element ရှိမရှိ စစ်ဖို့ bit array သုံးတာ (database, cache)။

Questions

Bit manipulation ပြဿနာ ဖြေတဲ့အခါ — ဂဏန်းကို **binary အဖြစ် မြင်**ပြီး — XOR ဂုဏ်သတ္တိ (`a^a=0`)၊ `n & (n-1)` (အညာဆုံး bit ဖျက်)၊ `1 << i` (mask) စတဲ့ trick တွေ သုံးတာပါ။ classic ၅ ခု ဖြေကြည့်ရအောင်။

၁။ Single Number

integer array `nums` — element တိုင်း **၂ ခါစီ** ပါ၊ တစ်ခုတည်းသာ **၁ ခါ** ပါတယ်။ အဲ့ဒီ တစ်ခါပဲ ပါတဲ့ element ကို ရှာပါ။ (extra memory မသုံးဘဲ $O(n)$)။

```
Input: nums = [4,1,2,1,2]
Output: 4
```

ရှင်းလင်းချက်

XOR ဂုဏ်သတ္တိ: `a ^ a = 0` နဲ့ `a ^ 0 = a` ။ array အကုန်ကို XOR ပေါင်းလိုက်ရင် — ၂ ခါစီ ပါတဲ့ element တွေ အချင်းချင်း ဖျက်ပြီး (`x ^ x = 0`) — တစ်ခါပဲ ပါတာ ကျန်တယ်။ hash table မလို၊ space $O(1)$။

`nums = [4,1,2,1,2]` ကို XOR —

```
0 ^ 4 = 4
4 ^ 1 = 5
5 ^ 2 = 7
7 ^ 1 = 6
6 ^ 2 = 4      ← 1,1 ဖျက်၊ 2,2 ဖျက် → 4 ကျန် ✓
```

> **Time Complexity:** $O(n)$ - တစ်ခေါက်ပဲ ဖြတ်။
> **Space Complexity:** $O(1)$ - variable တစ်ခုပဲ။

Java Solution

```
class Solution {
    public int singleNumber(int[] nums) {
        int single = 0;
        for (int x : nums) single ^= x;         // ၂ ခါ ပါတာ အချင်းချင်း ဖျက်
        return single;                          // တစ်ခါပဲ ပါတာ ကျန်
    }
}
```

၂။ Number of 1 Bits (Count Set Bits)

integer တစ်ခု ပေးထားသည်။ သူ့ binary မှာ `1` **bit ဘယ်နှ လုံး** ရှိလဲ ပြန်ပါ (Hamming weight)။

```
Input: n = 11  (binary 1011)
Output: 3
   (bit position 0, 1, 3 — '1' ၃ လုံး)
```

ရှင်းလင်းချက်

`n & (n - 1)` trick: ဒီ operation က — **အညာဆုံး `1` bit ကို `0` ပြောင်း**ပစ်တယ်။ ဒါကြောင့် — `n` က `0` မဖြစ်မချင်း ဒီ operation ထပ်လုပ်ပြီး၊ ဘယ်နှ ခါ လုပ်ရလဲ ရေတွက်ရင် — `1` bit အရေအတွက် ရတယ်။ bit အကုန် (၃၂ လုံး) loop မလိုဘဲ — `1` **bit အရေအတွက် အတိုင်း**ပဲ loop လုပ်လို့ ပိုမြန်တယ်။

`n = 11 (1011)` ကို လိုက်ကြည့်ရအောင် —

```
n = 1011,  n-1 = 1010  →  n & (n-1) = 1010  (count=1)
n = 1010,  n-1 = 1001  →  n & (n-1) = 1000  (count=2)
n = 1000,  n-1 = 0111  →  n & (n-1) = 0000  (count=3)
n = 0  →  ရပ်။  '1' bit = 3 ✓
```

> **Time Complexity:** $O(k)$ - `k = 1` bit အရေအတွက် (≤ 32)။
> **Space Complexity:** $O(1)$။

Java Solution

```
class Solution {
    public int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            n &= (n - 1);                    // အညာဆုံး '1' bit ဖျက်
            count++;
        }
        return count;
    }
}
```

၃။ Power of Two

integer `n` ပေးထားသည်။ `n` က **၂ ရဲ့ ထပ်ကိန်း** (`1, 2, 4, 8, ...`) ဖြစ်/မဖြစ် ပြန်ပါ။

```
Input: n = 16
Output: true   (16 = 2⁴)

Input: n = 6
Output: false  (6 = 2 × 3)
```

ရှင်းလင်းချက်

အဓိက အသိ: ၂ ရဲ့ ထပ်ကိန်းတွေက binary မှာ `1` **bit တစ်လုံးတည်း** ရှိတယ် (`1=0001, 2=0010, 4=0100, 8=1000`)။ ဒါကြောင့် — `n & (n-1) == 0` (အညာဆုံး `1` bit ဖျက်လိုက်ရင် `0` ကျန်) ဆို — `1` bit တစ်လုံးတည်း ရှိ = power of two။ (`n > 0` ပါ စစ်ရတယ် — `0` နဲ့ နုတ်ကိန်း မပါအောင်)။

```
16 = 10000,  15 = 01111  →  16 & 15 = 00000 = 0  → true ✓
6  = 00110,   5 = 00101  →   6 &  5 = 00100 ≠ 0  → false
```

> **Time Complexity:** $O(1)$ - operation ၂ ချက်ပဲ။
> **Space Complexity:** $O(1)$။

Java Solution

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;    // '1' bit တစ်လုံးတည်း
    }
}
```

၄။ Missing Number

`0` ကနေ `n` ထိ ဂဏန်း `n` ခု ထဲက — **တစ်ခု ပျောက်နေတဲ့** array `nums` (အရှည် `n`) ပေးထားသည်။ ပျောက်နေတဲ့ ဂဏန်း ပြန်ပါ။

```
Input: nums = [3,0,1]
Output: 2
   (0,1,2,3 ထဲက 2 ပျောက်)
```

ရှင်းလင်းချက်

XOR trick: `index` အကုန် (`0..n`) နဲ့ `nums` value အကုန်ကို XOR ပေါင်းလိုက်ရင် — ရှိတဲ့ ဂဏန်းတွေ အချင်းချင်း ဖျက် (value နဲ့ index တူတာ ဖျက်) ပြီး — **ပျောက်နေတဲ့ ဂဏန်း** ကျန်တယ်။ (`a^a=0` ဂုဏ်သတ္တိ — Single Number နဲ့ တူ)။ Sum နည်း (`n(n+1)/2 − sum`) နဲ့လည်း ရပေမယ့် — XOR က overflow မဖြစ်လို့ ပိုလုံခြုံ။

`nums = [3,0,1]` , `n = 3` —

```
res = 3 (=n)
i=0:  res ^ 0 ^ nums[0]=3  →  3 ^ 0 ^ 3 = 0
i=1:  res ^ 1 ^ nums[1]=0  →  0 ^ 1 ^ 0 = 1
i=2:  res ^ 2 ^ nums[2]=1  →  1 ^ 2 ^ 1 = 2   → 2 ✓
```

> **Time Complexity:** $O(n)$။
> **Space Complexity:** $O(1)$။

Java Solution

```java
class Solution {
    public int missingNumber(int[] nums) {
        int res = nums.length;                // n နဲ့ စ (index 0..n−1 အပြင် n ပါ)
        for (int i = 0; i < nums.length; i++) {
            res ^= i ^ nums[i];               // index နဲ့ value ၂ ခုလုံး XOR
        }
        return res;                           // ဖျက်လို့ မရတဲ့ (ပျောက်) ဂဏန်း ကျန်
    }
}
```

၅။ Subsets (using Bits)

ထပ်တူ မပါတဲ့ integer array `nums` (အရှည် `n`) ပေးထားသည်။ ဖြစ်နိုင်တဲ့ **subset အကုန်လုံး** (power set) ကို ပြန်ပါ။

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
   (subset 2³ = 8 ခု)
```

ရှင်းလင်းချက်

element `n` ခုရဲ့ subset က 2^n **ခု** ရှိ — element တစ်ခုစီအတွက် "ပါ (1) / မပါ (0)" ၂ ရွေး။ ဒါက **bit ပုံစံ**နဲ့ တိုက်ဆိုင်တယ်! `0` ကနေ `2^n - 1` ထိ ဂဏန်း တစ်ခုစီကို — **bitmask** အဖြစ် ကြည့်ရင် — bit `i` က `1` ဆို `nums[i]` ပါ၊ `0` ဆို မပါ — subset တစ်ခုစီ ရတယ်။

`nums = [1,2,3]` — mask `0..7` —

```
mask  binary   subset
0     000      []
1     001      [1]           (bit 0 ဖွင့်)
2     010      [2]
3     011      [1,2]
4     100      [3]
5     101      [1,3]
6     110      [2,3]
7     111      [1,2,3]
```

> **Time Complexity:** $O(n \times 2^n)$ - subset 2^n ခု × element `n` စစ်။
> **Space Complexity:** $O(n \times 2^n)$ - output (extra $O(1)$)။

Java Solution

```java
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        int n = nums.length;
        List<List<Integer>> result = new ArrayList<>();
        for (int mask = 0; mask < (1 << n); mask++) {   // 0 .. 2^n - 1
            List<Integer> subset = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                if ((mask & (1 << i)) != 0) subset.add(nums[i]);  // bit i ဖွင့် → ပါ
            }
            result.add(subset);
        }
        return result;
    }
}
```

နောက်ဆက်တွဲ ၁ - P, NP, NP-Hard, NP-Complete

ဒီစာအုပ်တစ်လျှောက် — ပြဿနာတစ်ခုကို **ပိုမြန်အောင်** ($O(n^2) \to O(n \log n) \to O(n)$) ဖြေနည်းတွေ လေ့လာခဲ့ပါတယ်။ ဒါပေမယ့် developer အနေနဲ့ တစ်ခါတစ်ရံ — **ဘယ်လို optimize လုပ်လုပ် မြန်အောင် မရတဲ့** ပြဿနာတွေ တွေ့ရတယ်။ ဥပမာ — "delivery truck တစ်စီးက မြို့ ၂၀ ကို အကုန်လည်ပြီး အတိုဆုံး လမ်းနဲ့ ပြန်ရောက်ဖို့" — ဒါက ရိုးရှင်းသလို ထင်ရပေမယ့် — တကယ်တမ်း computer အကြီးတွေတောင် အချိန်ကုန် မဖြေနိုင်တဲ့ ပြဿနာ ဖြစ်နေတယ်။

ဘာကြောင့်လဲ? ဒီနောက်ဆက်တွဲမှာ — **ဘယ် ပြဿနာတွေက "လွယ်" ပြီး ဘယ်ဟာတွေက တကယ် "ခက်" လဲ** ဆိုတဲ့ theory ကို — developer အတွက် လိုအပ်သလောက်သာ ရှင်းပြပါမယ်။ ဒါက အဓိက သင်ခန်းစာ တစ်ခု ပေးတယ် — **ပြဿနာ တိုင်းမှာ ပြီးပြည့်စုံတဲ့ မြန်တဲ့ အဖြေ မရှိ**ဘူး၊ တစ်ခါတစ်ရံ "**လုံလောက်တဲ့ အဖြေ (good enough)**" ကို လက်ခံတတ်ဖို့ လိုတယ်။

> **မှတ်ရန်:** ဒီအပိုင်းက main learning flow မဟုတ်ဘဲ — appendix ပါ။ academic proof တွေ မဟုတ်ဘဲ — "ဒီ ပြဿနာ ခက်လား၊ ခက်ရင် ဘာလုပ်မလဲ" ဆိုတာ ဆုံးဖြတ်နိုင်ဖို့ အဆင့်ထိပဲ ရှင်းပြပါမယ်။

Brute Force Explosion — ဘာကြောင့် တချို့ ပြဿနာ ခက်လဲ

ပြဿနာ တချို့ ခက်ရတဲ့ အကြောင်းရင်းက — **ဖြစ်နိုင်ခြေ (possibilities) အရေအတွက်က input ကြီးလာတာနဲ့ ပေါက်ကွဲ (explode)** သွားလို့ပါ။

မြို့ `n` မြို့ကို လည်တဲ့ လမ်းကြောင်း (Traveling Salesman) ကို ကြည့်ရအောင် — ဖြစ်နိုင်တဲ့ လမ်းကြောင်း အရေအတွက်က `n!` **(factorial)** —

မြို့ အရေအတွက် (n)	ဖြစ်နိုင်တဲ့ လမ်းကြောင်း (n!)	မှတ်ချက်
5	120	
10	3,628,800	
15	1,307,674,368,000	၁.၃ ထရီလီယံ
20	2,432,902,008,176,640,000	၂.၄ ဘီလီယံ ဘီလီယံ
25	15,511,210,043,330,985,984,000,000	ဂဏန်း ၂၆ လုံး!

`n = 25` ဆိုရင် — တစ်စက္ကန့်မှာ လမ်းကြောင်း သန်းပေါင်း ၁,၀၀၀ စစ်နိုင်တဲ့ computer နဲ့တောင် — **စကြဝဠာ သက်တမ်းထက် ကြာ**မယ်။ ဒါက "brute force explosion" — ဖြစ်နိုင်ခြေ အကုန် စစ်တဲ့ နည်းက input နည်းနည်း ကြီးတာနဲ့ လက်တွေ့ မဖြစ်နိုင်တော့တာပါ။

ဒီလို ပြဿနာတွေကို — computer scientist တွေက အုပ်စု ဖွဲ့ပြီး အမည် ပေးထားတယ် — **P, NP, NP-Hard, NP-Complete**။

P — လွယ်တဲ့ ပြဿနာ (အမြန် ဖြေနိုင်)

P (Polynomial time) = **polynomial time ($O(n)$, $O(n^2)$, $O(n^3)$ ...) နဲ့ ဖြေနိုင်တဲ့** ပြဿနာတွေ။ ဒီစာအုပ်က ပြဿနာ အများစုဟာ P ထဲ ပါတယ် —

- list ကို sort လုပ် ($O(n \log n)$)
- array ထဲ element ရှာ ($O(n)$ / $O(\log n)$)
- graph မှာ shortest path (Dijkstra — $O(E \log V)$)

P ပြဿနာတွေက — input ကြီးလာရင် အချိန် "သင့်တင့်စွာ" တိုးတယ် (ပေါက်ကွဲ မသွား)။ ဒါတွေက **"လက်တွေ့ ဖြေနိုင်တဲ့" (tractable)** ပြဿနာတွေပါ။

NP — အဖြေ မှန်မမှန် အမြန် စစ်နိုင်

NP (Nondeterministic Polynomial time) = **အဖြေ တစ်ခု ပေးထားရင် — အဲ့ဒါ မှန်မမှန်ကို polynomial time နဲ့ အမြန် စစ်နိုင်တဲ့** ပြဿနာတွေ။

အရေးကြီးတဲ့ ကွဲပြားချက်က — "**ဖြေဖို့**" ခက်ပေမယ့် "**စစ်ဖို့**" လွယ်တာ —

```
Sudoku ဥပမာ —
ဖြေဖို့:    ဆဲလ် အကုန် မှန်အောင် ဖြည့်ဖို့ — ဖြစ်နိုင်ခြေ အများကြီး စစ်ရ (ခက်)
စစ်ဖို့:    ဖြည့်ပြီးသား Sudoku တစ်ခု မှန်မမှန် — တန်း/ကော်လံ/box စစ်ရုံ (လွယ်, polynomial)
```

NP ထဲမှာ — P ပြဿနာ အကုန် ပါတယ် (မြန်မြန် ဖြေနိုင်ရင် — မြန်မြန် စစ်နိုင်တာ သေချာ)။ ဒါပေမယ့် **"ဖြေဖို့လည်း လွယ်ရဲ့လား"** (P = NP လား) ဆိုတာ — computer science ရဲ့ **အဖြေမရှိသေးတဲ့ အကြီးမားဆုံး မေးခွန်း**ပါ ($1 သန်း ဆုကြေး ရှိ)။ အများစုက "P ≠ NP" (ဖြေဖို့ တကယ် ခက်တယ်) လို့ ယုံကြတယ်။

NP-Hard — အနည်းဆုံး NP လောက် ခက်

NP-Hard = **NP ထဲက ပြဿနာ အကုန်လုံး လောက် (အနည်းဆုံး) ခက်တဲ့** ပြဿနာတွေ။ ဒါတွေက — အဖြေ စစ်ဖို့တောင် polynomial time နဲ့ မရနိုင်တာ ပါနိုင်တယ် (NP ထဲ မပါတာ ပါ ပါနိုင်)။

ဥပမာ — **Traveling Salesman (TSP) ရဲ့ optimization version** ("အတိုဆုံး လမ်းကြောင်း ရှာ") က NP-Hard ပါ — ပေးထားတဲ့ လမ်းကြောင်း တစ်ခုက "အတိုဆုံး ဟုတ်မဟုတ်" စစ်ဖို့တောင် — ကျန်တဲ့ လမ်းကြောင်း အကုန်နဲ့ နှိုင်းရလို့ ခက်တယ်။

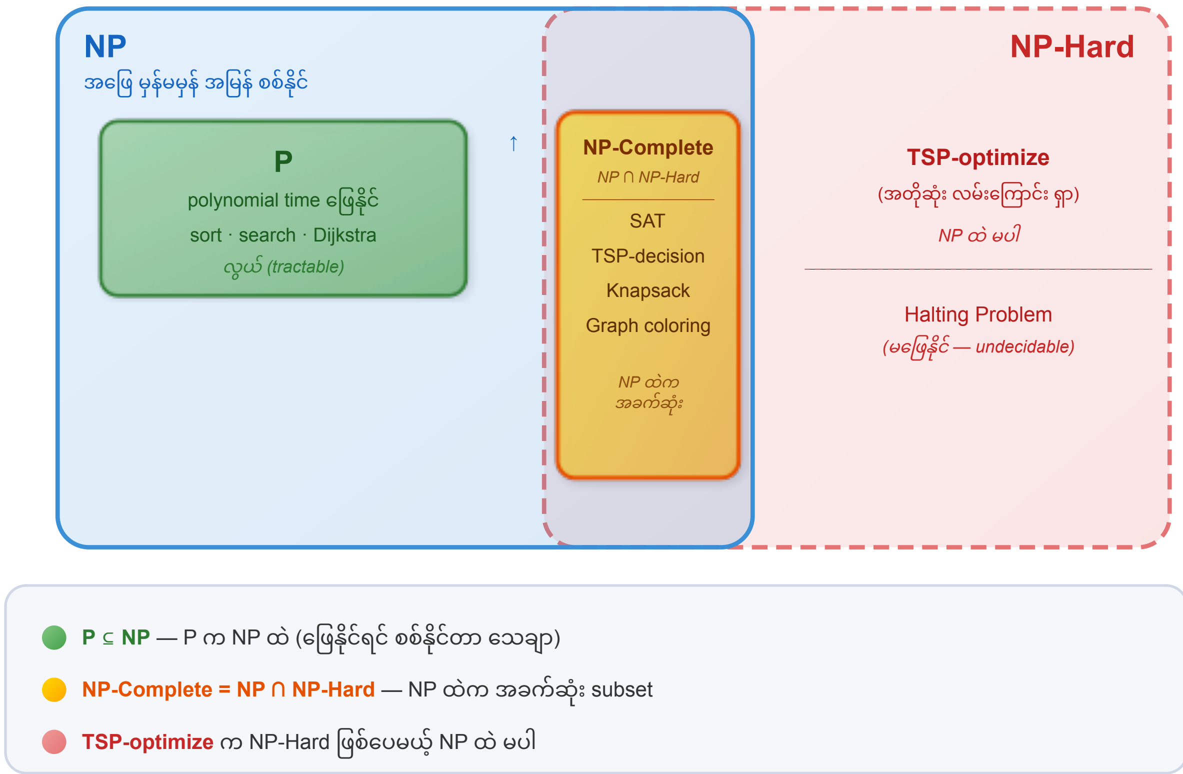
NP-Complete — အခက်ဆုံး NP ပြဿနာများ

NP-Complete = **NP ထဲမှာ ပါပြီး၊ NP-Hard လည်း ဖြစ်တဲ့** ပြဿနာတွေ — ဆိုလိုတာ "NP ထဲက အခက်ဆုံး" ပြဿနာတွေ —

```
NP-Complete = NP  ∩  NP-Hard
              (NP ထဲ ပါ)   (NP လောက် ခက်)
```

အရေးကြီးတဲ့ ဂုဏ်သတ္တိ — **NP-Complete ပြဿနာ တစ်ခုကို polynomial time နဲ့ ဖြေနိုင်ရင် — NP ပြဿနာ အကုန်လုံးကို ဖြေနိုင်**မယ် (P = NP ဖြစ်သွားမယ်)။ ဒါပေမယ့် အခုထိ ဘယ်သူမှ မရှာနိုင်သေး။ ဥပမာ NP-Complete ပြဿနာတွေ — Sudoku (general)၊ Boolean satisfiability (SAT)၊ Knapsack (decision)၊ Graph coloring။

ဆက်စပ်ပုံ Diagram



- **P ⊆ NP** — P က NP ထဲ (ဖြေနိုင်ရင် စစ်နိုင်တာ သေချာ)
- **NP-Complete = NP ∩ NP-Hard** — NP ထဲက အခက်ဆုံး subset
- **TSP-optimize** က NP-Hard ဖြစ်ပေမယ့် NP ထဲ မပါ

ခက်တဲ့ ပြဿနာ တွေ့ရင် — ဘာလုပ်မလဲ

NP-Hard / NP-Complete ပြဿနာ တွေ့ရင် — "ပြီးပြည့်စုံတဲ့ အဖြေ" ကို အချိန်ကုန်ခံ ရှာနေမယ့်အစား — developer တွေက လက်တွေ့နည်း ၂ မျိုး သုံးတယ်။

Approximation (ခန့်မှန်း အဖြေ)

"အကောင်းဆုံး မဟုတ်ပေမယ့် — အကောင်းဆုံးနဲ့ **နီးစပ်တဲ့** အဖြေ"ကို polynomial time နဲ့ ရှာတာ။ ဥပမာ — TSP အတွက် "အတိုဆုံးထက် အများဆုံး ၂ ဆ မပိုတဲ့ လမ်းကြောင်း" ကို အမြန် ရှာပေးတဲ့ approximation algorithm ရှိတယ်။

Heuristic (အတွေ့အကြုံ နည်းလမ်း)

"အမြဲ မှန်တယ်လို့ အာမ မခံပေမယ့် — လက်တွေ့မှာ ကောင်းတဲ့ အဖြေ ပေးလေ့ရှိတဲ့" rule of thumb။ ဥပမာ — TSP အတွက် **greedy heuristic**: "အခု ရှိတဲ့ မြို့ကနေ — အနီးဆုံး မြို့ကို အရင်သွား" (nearest neighbor)။ အမြဲ အကောင်းဆုံး မဟုတ်ပေမယ့် — အလွန် မြန်ပြီး၊ လက်တွေ့ လုံလောက်တဲ့ အဖြေ ပေးတယ်။

```
// TSP — Nearest Neighbor heuristic (အကောင်းဆုံး မဟုတ်၊ ဒါပေမယ့် မြန် + good enough)
int nearestNeighborTour(int[][] dist, int start) {
    int n = dist.length;
    boolean[] visited = new boolean[n];
    int cur = start, total = 0;
    visited[start] = true;
    for (int step = 1; step < n; step++) {
        int next = -1, best = Integer.MAX_VALUE;
        for (int j = 0; j < n; j++) {          // အနီးဆုံး မ visit ရသေးတဲ့ မြို့
            if (!visited[j] && dist[cur][j] < best) {
                best = dist[cur][j];
                next = j;
            }
        }
        visited[next] = true;
        total += best;
        cur = next;
    }
    return total + dist[cur][start];           // start ဆီ ပြန်
}
```

> ဒီ heuristic က $O(n^2)$ ပဲ — `n=25` အတွက်တောင် ချက်ချင်း ပြီးတယ် (brute force က စကြဝဠာ သက်တမ်း)။ အဖြေက "အတိုဆုံး" မဟုတ်ပေမယ့် — လက်တွေ့ delivery app တွေမှာ ဒီလို heuristic + optimization တွေ ပေါင်းသုံးကြတာပါ။

Real-world Examples

ဒီ "ခက်တဲ့" ပြဿနာတွေက — academic theory မဟုတ်ဘဲ — developer တွေ နေ့စဉ် တွေ့ရတဲ့ ပြဿနာတွေပါ —

- **Scheduling** — ဝန်ထမ်း shift, meeting room, CPU job ကို အကောင်းဆုံး စီစဉ်ဖို့ — constraint များလာရင် NP-Hard (လက်တွေ့မှာ heuristic သုံး)။
- **Route Optimization** — delivery, logistics ("မြို့ အကုန် လည်ပြီး အတိုဆုံး") — TSP — NP-Hard (Google Maps, ride-share က approximation သုံး)။
- **Resource Allocation** — server / cloud capacity ကို request တွေဆီ အကောင်းဆုံး ခွဲဝေ — Bin Packing / Knapsack — NP-Hard။
- **Traveling Salesman Problem (TSP)** — အထက်ပါ အားလုံးရဲ့ "ပုံစံ" — NP-Hard ရဲ့ classic ဥပမာ။

Practical Message — Developer အတွက် သင်ခန်းစာ

ဒီနောက်ဆက်တွဲရဲ့ အဓိက ရည်ရွယ်ချက်က — proof တွေ ကျက်ဖို့ မဟုတ်ဘဲ — **developer အနေနဲ့ ဆုံးဖြတ်ချက် ချနိုင်ဖို့**ပါ —

1. **ပြဿနာ တိုင်းမှာ ပြီးပြည့်စုံ မြန်တဲ့ အဖြေ မရှိ** — တချို့ ပြဿနာတွေက အခြေခံကတည်းက ခက်တယ် (NP-Hard)။ ကိုယ့်ဘက်က code မညံ့လို့ မဟုတ်။
2. **"ခက်တဲ့ ပြဿနာ" ဆိုတာ မှတ်မိဖို့** — ပြဿနာတစ်ခု TSP / scheduling / packing ပုံစံ ဆိုရင် — "perfect solution" ရှာနေမယ့်အစား — approximation / heuristic ဆီ ချက်ချင်း ကူးတာ ပိုသင့်တယ်။
3. **"Good enough" ကို လက်ခံတတ်ဖို့** — လက်တွေ့ business မှာ — "၉၅% အကောင်းတဲ့ အဖြေ ၁ စက္ကန့်" က "၁၀၀% အဖြေ ၁ နှစ်နဲ့" ထက် အများကြီး တန်ဖိုးရှိတယ်။
4. **ဘယ်တော့ optimize ရပ်ရမလဲ သိဖို့** — over-engineering မလုပ်ဘဲ — ပြဿနာရဲ့ သဘာဝ ခက်ခဲမှုကို နားလည်ပြီး — အချိန်/အရင်းအမြစ်ကို သင့်တော်စွာ သုံးတတ်ဖို့။