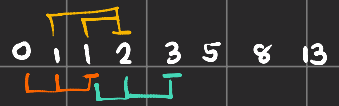
Recursion - when function calls itself

↳ jub bigger problem ka solution depend karta ho choti and same type ki problem par

→ fibonacci series



n = (n-1) + (n-2)

f(n) → print counting from n to 1
f(5) → print counting from 5 to 1

f(5) → print (5) + f(n)

badi problem — choti problem

Badi Problem → 5!

5 x 4!

choti

→ 2^5 = (!)

Badi Problem → 2^5

2×2^4

choti problem

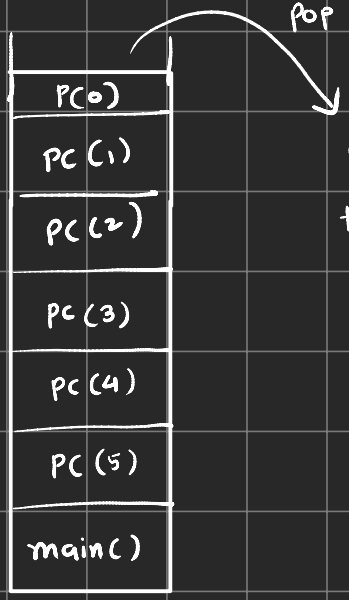
→ Recursive code

- ↳ Base condition (when stop code)
- ↳ recursive relation / recursive call [f(n) = n * f(n-1)]
- ↳ processing (optional)

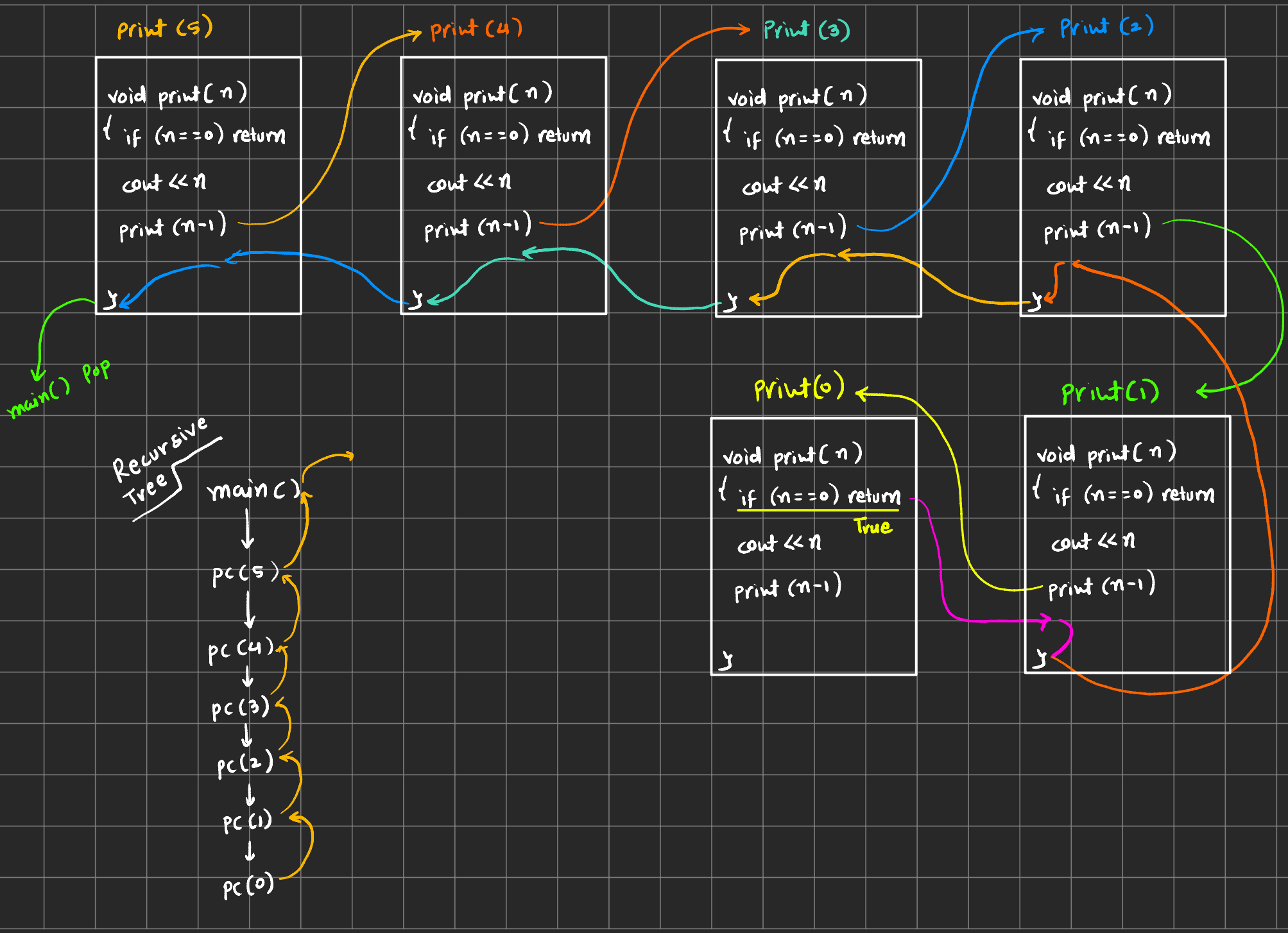
printCounting

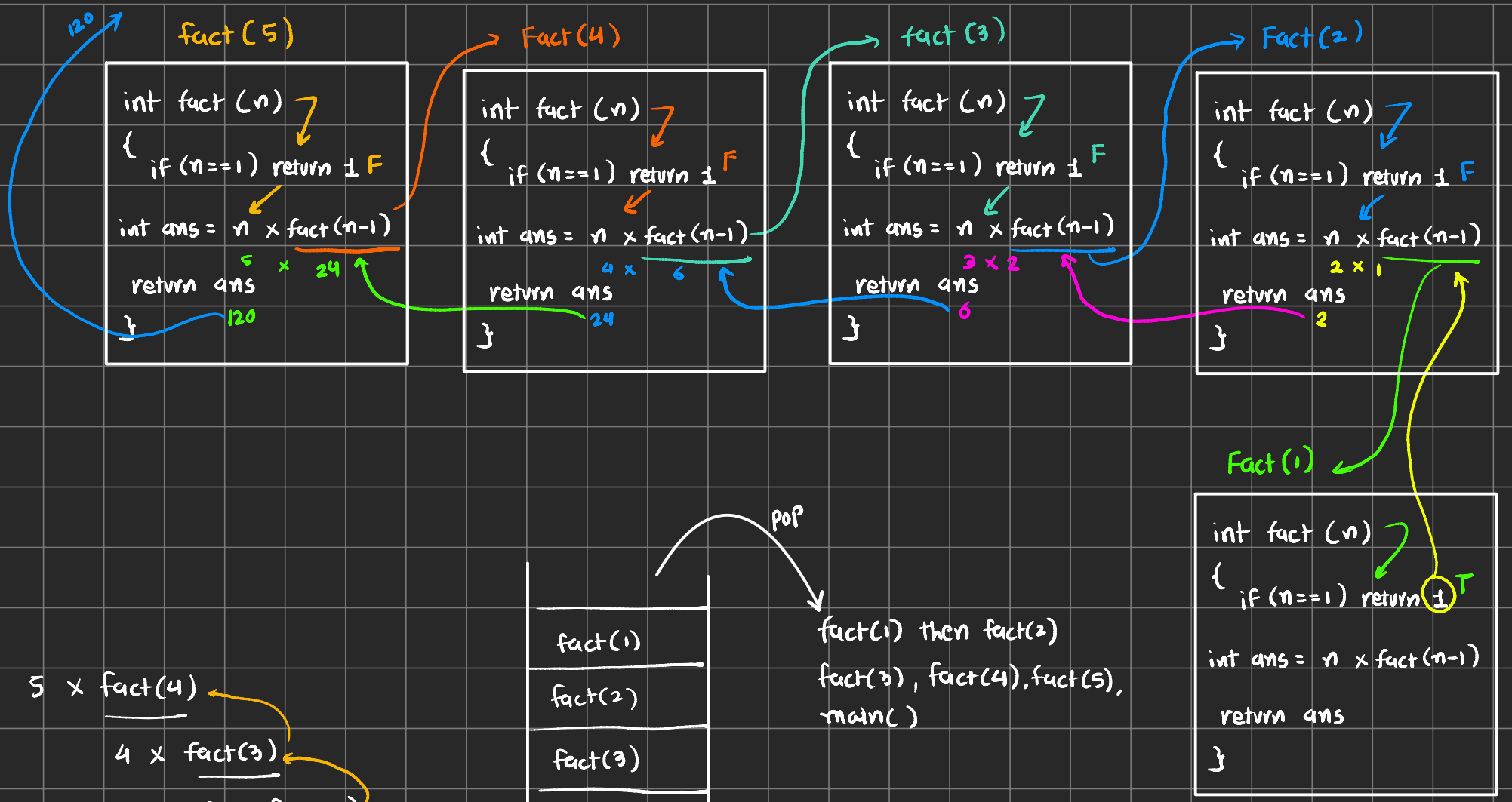
```
void counting(int n){

    // base condition
    if(n==0){
        return ;
    }

    // processing
    cout << "function is called for "<< n << endl;

    // recursive relation
    counting(n-1);
}
```

→ come here after pop from stack



PC(0) then PC(1) then PC(2) then PC(3) then PC(4) then PC(5) then main()





```
void solve ()
{
  // Base case
  // processing
  // Recursive  → Tail recursion
}
```

```
void solve()
{
  // Base case
  // recursive  → Head recursion
  // processing
}
```

```
Void print (n)
{
  if (n==0) return
  cout << n
  print (n-1)  → Tail Recursion
}
```

output → 5 4 3 2 1

```
void print (n)
{
  if (n==0) return
  print (n-1)  → Head Recur
  cout << n
}
```

output → 1 2 3 4 5

why?

Print (5)

```
void print (n)
{
  if (n==0) return
  print (n-1)
  cout << n   5
}
```

Print (4)

```
void print (n)
{
  if (n==0) return
  print (n-1)
  cout << n   4
}
```

Print (3)

```
void print (n)
{
  if (n==0) return
  print (n-1)
  cout << n   3
}
```

Print (2)

```
void print (n)
{
  if (n==0) return
  print (n-1)
  cout << n   2
}
```

Print (0)

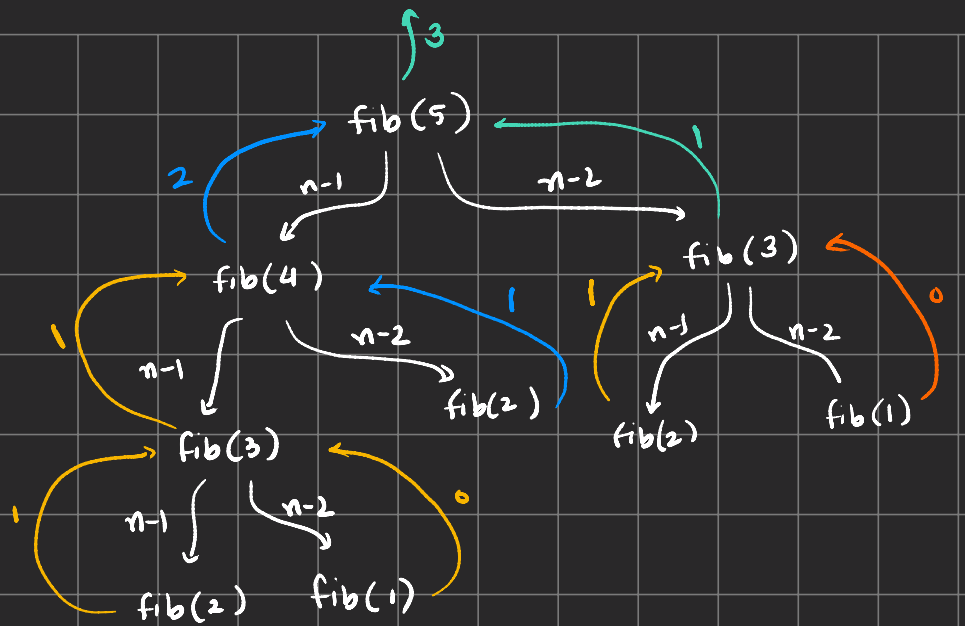
```
void print (n)
{
  if (n==0) return TRUE
  print (n-1)
  cout << n
}
```

Print (1)

```
void print (n)
{
  if (n==0) return
  print (n-1)
  cout << n   1
}
```

Output
1, 2, 3, 4 5

```
// 0       1       1      2       3      4
// first  second  thrid  fourth  fifth  sixth

int fibo(int n){

    // base case
    if(n==1) return 0;  // first term
    if(n==2) return 1;  // second term

    // recursive rel [ f(n) = f(n-1)+f(n-2) ]
    int ans = fibo(n-1) + fibo(n-2) ;

    return ans;
}
```



fib(4)

```
int fibo(int n) {
    if (n == 1) return 0;
    if (n == 2) return 1;
    int ans = fibo(n - 1) + fibo(n - 2);
    return ans;
}
```

fib(3)

```
int fibo(int n) {
    if (n == 1) return 0;
    if (n == 2) return 1;
    int ans = fibo(n - 1) + fibo(n - 2);
    return ans;
}
```

fib(2)

```
int fibo(int n) {
    if (n == 1) return 0;
    if (n == 2) return 1; True
    int ans = fibo(n - 1) + fibo(n - 2);
    return ans;
}
```

fib(1)

```
int fibo(int n) {
    if (n == 1) return 0; True
    if (n == 2) return 1;
    int ans = fibo(n - 1) + fibo(n - 2);
    return ans;
}
```

fib(2)

```
int fibo(int n) {
    if (n == 1) return 0;
    if (n == 2) return 1; True
    int ans = fibo(n - 1) + fibo(n - 2);
    return ans;
}
```

Magical Line - ek case solve kar do baki recursion sambhal lega

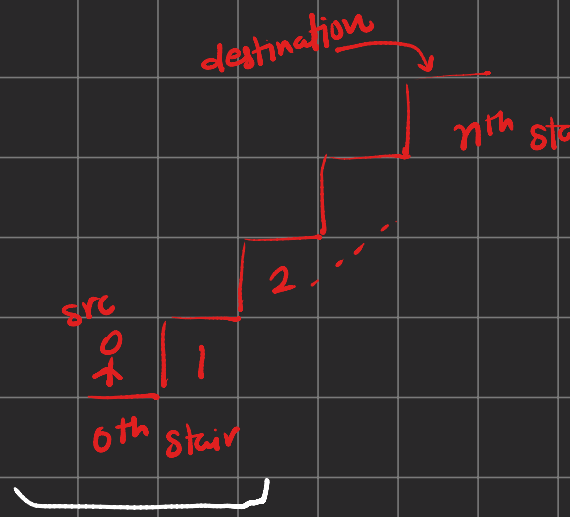
$5! = 5 \times 4!$ → solve this case

Badi problem

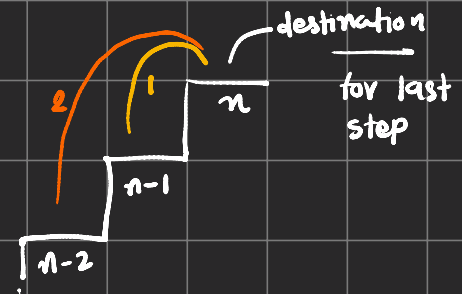
choti problem

ye recursion sambhal lega

* steps allowed
  - 1 stairs at a time
  - 2 stairs at a time



To reach to the destination we have two types



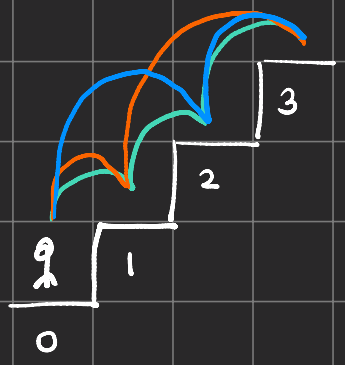
Base case

0th stair par jane k tarike = 1 (wo he stair par kudna)

1th stair par jane k tarike = 1

$f(n) = f(n-1) + f(n-2)$

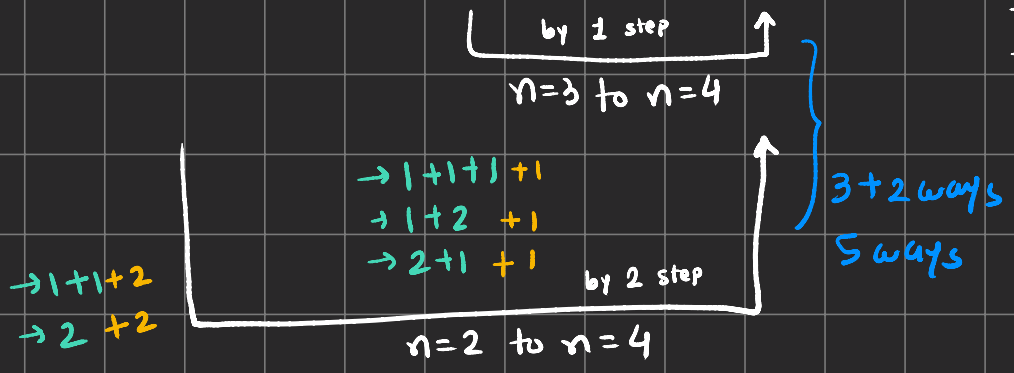
To reach to stair num 3



3 ways → 1-1-1
1-2
2-1

	n=1	n=2	n=3	n=4	n=5
No of ways	1	2	3	5	8
Which ways?	→ 1	→ 1+1 → 2	→ 1+1+1 → 1+2 → 2+1	→ 1+1+1+1 → 1+1+2 → 1+2+1 → 2+1+1 → 2+2	→ 1+1+1+1+1 → 1+1+1+2 → 1+1+2+1 → 1+2+1+1 → 2+1+1+1 → 1+2+2 → 2+2+1 → 2+1+2

Base case
n=0 → 1 step
n=1 → 1 step



✱ Print array

arr [1,2,3,4,5] n=5

0 1 2 3 4

i=0

```
printArr (arr,n,i)
{
    if (i>=n) return
    cout << arr[i]
    printArr (arr, n, i+1)
}
```

```
printArr (int arr[], n)
{
    if (n==0) return
    cout << arr[0] << *arr
    print (arr+1, n-1)
}
```

✱ find max ele from arr

```
int findMaximumEle(int arr[], int n, int i, int& maxi){

    // base case
    if(i>=n){
        return maxi;
    }

    // progress
    if(maxi<arr[i]){
        maxi = arr[i];
    }

    // recursion
    findMaximumEle(arr,n,i+1, maxi);

}
```

✱ key is present in char array or not

string str = "lovebabbar" , key = 'r' → is r present in str or not

```
int findKey(string& str, int& n, char& key, int  i) {
    if (i >= n) {
        return -1;
    }

    if (str[i] == key) {
        cout << "found at index =>" << i << endl;
        return i;
    }

    int ans = findKey(str, n, key, i + 1);
    return ans;
}
```

✱ Print all digits of number ip → 647

647 /·10 = 7

↓

647/10 = 64

64 /·10 = 4

↓

64/10 = 6 /·10 = 6

↓

6/10 = 0 → stop

```
void printDigit(int num){

    if( num == 0 ){
        return;
    }

    // int digit = num%10;
    // cout << digit << " ";       // that gives 5 4 6

    int newnum = num/10;
    printDigit(newnum);

    int digit = num%10;
    cout << digit << " ";          // niche ghusa do ulata kar dega

}
```

ip → array → sorted in ascending order or not

Binary Search

```cpp
int binarySearch(vector<int>& v, int s, int e, int key){

    // base case
    if( s>e ){
        return -1;  // not found
    }
    int mid = (s+e)/2;
    if( v[mid] == key ){
        return mid; // got key
    }


    if( v[mid] < key ){
        binarySearch(v, mid+1, e, key); // right search
    }
    else{
        binarySearch(v, s, mid-1, key);
    }
}
```

→ Avoid to pass by value,

Pass by value - 250 MB - 90 ms

Pass by referance - 6 MB - 20 ms

return concept

```cpp
void solve ( )
{

    return ;

}
```

```cpp
int solve ( )
{

    return int value;

}
```

func call

Subsets / Subsequences of String

ip : "abc" op - print all subsequences

n = 3

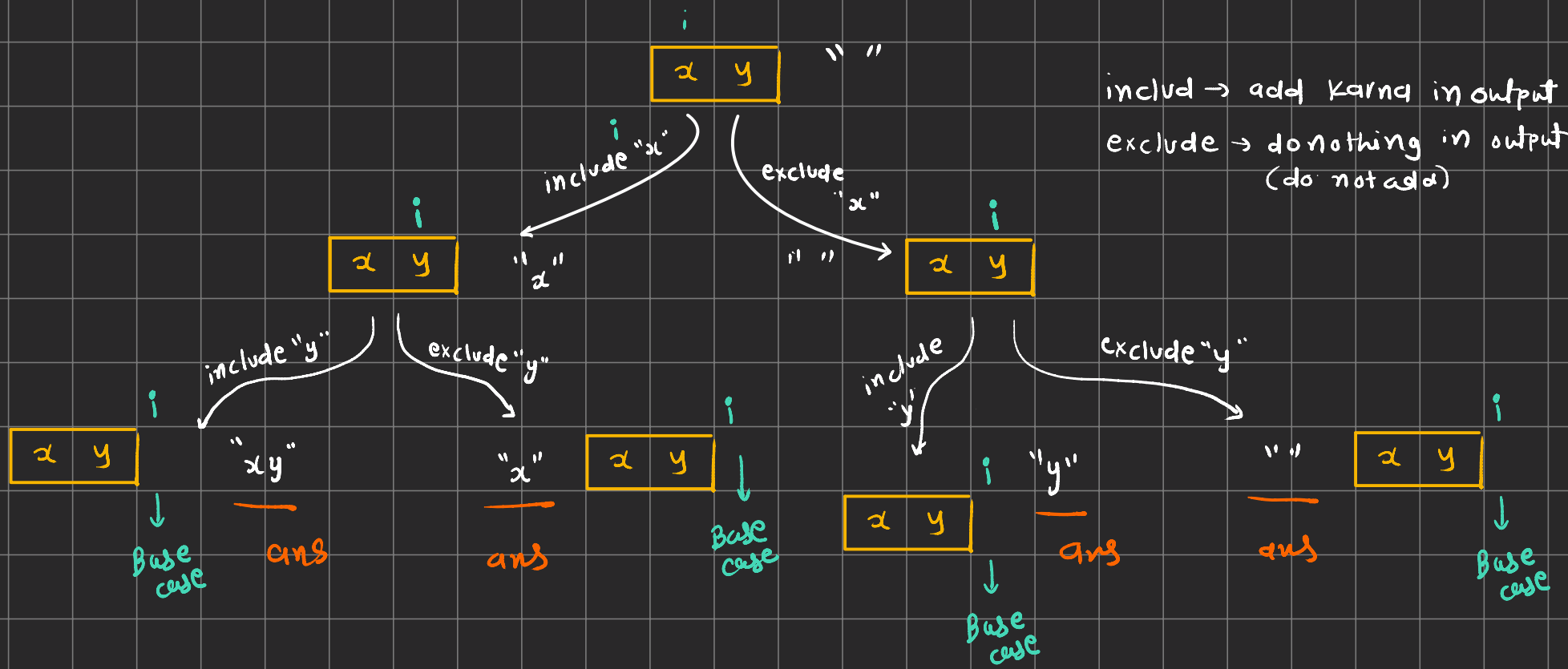
abc	
✓ ✓ ✓	- abc
✓ ✓ ✗	- ab
✓ ✗ ✗	- a
✗ ✗ ✗	-
✗ ✓ ✗	- b
✗ ✗ ✓	- c
✗ ✓ ✓	- bc
✓ ✗ ✓	- ac

include exclude pattern

pick - not pick pattern

no of chars	n = 1	n = 2	n = 3
subseqs	2	4	8

```cpp
void printSubstring(string str, string output, int i){

    //base case
    if(i>= str.length()){
        cout << output << endl;
        return;
    }

    // rr
    // 1)exclude
    printSubstring(str, output, i+1);

    // 2)include
    output.push_back(str[i]);      ⟶ output + "~"
    printSubstring(str, output, i+1);

}
```



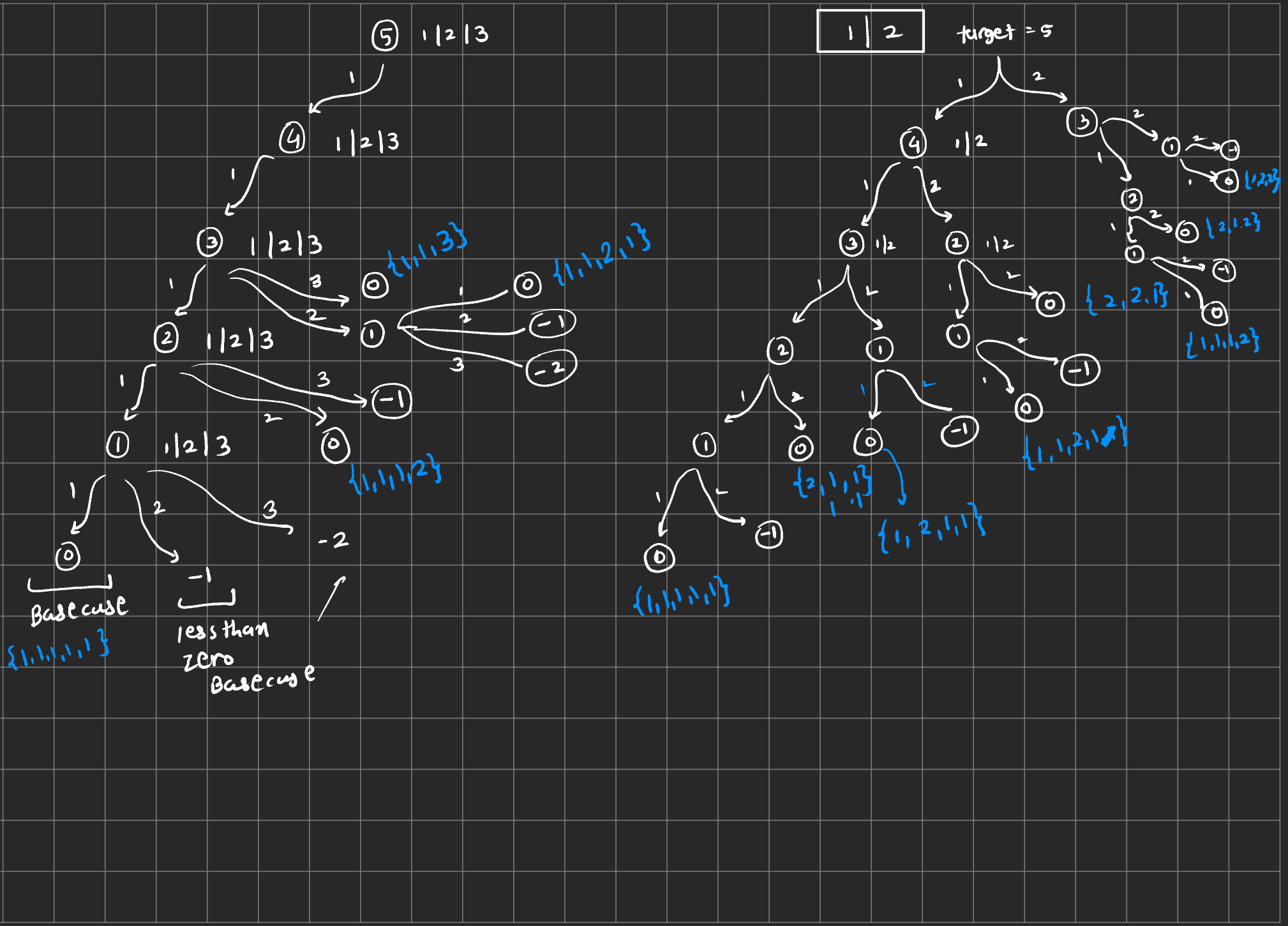
* You have to tell minimum number of elements required to reach target sum [COIN CHANGE]

IP → | 1 | 2 | 3 | → infinite supply of this all elements

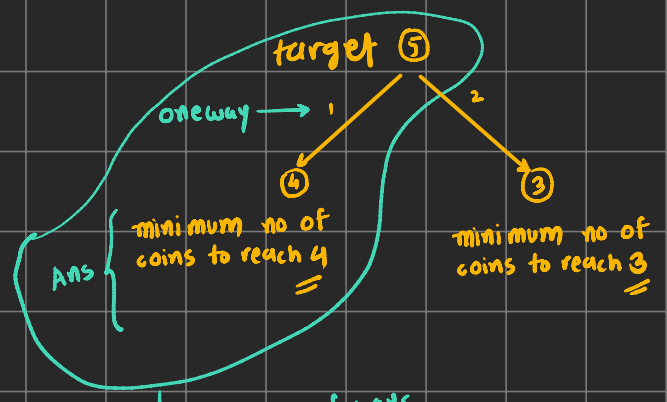
targetSum = 5

5 → {1,1,1,1,1}
{1,1,1,1,2}
{2,2,1}
{3,1,1}
{3,2} → min (so minimum 2 elements are required to reach to ans)
{1,2,1,1}

Imp pattern
for loop in recursion

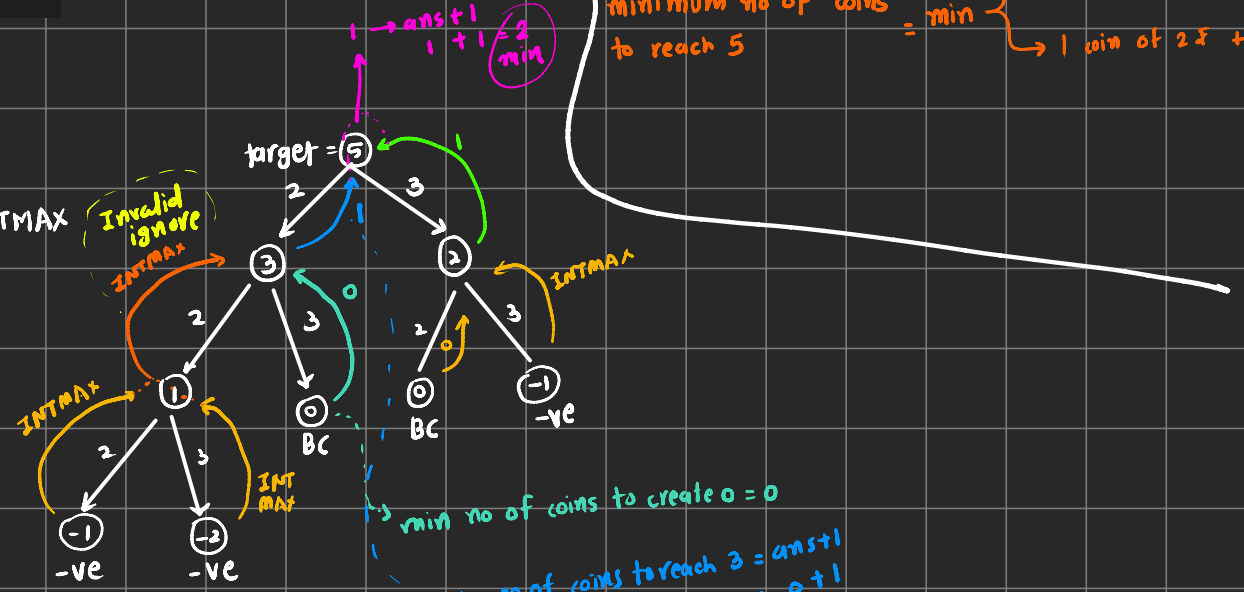


```cpp
int coinChange(vector<int>& arr, int target){

    if( target == 0 ){
        return 0;
    }
    if(target<0){
        return INT_MAX;
    }

    int mini = INT_MAX;
    for(int i=0; i<arr.size(); i++){
        int ans = coinChange(arr, target-arr[i]);
        if(ans!=INT_MAX){
            mini = min(mini, ans+1);
        }
    }
    return mini;

}
```



min no of ways to reach to the target is
= ans + one way
= ans + 1

minimum no of coins to reach 5 = min { 1 coin of 1 ₹ + min no of ways to reach 4 ₹s ; 1 coin of 2 ₹ + min no of ways to reach 3 ₹s }



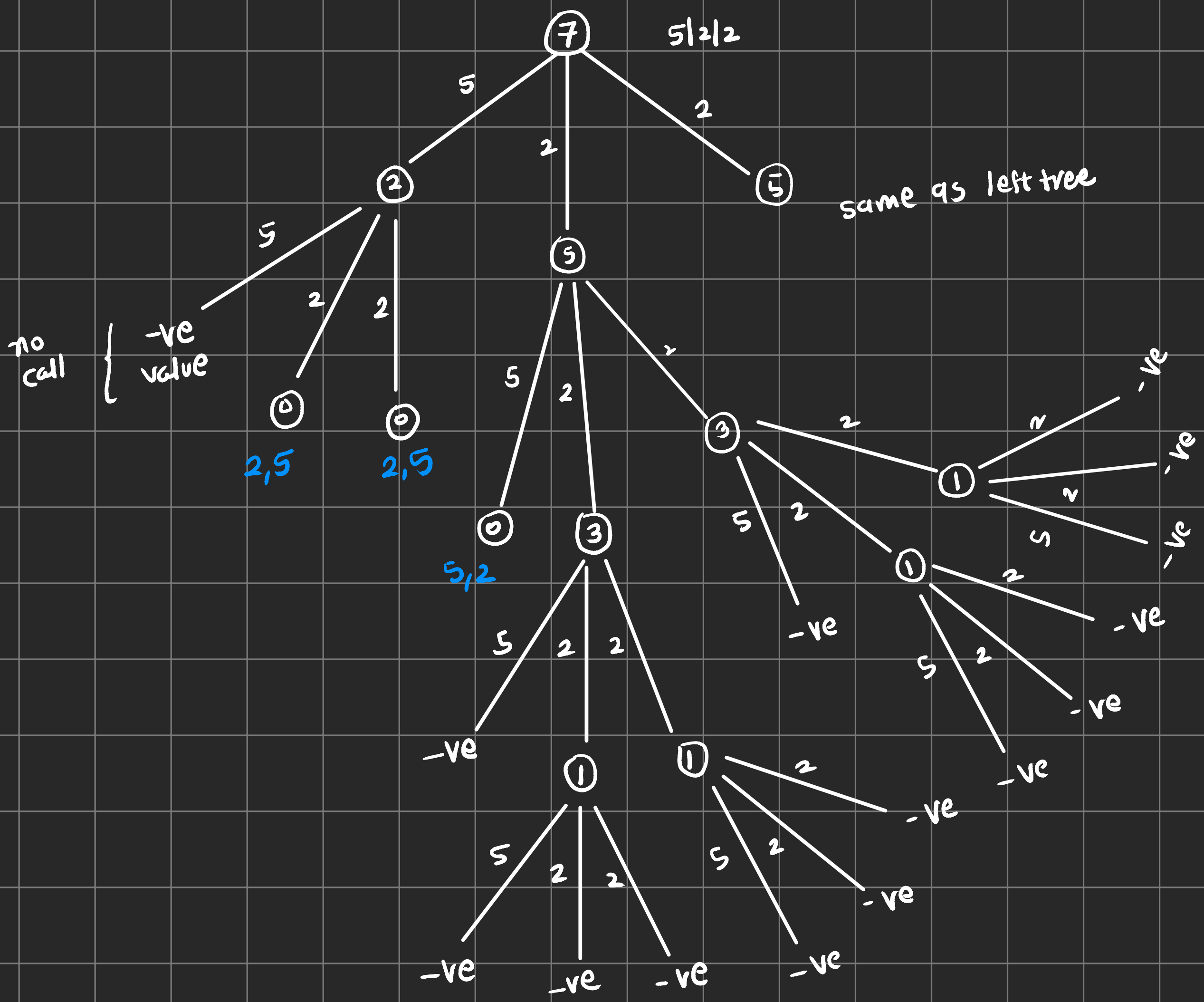
if (target = 0) → return 0

if (target = -ve) → return INTMAX

∴ min no of coins to create 0 = 0

∴ min no of coins to reach 3 = ans+1
= 0 + 1
= 1

★ cut into segments   ip → N → rod length 7   find maximum no of segment to cut rod in parts like x = 5, y = 2, z = 2, N = 7




```cpp
int cutIntoSegment(int n, int x, int y, int z){

    // base case
    if( n == 0 ){
        return 0;
    }
    if( n<0 ){
        return INT_MIN;
    }

    int a = cutIntoSegment(n-x, x,y,z) + 1;
    int b = cutIntoSegment(n-y, x,y,z) + 1;
    int c = cutIntoSegment(n-z, x,y,z) + 1;

    int ans = max( a, max(b,c) );
    return ans;

}
```

```cpp
int ans = cutIntoSegment(n, x,y,z);
if(ans<0){
    cout << "ans is" << 0 << endl;
}
cout << "ans is" << ans << endl;
```

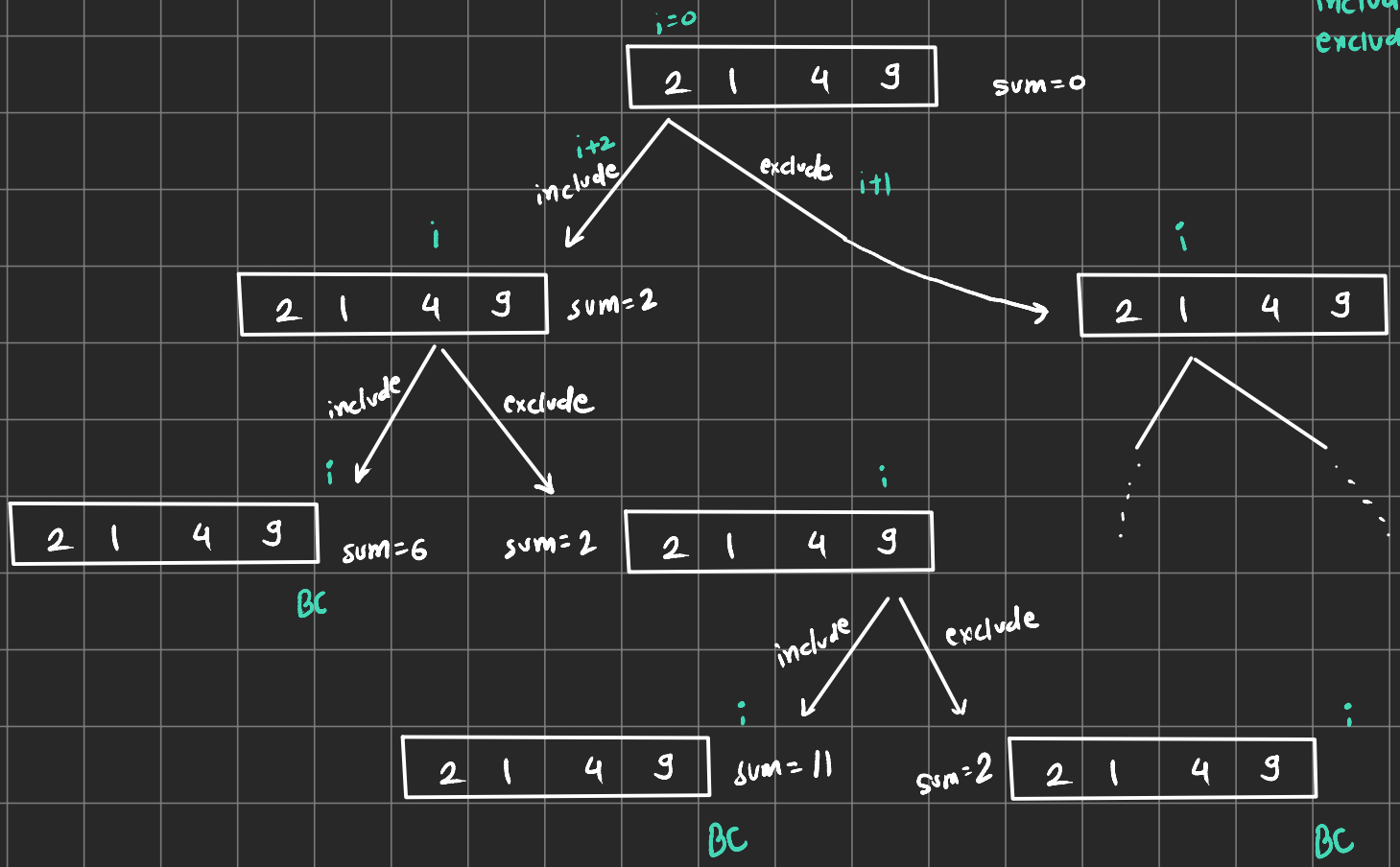
cout = 1
7 max → rod 5 + max no of segment for length 2
→ rod 2 + " " " " length 5
cout = 1

max of Both { ansA = funct ( ) + 1
ansB = funct ( ) + 1

Created by Notein

Maximum sum of non adjacent element – return the maximum sum of subsequence in which no two elements are adjacent

include exclude pattern



maxi = INT_MIN

```cpp
void maxSumOfNonAdjacentEle(vector<int>& arr, int sum, int& maxi, int i){

    //base
    if(i>=arr.size()){
        maxi = max(sum, maxi);
        return;
    }

    //include
    maxSumOfNonAdjacentEle(arr, sum+arr[i], maxi, i+2);

    //exclude
    maxSumOfNonAdjacentEle(arr, sum, maxi, i+1);

}
```

✱ Last occurance of a char

string → " abcddedg "
0 1 2 3 4 5 6 7

char x = 'd' ans → 6 index

① search from left to right (iterative method) ② search from right to left (iterative method)

③ STL function strrchr()

④ Base case → if (i >= string.length()) return

→ if (s[i] == target) → ans = i

solve (s, target, ans, i+1)

✱ Time and Space Complexity of Recursive Solution

```
void printArray(int a[], int n){
  if(n == 0) return;
  cout<<*a<<" ";
  printArray(a+1, n-1);
}
```

→ K time taken operation

$F(n) = K + F(n-1)$ → two operations so K time

$F(n-1) = K + F(n-2)$

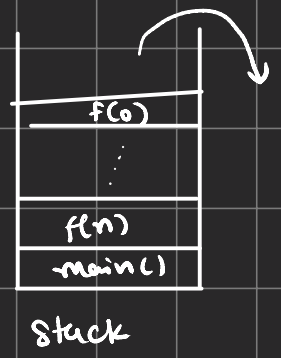
$F(n-2) = K + F(n-3)$

$F(1) = K + F(0)$

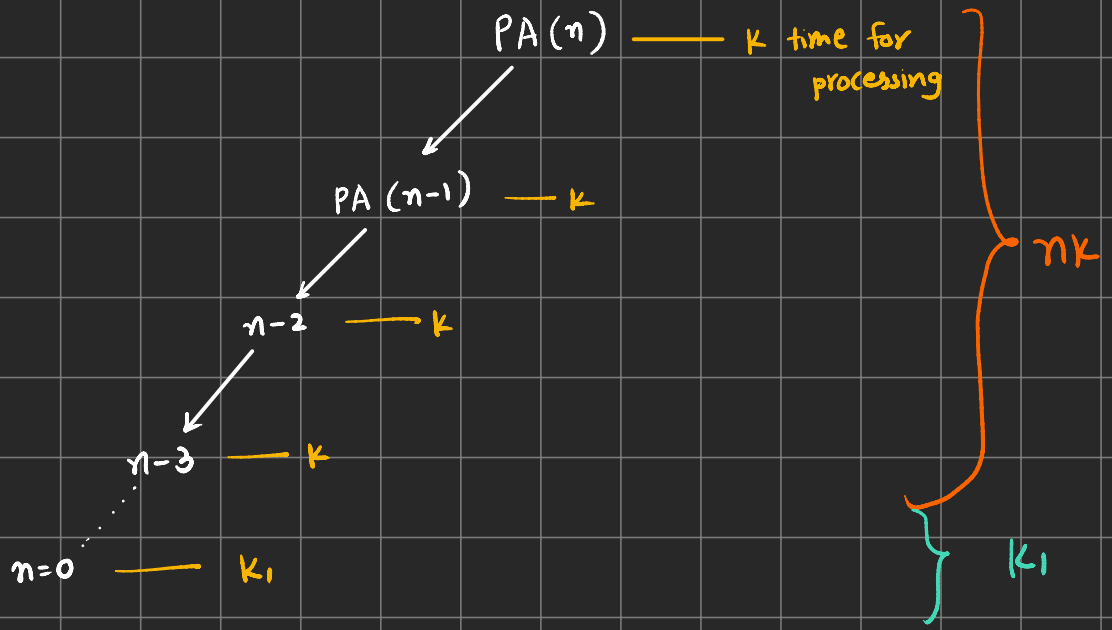
$F(0) = K_1$ Base case → return → only one operation so K_1 time

$F(n) = nK + K_1$ → K_1 is constant so ignore → $O(n(K))$ → K is constant so ignore → $O(n)$

TC



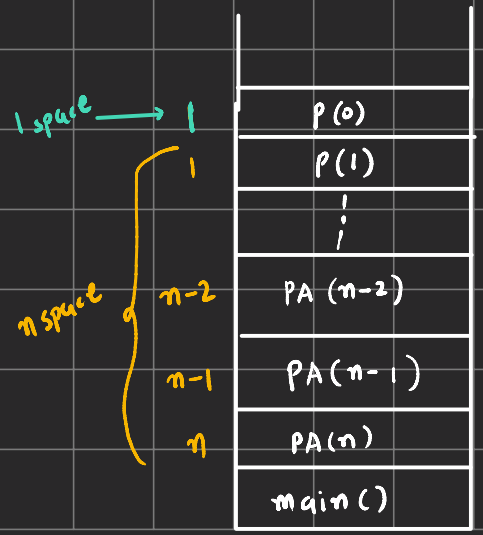
→ Recursive tree method



$F(n) = nk + k_1$

$O(n)$ → TC

→ Space complexity



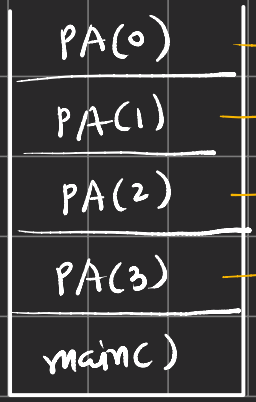
$O(n\,space + 1\,space)$

$O(n+1)$

$O(n)$

SC

if $n = 3$



$O(4)$

$O(n+1)$

$O(3+1)$

$O(3)$ → SC

→ Factorial

```c
int fact(int n)
{
    if (n == 1)
        return 1;

    return n * fact(n - 1);
}
```

$F(n) = n * F(n-1)$

$T(n) = K * T(n-1)$ → equation

$T(n) = K + T(n-1)$

$T(n-1) = K + T(n-2)$

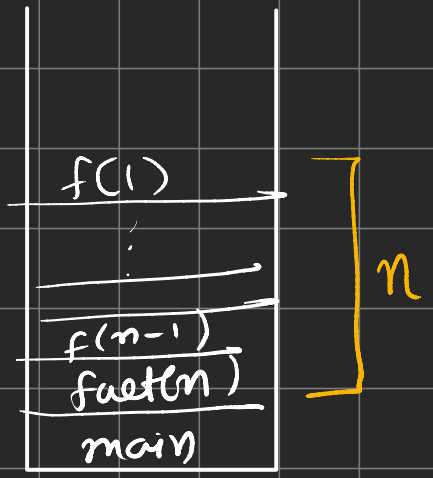
⋮

$T(1) = K$

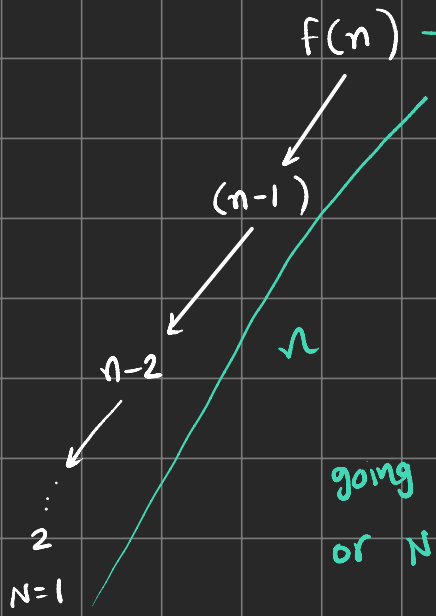
$T(n) = nK$

$O(T_n) = O(nk)$

$= O(n)$ TC



SC, O(n)



$T_n = nK$

$O(n)$

going from 1 → N

or N → 1

→ Binary Search

```
int BS(int a[], int k, int start, int end)
{
    if (start > end)
        return -1;

    int mid = start + (end - start) / 2;
    if (a[mid] == k)
        return mid;
    else if (k > a[mid])
    {
        return bsr(a, k, mid + 1, end);
    }
    else
    {
        return bsr(a, k, start, mid - 1);
    }
}
```

$F(n) = k + F(n/2)$

$F(n/2) = k + F(n/4)$

$F(n/4) = k + F(n/8)$

⋮

$F(2) = k + F(1)$

$F(1) = k$

$F(n) = a * k$

$= O(ak)$

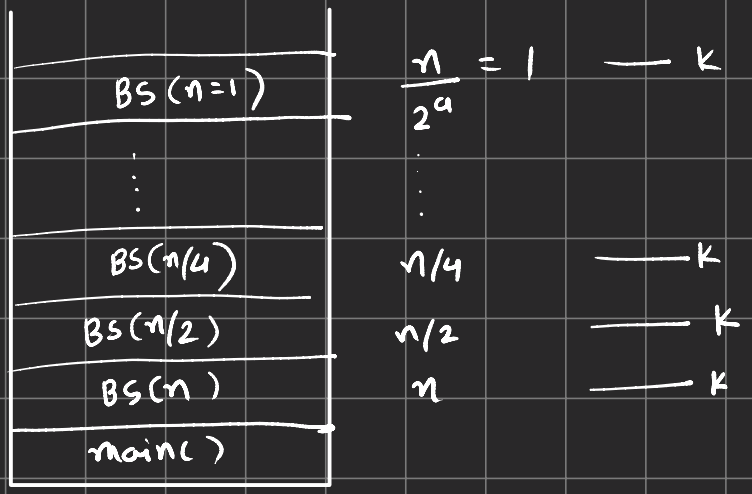
$= O(\log n \cdot k)$ k is const

$= O(\log n)$ Tc

n → n/2 → n/4 → n/8 ⋮ 1 } 2^a

$\frac{n}{2^a} = 1$

$a = \log n$



$a = \log n$

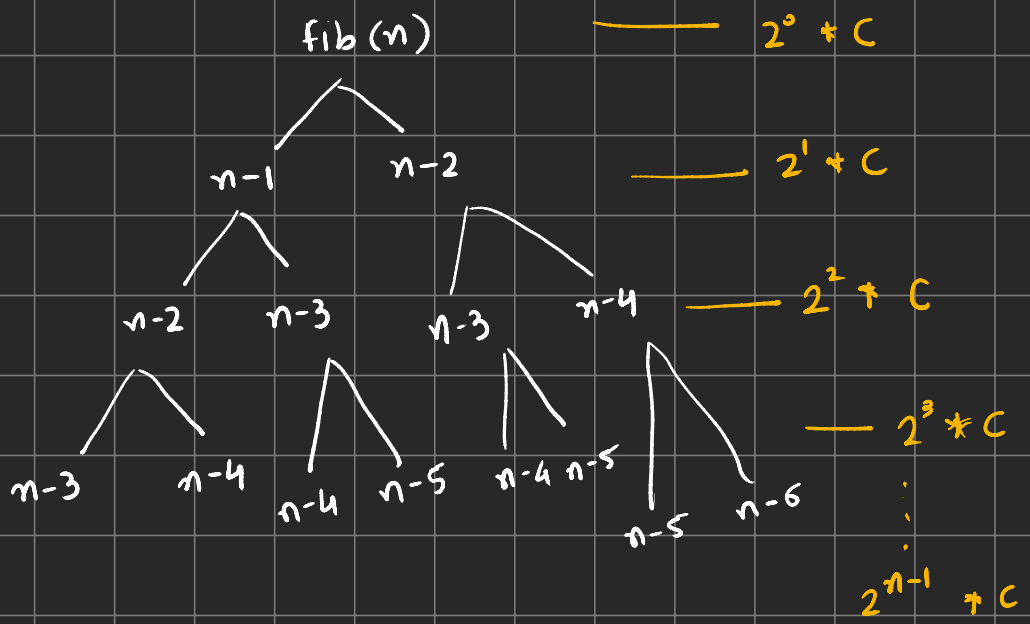
$O(T_n) = O(k \log n)$

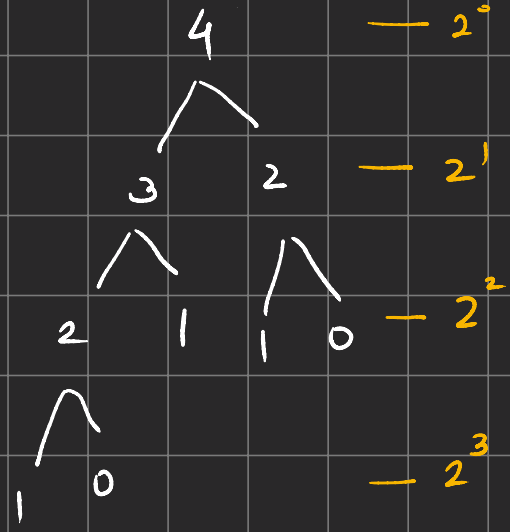
$O(n) = \log n$ Sc

simple iterative BS has not space complexity so this is the drawback of recursion it comes with some space complexity

→ fibonacci

```
int fib(int n){
  if(n == 0 || n == 1) return n;

  return fib(n - 1) + fib(n - 2);
}
```





SC → O(4)

O(n)

$T(n) \leq C\,[2^0 + 2^1 + 2^2 + \dots + 2^{n-1}]$

$T(n) \leq C\,[2^n - 1]$

$T(n) = O(2^n)$ → crazy wigh TC exponential that's why } Bekar way nai

$T(n) \leq 2^0 + 2^1 + 2^2 + 2^3$

✱ Reverse string recursively

s = "abcd" → swap in RE

a	b	c	d
↑ s			↑ e

swap (s,e)

Base case (s >= e)

TC = O(N/2) = O(N)

SC = (N/2 + 1) = O(N)

✱ Add Strings

```
  1   1
  4   5   6
      7   7
  ---------
  5   14  13   ~> 13 % 10 = 3
                  13 / 10 = 1 -> carry
Ans -> 5   4   3
```

```
string addString(string &s1, string &s2, int p1, int p2, int carry, string &ans){

    // base
    if( p1 < 0 && p2 < 0 ){
        if(carry!=0){
            ans.push_back(carry+'0');
            return string(1,carry+ '0');
        }
        return "";
    }

    int n1 = ( p1>=0 ? s1[p1] - '0' : '0');
    //int n2 = s2[p2] - '0';  // int convert
    int n2= ( p2>=0 ? s2[p2] - '0' : '0');

    int total = n1+n2+carry;

    int digit = total%10;
    carry = total/10;

    string rAns = "";

    ans.push_back(digit+'0');
    rAns.push_back(digit+'0');

    // rr
    rAns += addString(s1,s2,p1-1, p2-1, carry, ans);
    return rAns;
}
```

TC = p1=2, p2=2 → K

1,1 → K

0,0 → K

-1,-1 → Kp

N = 3

O(N+1)

O(N)

SC O[(N+1)+1]

O(N)

→ reverse Ans in end

#include <algorithm> reverse (ans.start(), ans.end())

✱ Palindrome check

r	a	c	e	c	a	r
s						e

TC as above

O(N)

SC as above

O(N)

Remove all occurances of subarray

S = d a a b c b a a b c b c (0 1 2 3 4 5 6 7 8 9 10 11) part = "abc"

2

→ ① find partstring position in s
② remove partstring from s
③ call again function for new string

```cpp
void removeOcc(string &s, const string &part) {
    int found = s.find(part);

    if (found != string::npos) {
        // Part string is found
        string leftPart = s.substr(0, found);
        string rightPart = s.substr(found + part.size());
        s = leftPart + rightPart;

        // s.erase(found, part.length());   // another way

        // Recursively call removeOcc to remove further occurrences
        removeOcc(s, part);
    }else{
        return;
    }
}
```

"abcabc" → s size = 6
part size = 3

2 calls here

$= \frac{6}{3} = \frac{N}{m} = 2$ calls

SC

total calls = O(N/m)

How many recursive calls in worst case?

s → N size
part → m size

total calls = for each part in s

$= \frac{N}{m}$ worst case

$= O\left(\frac{N}{m}\right) * O(N/m)$

$= O(n^2)$

TC : find() → O(N.m)

N → s.size
m → part.size

leftpart() → O(n)
rightpart() → O(n)
general → O(n)

$= O(N.m) + O(n) + O(n) + O(n)$

max

$= O(N.m)$ ⟶ TC of one case

Printing all subarrays

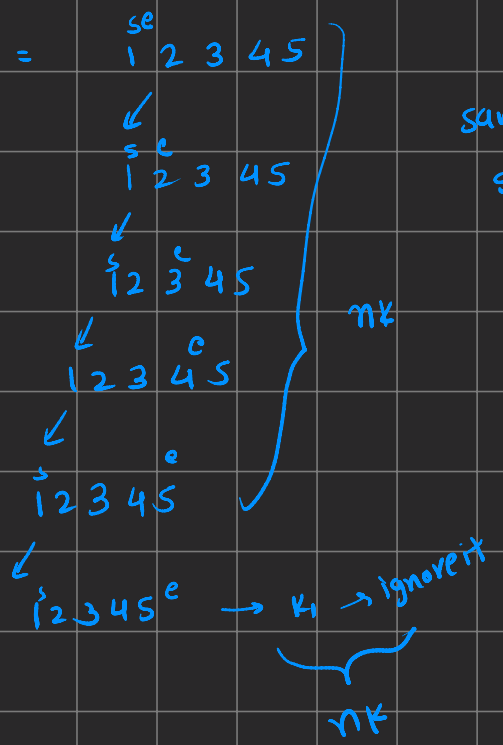
1	2	3	4	5

Subarrays

1 2 3 4 5
12 23 34 45
123 234 345
1234 2345
12345

```cpp
void printAllSubarrays(vector<int>& arr, int start, int end) {
    // Base case
    if (end == arr.size()) {
        return;
    }

    for (int i = start; i <= end; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    printAllSubarrays(arr, start, end + 1);
}

void forLoopFunc(vector<int>& arr) {
    // Loop over all possible starting points
    for (int start = 0; start < arr.size(); start++) {
        int end = start;
        printAllSubarrays(arr, start, end );
    }
}
```

TC =



same for

s → 0 → 5

= n × nk

= $n^2 k$

= $O(n^2)$

SC

O(n+2)

O(n)

✱ Buy and sell stocks

price	7	1	5	3	6	4
oth day	1	2	3	4	5	

Buy → stock price is lowest

sell → stock price is highest in future

7	1	5	3	6	4
0	1	2	3	4	5
minPrice=7 todayprofit=0 maxProfit=0	minP=1 tp=-6 maxP=0	minP=1 tp=4 maxP=4	minP=1 tp=2 maxP=4	minP=1 tp=5 maxP=5	minP=1 tp=4 maxP=5

```cpp
void maxProfitFinder(vector<int>& prices, int i, int& minPrice, int& maxProfit) {
    // Base case: if we've processed all days
    if (i == prices.size()) {
        return;
    }

    // Update minPrice if current price is lower
    if (prices[i] < minPrice) {
        minPrice = prices[i];
    }

    // Calculate today's profit and update maxProfit if it's higher
    int todayProfit = prices[i] - minPrice;
    if (todayProfit > maxProfit) {
        maxProfit = todayProfit;
    }

    // Recursive call for the next day
    maxProfitFinder(prices, i + 1, minPrice, maxProfit);
}
```

TC = O(n+1)

= O(n)

SC = O(n)

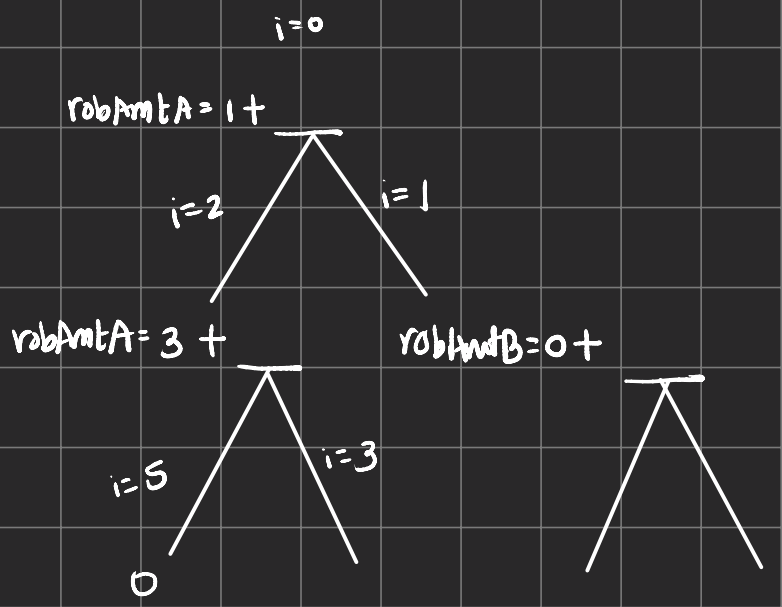
House Robber

1	2	3	1
0	1	2	3

include - exclude solution

- cannot rob adjacent house
- find max amount that you can rob

```
int robHelper(vector<int>& nums, int i){

    if( i>=nums.size() ){
        return 0;
    }

    int robAmt1 = nums[i] + robHelper(nums, i+2); // include
    int robAmt2 = 0 + robHelper(nums, i+1);  // exclude

    return max(robAmt1, robAmt2);
}
```



✱ INT to English words

1	2	3	4	5	6	7	→	1 million	- one million
								234 thousands	two hundred thirty four thousands
								567	Five hundred sixty seven

Hashes					
	1 - one	11 - eleven	20 - twenty	100 - Hundred	
	2 - two	12 - twelve	30 - thirty	1000 - thousands	
	3	13	40	10^6 - million	
	4	14	50	10^9 - billion	
	5	15	60		
	6	16	70		
	7	17	80		
	8	18	90		
	9	19			
	10 - ten				

N = 123

100

20

3

Num = 123 > 100

= 123 / 100 = 1.23 = 1 → one Hundred

updated Num = 123 % 100

= 23

Num = 23 > 20 → twenty

updated Num = 23 % 20

= 3

Num = 3 == 3 → three

① Find N just 7 = in Map

IF (N >= 100) → find how many nums of Hundred

= N / mp.num → Recursive call

② N = N % mp.num

```cpp
vector<pair<int, string>> mp = {
    {1000000000, "Billion"}, {1000000, "Million"}, {1000, "Thousand"}, {100, "Hundred"},
    {90, "Ninety"}, {80, "Eighty"}, {70, "Seventy"}, {60, "Sixty"}, {50, "Fifty"},
    {40, "Forty"}, {30, "Thirty"}, {20, "Twenty"}, {19, "Nineteen"}, {18, "Eighteen"},
    {17, "Seventeen"}, {16, "Sixteen"}, {15, "Fifteen"}, {14, "Fourteen"}, {13, "Thirteen"},
    {12, "Twelve"}, {11, "Eleven"}, {10, "Ten"}, {9, "Nine"}, {8, "Eight"},
    {7, "Seven"}, {6, "Six"}, {5, "Five"}, {4, "Four"}, {3, "Three"},
    {2, "Two"}, {1, "One"}
};
```

```cpp
string numberToWords(int num) {
    if (num == 0) {
        return "Zero";
    }

    for (auto it : mp) {
        if (num >= it.first) {
            string a = "";

            if (num >= 100) {
                a = numberToWords(num / it.first);
                a = a + " ";
            }

            string b = it.second;
            string c = "";

            if (num % it.first != 0) {
                c = numberToWords(num % it.first);
                c = " " + c;
            }

            return a + b + c;
        }
    }

    return "";
}
```

Wild card matching

? → denotes single character

* → denotes any number of char [0,∞]

s → 'aa'
p → '*' } True

s → 'ab'
p → 'a?' } True

s → 'aa'
p → 'aa*' } True
0,∞...

p → '**ab'
s → 'ab' } True

p → 'ab?d'
s → 'abcc' } False

s → a b c d e f g
p → a b * f g

s → a b c d e f g
p → a b * f g

if (P[i] == '*')

cdefg | *fg
si pi

* is null empty

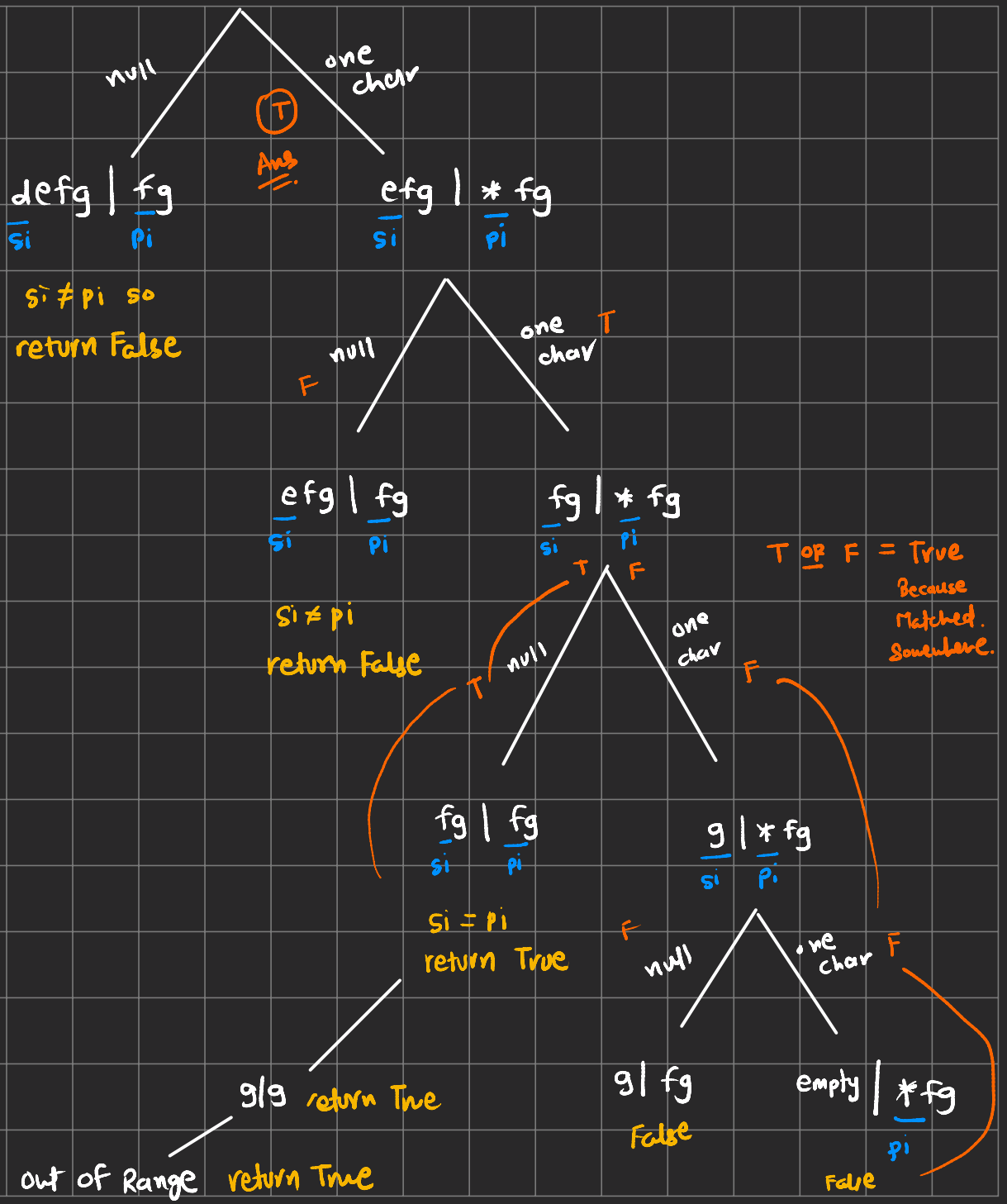
consume one character of s string

cdefg | fg
si pi

defg | *fg
si pi

si ≠ pi so
return False

```cpp
bool isMatcherHelper(string &s, int si, string &p, int pi) {
    // base case
    // 1) complete
    if (si == s.size() && pi == p.size()) {
        return true;
    }

    // 2) s=abc p=abc****
    if (si == s.size() && pi < p.size()) {
        while (pi < p.size()) {
            if (p[pi] != '*') {
                return false;
            }
            pi++;
        }
        return true;
    }

    // rr
    // single char matching
    if (s[si] == p[pi] || '?' == p[pi]) {
        return isMatcherHelper(s, si + 1, p, pi + 1);
    }

    if (p[pi] == '*') {
        // treat '*' as empty or null
        bool caseA = isMatcherHelper(s, si, p, pi + 1);

        // treat '*' consume 1 char
        bool caseB = isMatcherHelper(s, si + 1, p, pi);

        return caseA || caseB;
    }

    return false;
}
```



Perfect Squares

Perfect Square means 1, 4, 9, 16, 25

N = 13

$1^2 = 1+1+1... +1 = 13$

$2^2 = 4+4+4 > 13$ cannot use ✗

$= 4+1+1+1+...$ 9 times $= 13$

$2^2, 3^2 = \underline{4} + \underline{9} = 13$

1 1 → 2

N = 12

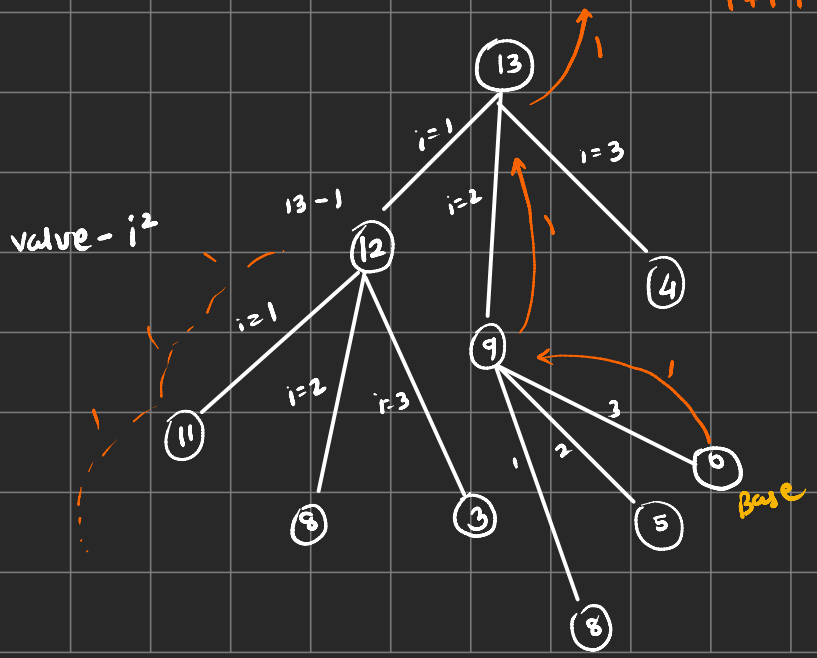
$2^2 + 2^2 + 2^2$

$\underline{4} + \underline{4} + \underline{4}$

3 ans

1+1+1 = 3 but we need 2

= 3 − 1 = 2 ans




```java
int numSquareHelper(int n){
    if(n==0) return 1;
    if(n<0) return 0;

    int ans = INT_MAX;
    int i = 1 ;
    int end = sqrt(n);

    while (i<=end)
    {
        int perfectSquare = i * i;

        int numberOfPerfectSquare = 1 + numSquareHelper(n-perfectSquare);

        if(numberOfPerfectSquare<ans){
            ans = numberOfPerfectSquare;
        }

        i++;
    }

    return ans;

}

int numSquares(int n){
    return numSquareHelper(n) - 1;
}
```

- leaving 2 questions videos (106, 107)