Approach → First thing which comes to mind is Take / Not take

[2, 3, 6, 7] target = 7

So, 3 main parameters will be
f(ind, target, ds[])

f(0, 7, [])

When we are picking we will not increment index, as we can again pick it.

Pick → f(0, 5, [2]) Not Pick → f(1, 7, [])

f(0, 5, [2]) → Pick → f(0, 3, [2,2]) Not → f(1, 5, [2])

f(0, 3, [2,2]) → Pick → f(0, 1, [2,2,2]) Not → f(1, 3, [2,2])

f(0, 1, [2,2,2]) → Not pick → f(1, 1, [2,2]) → Not pick → f(2, 1, [2,2]) → False

f(1, 3, [2,2]) → Pick → f(1, 0, [3,2,2]) → True

X

It will not be able to pick here because target < arr[index])

```
if (ind == n)
    target == 0
    push.
```

Recursive calls:

f(ind, target, ds)

Pick → f(ind, target - a[ind], ds) → if (a[ind] <= target)

Not Pick → f(ind+1, target, ds)

Base Case:

```
if (ind == n)
    if (target == 0) ds → []
    else
        return;
```

where t = target
k = avg length

TC - $2^t \times k$

Because each index has 2 options pick / non pick

Combination Sum II →

Find all unique combinations in arr where the sum == target

Each no. in arr may only be used once in the combination.

Eg 1 → candidates = [10, 1, 2, 7, 6, 1, 5] target = 8

O/P → [[1,1,6], [1,7], [2,6], [1,2,5]]

Eg 2 → candidates = [2, 5, 2, 1, 2] target = 5

O/P → [[2,2,1], [5]]

Combinations should be in lexicographically sorted order.

Q. (PS: above combinations are not in correct order)

Correct order is →

O/P (1) → [[1,1,6], [1,2,5], [1,7], [2,6]]

O/P (2) → [[1,2,2], [5]]

Brute Force → (Inspired from previous ques approach)

Generate all possible sequences

In this approach, we will change f(ind+1, ...) when we include an element in the data structure.

i.e. We will move to the next index after picking an element, because in this ques duplicates are not allowed.

We will store the ans in Set to store unique only

TC → $(2^n) \times K \log N$

SC → ⊗⊗ N for set

Optimal →

This time instead of pick/not pick, we are gonna try to pick subsequences.

f(index, target, [])

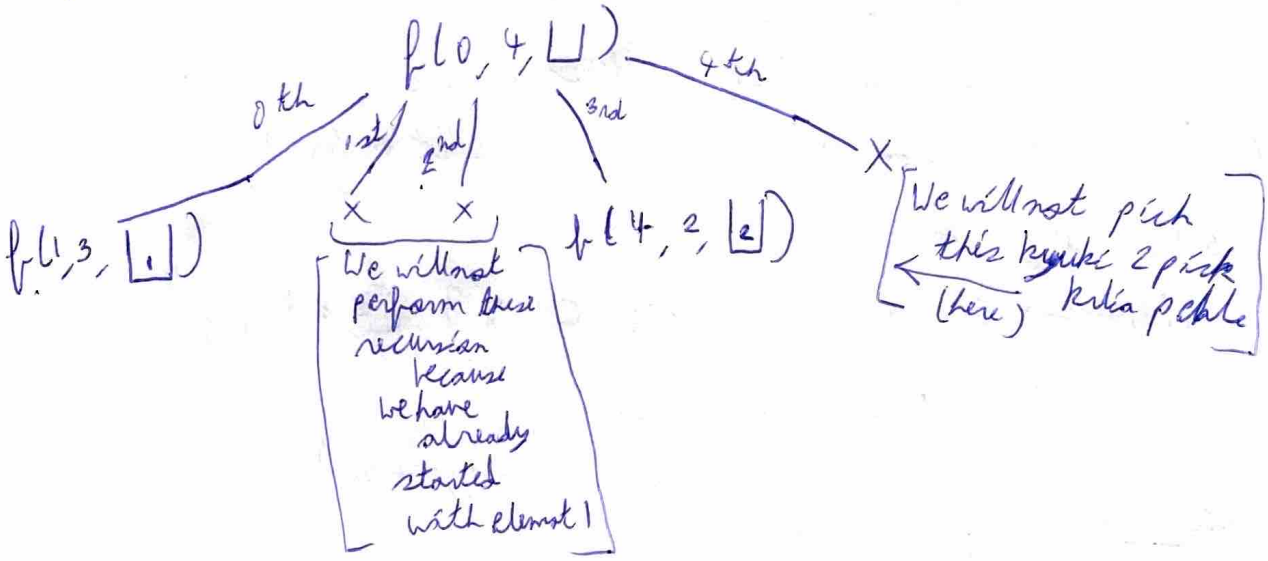
arr[] = [1, 1, 1, 2, 2]

f(0, 4, ⌴)

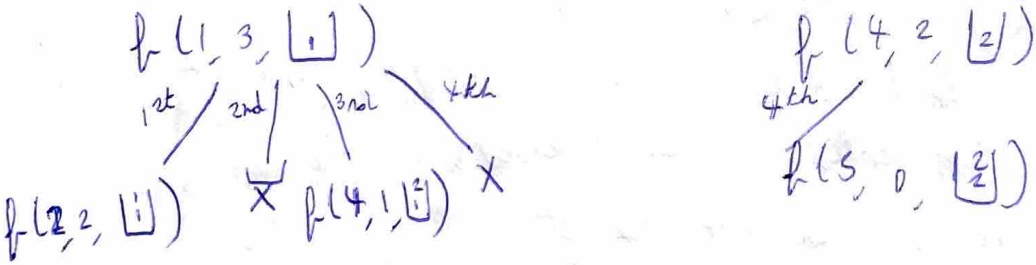
We can start from any index, which means we can pick element from any index.

Eg - Pick 1 from index 0,
Pick 2 from index 3
Pick 2 from index 4
→ 1 2 2

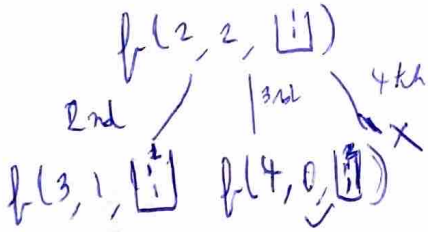
So, in this case we have 5 possibilities of chosing index.



Now to chose 2nd element of combination, we have 4 possibilities



Now to chose 3rd element, we have 3 possibilities



So this is how the recursion tree proceeds, when elements are similar in the array, it will only make one fn call for that index, instead of making call again & again.

(1, 1, 2)
(2, 2) Will be the answer

V.IMP *

f(ind, target, ds, ans)

Now, initially when we are at 0th index, we are trying can we pick index 0, index 1, ... 2, 3, 4 (any index)

We try from 0 till the end.

Then we move to index 1,

we try from 1 till the end.

Similarly, we move to index 2 & try from 2 till the end

Basically, we are looping from index to n-1

Whenever it is possible we make a recursion call, otherwise no recursion call.

f(ind, target, ds, ans)

loop(ind → n-1) → i

Possible / Not possible X

ds.add(arr[i])

f(i+1, target - arr[i], ds, ans)

ds.remove()

if(arr[i] > target)
break;

Base case → if(target == 0)
ans.add(ds)

TC → $2^n \times k$ — k is avg length of every combination

SC → $k \times x$ — x is no. of combinations

Note: Array must be sorted beforehand.

```
f( )
{
if(target == 0){
  ans.push_back(temp); return;
}
for (int i = index; i < arr.size(); i++)
{ if(i > index && arr[i] == arr[i-1]) continue;
  if(arr[i] > target) break;
};
```

← Baaki code to pta hi hai

Subset Sums →

Given an arr of N integers, print sums of all subsets in it.

N = 2 arr[] = {2, 3}

o/p → 0 2 3 5

N = 3 arr[] = {5, 2, 1}

o/p → 0 5 2 7 1 3 6 8

0 1 2 3 5 6 7 8 (Ascending order)

For n = 3, there are 2^3 subsets, i.e. 8

So, SC → 2^n always

Brute Force

Generate all subsets using Power set.

Power set algo takes $2^n \times N$ Time to generate all the subsets.

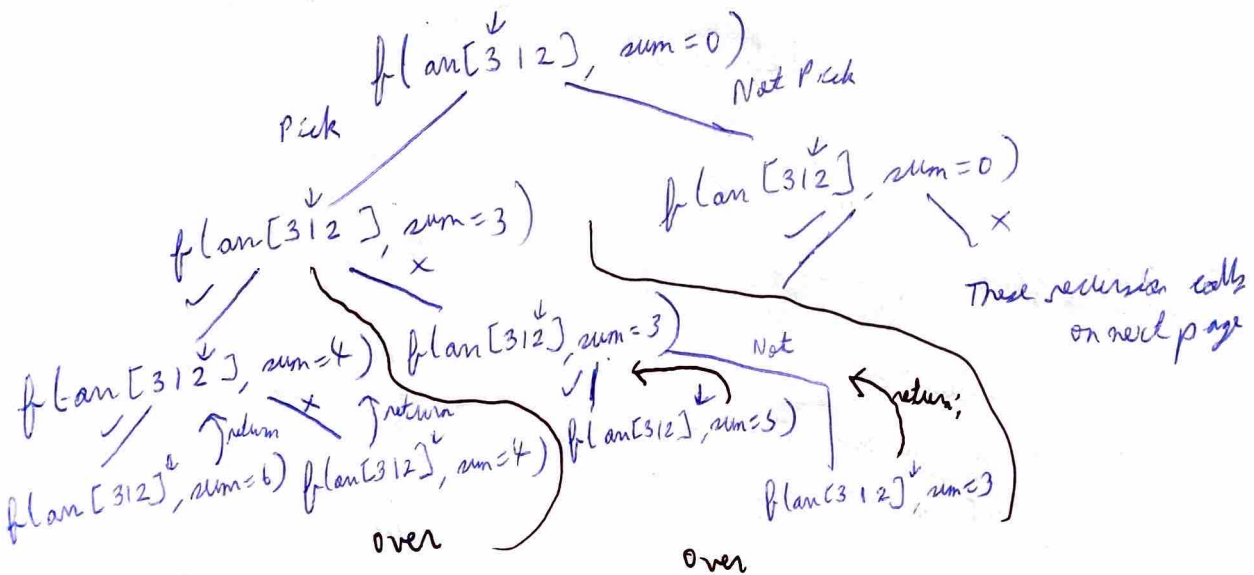
It uses bit manipulation

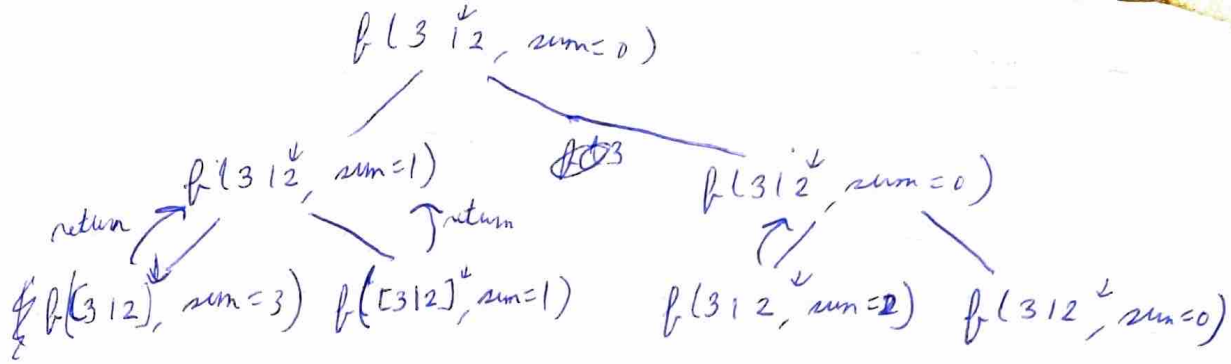
Optimal

We have 3 indices $\underset{0}{\_}\ \underset{1}{\_}\ \underset{2}{\_}$, if we decide which index to chose in subset, and which not to

select e.g. - $\underset{0}{\checkmark}\ \underset{1}{\times}\ \underset{2}{\checkmark}$ → [5, 1]

e.g. - $\checkmark\ \checkmark\ \times$ → [5, 2]

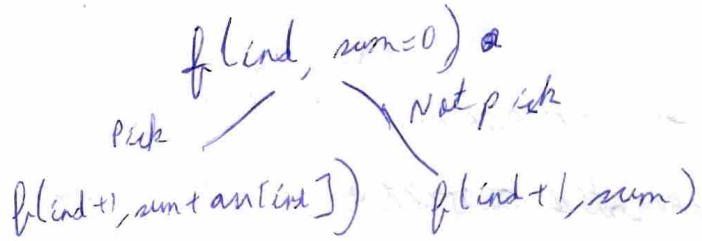




Now, all 8 subset sums get it.

[6, 4, 5, 3, 3, 1, 2, 0]

Sort in increasing before returning



Base case if(ind == n)
Push sum in arr array

TC - For every index we have 2 ways, so 2^n

Subsets II

Given an integer array nums that may contain duplicates, return all possible subsets (the power set).

The solution set must not contain duplicate subsets.
Return the solution in any order.

Eg 1- [1, 2, 2]

o/p → [[], [1], [2], [1,2], [2,2], [1,2,2]]

E.g 2- nums = [0]

o/p → [[], [0]]

Brute Force → We will use the pick & don't pick technique to generate 2^n subsets,

To remove duplicates we will put the subsets in a set first. Then later store in vector<vector>,

Extra time and space used to convert set to vector<vector>.

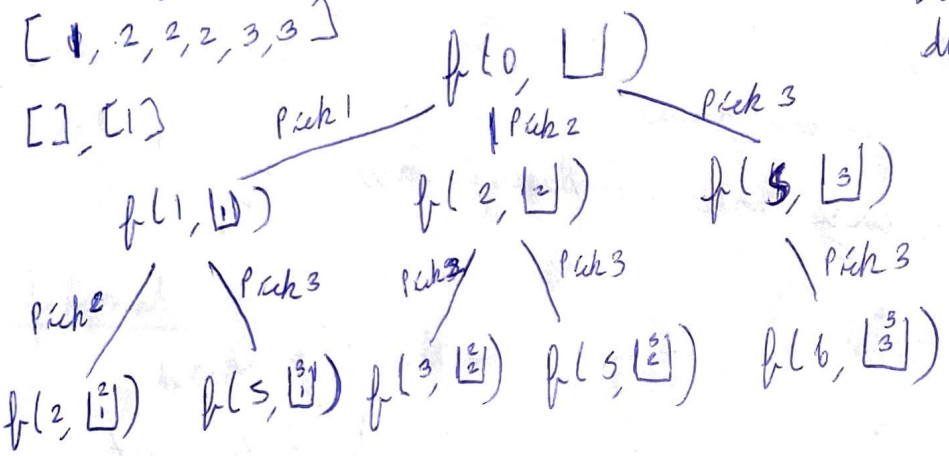
TC – $n \times \log m$ where $m = 2^n$

Optimal →

We have to write recursion in such a way such that at every step it generates a vector.

[1, 2, 2, 2, 3, 3]

We will not pick duplicate elements



First recursion call generated all subsets of size 1, i.e. [], [1], [2], [3]

Second generated subsets of size 2 i.e. [1,2] [1,3] [2,2] [2,3]

Furthermore recursion calls will generate subsets of size 3, 4, 5 & 6

Pseudo code →

f(ind, ds) → size → 2

ds.add(arr[i])

i → ind to n-1 avoiding duplicates

f(i+1, ds)

(2+1)

We will skip recursion calls for duplicate elements

```
if(i > ind && arr[i] == arr[i-1]) continue;
```

$TC \rightarrow 2^n \times n$

$SC \rightarrow 2^n \times k$ - (avg length of subsets)

$+O(n)$

Palindrome Partitioning

Given a string s. Partition s, such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.

E.g 1 s = "aab"

o/p → [["a","a","b"],["aa","b"]]

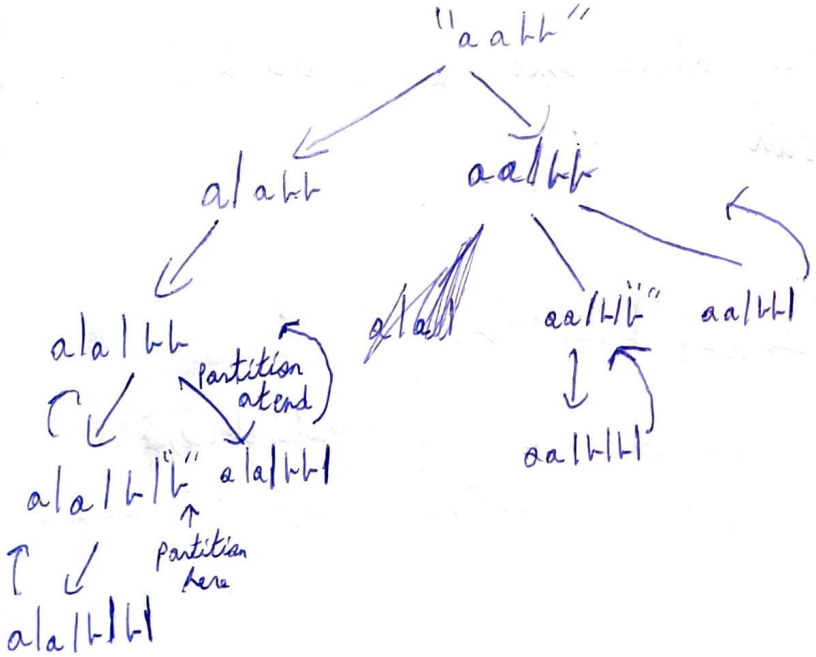
E.g 2 s = "a"

o/p → [["a"]]

Approach: "aabb"

Whenever we do a partition, every substring has to be a palindrome

We try to do partition whenever it is possible



{a,a,b,b}
{a,a,bb}
{aa,b,b}
{aa,bb}

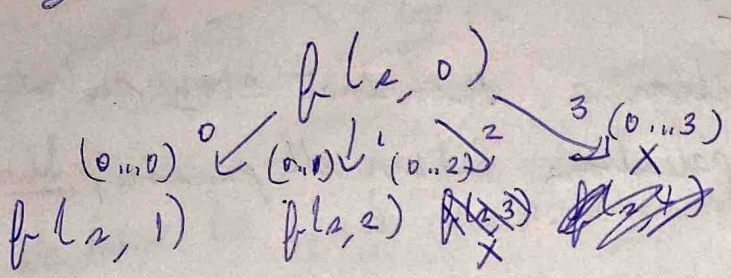
∴ {a,a,b,b} is a partition where every substring is a palindrome

Wherever we are able to ~~generate~~ partition in the end of substring, we will know that we have generated palindrome substrings.

So, basically wherever pointer reaches end, we return.

"a|a|b|b|"

We are trying to partition at all these positions initially.



Which means we will use a loop

Before ~~going~~ making a function call we will check whether it is a palindrome or not.

E.g - We will check (0...0) is a palindrome or not

Yes it is, so make a function call (a)

(0...1) is also a palindrome i.e. (aa)

(0...2) is not a palindrome i.e. (aab)

So, no function call

(0...3) is also not a palindrome i.e. (aabb)

So, no function call

f(s, ind)

↓ loop ind → n-1] → using i variable

(ind.....i) is a palindrome

then ✓ generate f^x n call

Base case →

if (ind == n)

return;

```
bool isPalindrome (string s, int start, int end) {
    while (start <= end) {
        if (s[start++] != s[end--]) {
            return false;
        }
    }
    return true;
}

f ( )
{
// base case
for (int i = index; i < s.size(); i++)
{
    if (isPalindrome(s, index, i)) {
        dr.push_back(s.substr(index, i - index + 1));
        ff(i+1, ...)
        dr.pop_back();
    }
}
```

Kth Permutation Sequence

Set [1, 2, 3, ..., n] contains a total of n! unique permutations.

For n = 3, permutations in order are:

"123" "132" "213" "231" "312" "321"

1 2 3 4 5 6

Given n & k. Return kth permutation sequence

Ex. 2 - n = 3, k = 3

o/p → "213"

Eg - n = 4, k = 9 o/p → "2314"

~~"1234"~~

Brute Force - n = 4, k = 14

Use recursion to generate all the permutations

Store it in a vector, then sort the vector

kth permutation will be at the dr[k-1] index

TC - $n!$ to generate all permutation
$*n$ to store copy of permutation
$+n \log N$ to sort

Optimal Solution →

$n=4, k=17$

For $n=4$, 24 permutations are possible

4 → (1, 2, 3, 4)
0 1 2 3

Starting permutation with 1 → 1 + (2, 3, 4)] 6
" " " 2 → 2 + (1, 3, 4)] 6
" " " 3 → 3 + (1, 2, 4)] 6
" " " 4 → 4 + (1, 2, 3)] 6
24

No. of permutations that start with 1 will be 6, this is same for other no.s also.

1st permutation → 1 2 3 4 (0^{th})

⋮

Last permutation → 4 3 2 1 (23^{rd})

So, we can say that,

Permutations starting with 1 → 6 permutations i.e. (0-5)
" " " 2 → 6 " i.e. (6-11)
" " " 3 → 6 " i.e. (12-17)
" " " 4 → 6 " i.e. (18-23)

So, as $k=17$ is asked in question, we'll be looking for the 16^{th} permutation.

As K=16th permutation lies b/w (12-17), meaning it is starting with 3 _ _ _

We can also determine this by index = 16th permutation / 6 permutations (k/fact)

= 2nd index of initial arr,

i.e. [1 2 3 4]
0 1 2 3
↑ This will be the starting digit, i.e. 3

Now from the 6 permutation in (12-17) range we have to find 16 % 6 = 4th permutation (k%fact)

→ Now, we have to figure out which is the 4th permutation in {1, 2, 4} , K=4 now

As we can see we are at the initial condition again (earlier we were finding {1, 2, 3, 4} & K=16th)

We will follow the same procedure as above

{1, 2, 4}, K=4

Now, Permutation starting with 1 → 1 _ _ [2 permutations i.e. 0-1]
" " " 2 → 2 _ _ [" " i.e. 2-3]
" " " 4 → 4 _ _ [" " i.e. 4-5]

Total = 6 permutations

Now, we can see that K=4 lies in 4-5th permutation i.e. starting with 4.

We will calculate it using this → index K=4 / 2 permutations = 2nd index

in {1, 2, 4}
0 1 2

Now, we have figured out 2 digits

3 4 _ _

Now, value of k left ⇒ 4 % 2 = 0
(k = k % fact)

→ Now we have a set of {1, 2, 3} — we have to find 0th permutation

{1, 2, 3} k = 0

Permutation starting with 1 → 1 _] 1 permutation i.e. 0

Permutation starting with 2 → 2 _] 1 permutation i.e. 1

We can see, permutation will begin with 1.

We will calculate index = k / no. of permutations (fact) = 0 / 1 = 0.

Now, k = 0 % 1 = 0

3 4 1 _

→ Now we have to find k = 0th permutation in {2 3}

Again follow the same process.

3 4 1 2

TC → O(N) × O(N)

SC → O(N)

Code ⇒ Find factorial of n & store 1 to n in an array.

Decrement k string ans = "";

while(true) {

ans = ans + to_string(arr[k / fact]);

arr.erase(arr.begin() + k/fact);

if(arr.empty()) break;

k = k % fact;

fact = fact / arr.size();

}

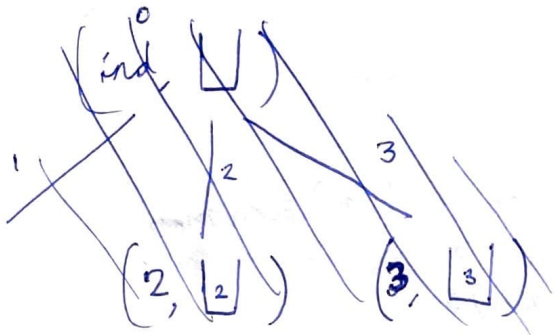
Permutations — Approach I

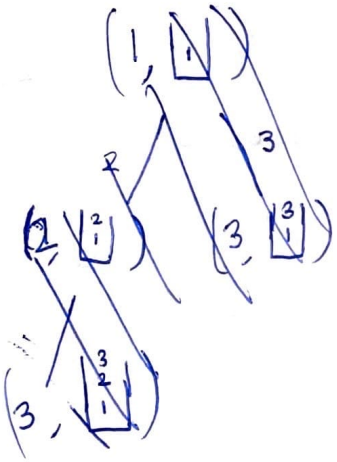
Given an array of nums of distinct integers, return all the possible permutations. You can return the answer in any order.

E.g- [1, 2, 3]

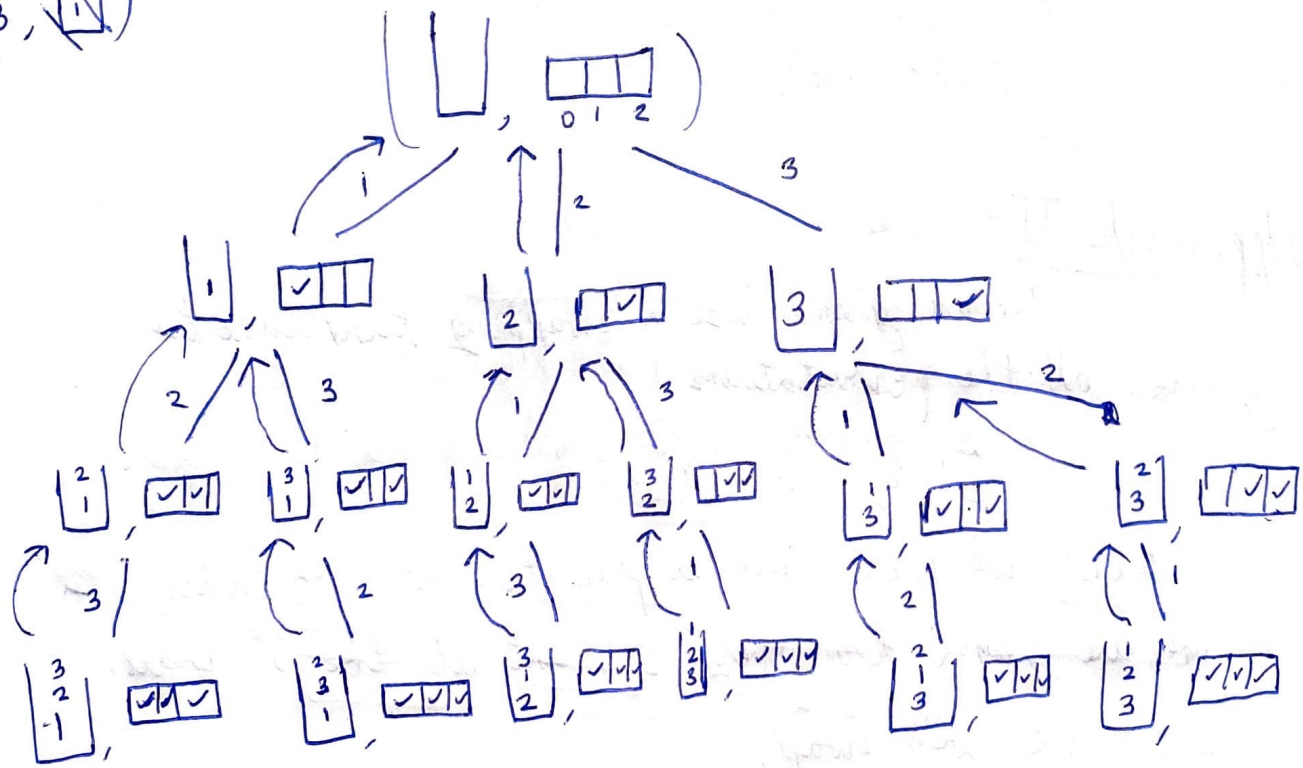
O/P → [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]

- First approach which comes in mind is to generate all permutations





In this approach, we will use a map instead of index to mark which index has been taken and which indexes are left



When we return back pop the element from ds and also unmark the index in the map.

f(ds, map)

↓ loop (0 - n-1)

if (i ! in map)

ds.add(arr[i])

map[i] = 1

↓

f(ds, map)

Base case → if (ds.size == n)

TC - n! × n

SC → O(n) for ds + O(n) for map

Code →

```
for (i = 0 to n-1) {
  if (!mp[i]) {
    mp[i] = true;
    ds.push(nums[i]);
    solve(mp, nums, ds, ans);
    ds.pop;
    mp[i] = false;
  }
```

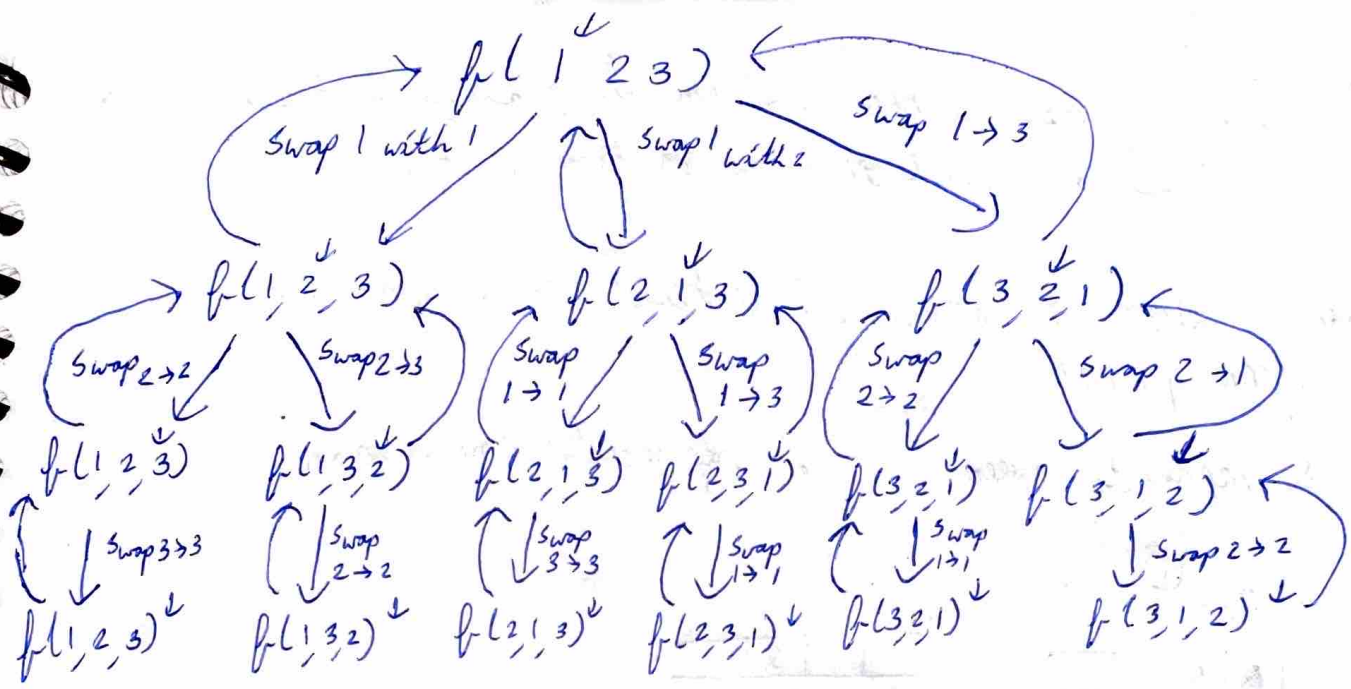
Approach II -

We are gonna use a swapping technique to generate all the permutations.

[1, 2, 3] n = 3

First we will have a pointer at 0th index we are gonna try every element at the 0th index.

i.e. we will swap,



When we go back ~~swap~~ swap ka reverse kar denge

Pseudo code →

```
f (ind, arr)
    loop i → (ind to n-1)
        Swap(a[ind], a[i]);
        f(ind+1, arr)
```

Base case → if (ind == n)

TC → n! permutations + O(n) loop we are running & n time to push permutation into ds.

So, O(n!) X O(n)

SC → O(n) auxiliary space for recursion depth

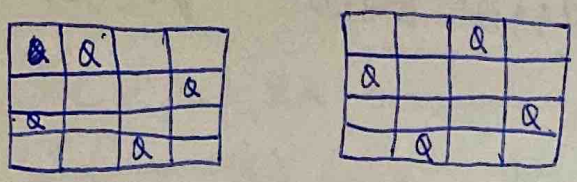
N-Queens →

n-queens puzzle is the problem of placing n queens on an nXn chessboard such that no of two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle.

→ 'Q' represents queen '.' represents empty space

Eg: n=4

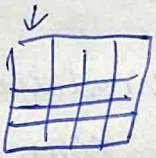


O/P → [[".Q..", "...Q", "Q...", "..Q."],
["..Q.", "Q...", "...Q", ".Q.."]]

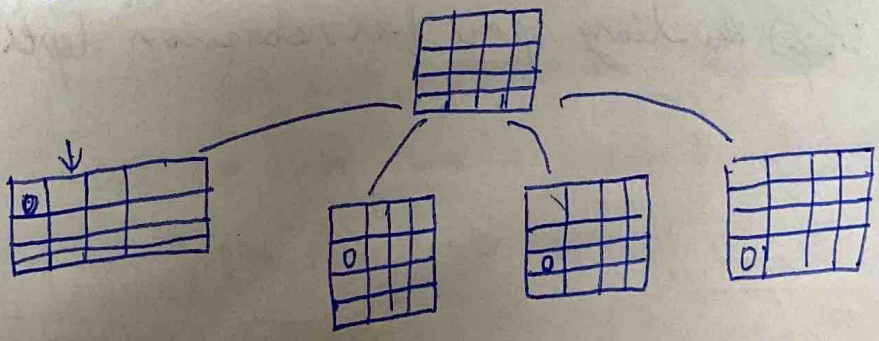
Approach:

Rules →
1) Every row → 1 Q
2) Every col → 1 Q
3) None of the Q should attack each other

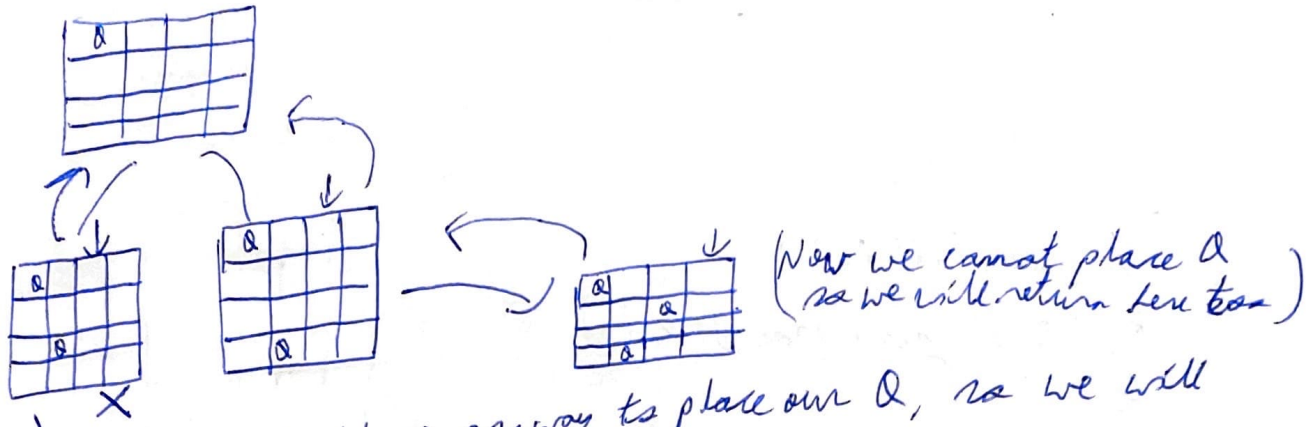
We will write a algo which will try to place Q in all the places starting from 1st column



We have 4 positions where we can place Q in the first column.



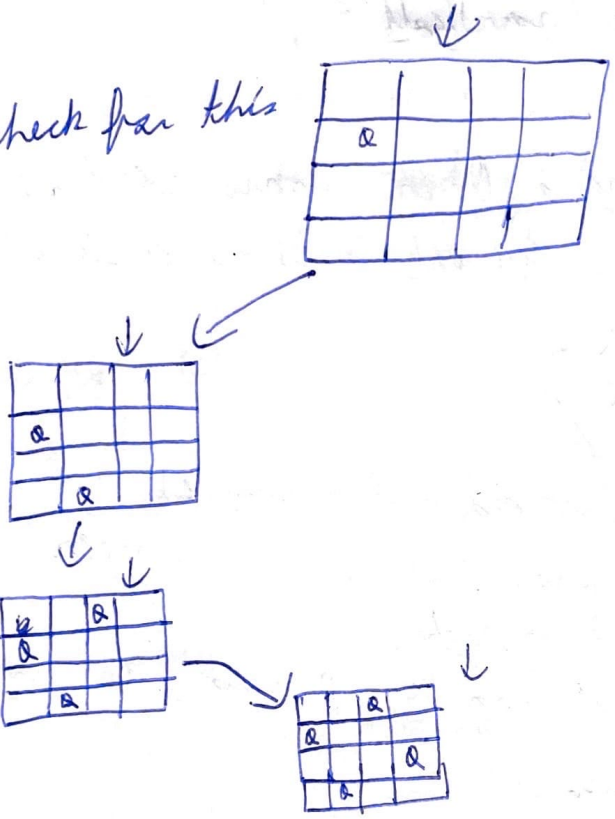
Now we move to col 2, and we will try possibilities In first 2nd row Q cannot be placed, because Q in col 1 will attack Q at these locations. So, now we have 2 possibilities



Now in col 3 we don't have any way to place our Q, so we will return back

When we go back, remember to remove this Q from that index.

Now, we will check for this



Now the moment we see our pointer goes out, we have successfully placed Q at every position

Store this answer in a ds, and ~~ans~~ return.

While returning remove all Q's.

Similarly, we will get the other answer also.

Thought process → We are trying to fill up columns

So, we will have f(col) parameter

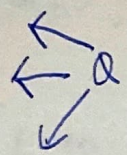
In f(col), we are checking every row from 0 to n-1, whether a Q can be placed or not.

```
f(col)
{

for(i=0 → n-1)
 {
  if(canfill Q){
  Fill Q at given[row][col] = "Q"
  }
}
//Move to next col
   f(col+1);
//When it comes back
Remove Q at given[row][col] = ".";
}
```

Before placing a Q in board, we don't need to check in all directions, we only need to check its left.

E.g.



only these 3 directions will be checked
Because in right we know there will be no queen as in recursion we are moving left → right
And in col only one Q can be placed, so no need to check in the col also.

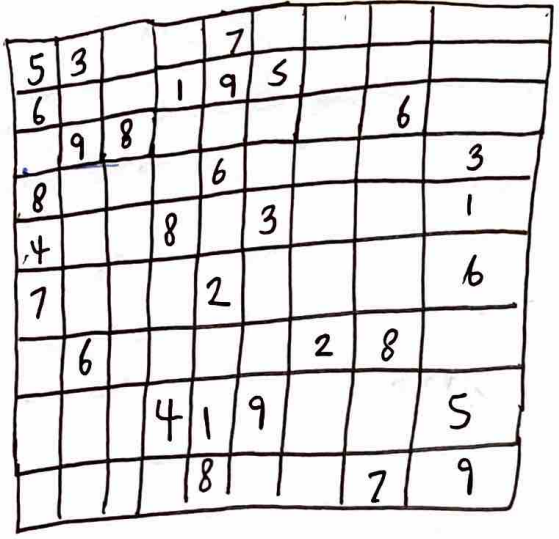
Sudoku Solver →

Write a program to solve a Sudoku Puzzle by filling the empty cells.
A sudoku solution must satisfy these rules:

1. Each of the digits 1-9 must occur exactly once in each row
2. Each of the digits 1-9 " " " " " " " col.
3. " " " " " " " " " " " " " of the 9 3×3 sub-boxes of the grid.

The '.' character indicates empty cells.

E.g-



Approach:

Trying every possible way in empty index

→ As we start traversing the board, we find an empty cell (arr[0][2]). We first try to fill "1" in that cell. Before filling we check its entire row, entire col and 3×3 grid to which that cell belongs whether 1 exists ~~in the~~ anywhere or not. If it doesn't exists we will fill 1.

Another possibilities which we can fill are 4, 2, 7. So, we will fill each and then make separate recursion calls for each.

→ Now, we go to the first recursion call, find next empty cell, consider all the possibilities.

Second empty cell is arr[0][3], no.s which we can put at this place are 2 & 6. We will put both separately and make a recursion call.

Then we go to next lower recursion call, i.e. 2 wale recursion call pe jaunge. In this way, we will proceed.

Pseudocode →

```
for (i = 1 → 9)
{
  if (possible)
  {
    f () → recursion
```

```
{
    for (i=0 to 9)
    {
        for(j=0 to 9)
        {
            if (b[i][j]==':'){
            for (char c='1'; c<='9'; c++)
            {
                if(isValid(b, i, j, c)) {
                    b[i][j] = c;
                if(solve(b) ==true)
                    return true;
                else
                    board[i][j]='.';
                }
            }
            return false;
        }
    }
    return true;
}

bool isVaid ( b , int row, int col, char c){
  {
    for (i = 0 to 9){
        if(b[i][col] ==c) return false;
        if(b[row][i] ==c) return false;
      if(b[3*(row/3) +i/3][3*(col/3) + i%3] == c)
        return false;
    }
  return true;
}
```

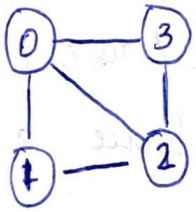
M - Coloring Problem →

Given an undirected graph and an integer M. The task is to determine if the graph can be colored with at most M colors such that no 2 adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices. Print 1 if it is possible to colour vertices and 0 otherwise.

Eg1: N = 4 M = 3 E = 5

Edges[] = {{0,1}, {1,2}, {2,3}, {3,0}, {0,2}}

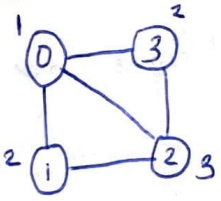
O/P → 1



We have to check whether this graph can be colored with 3 colors?

We assign 1 to 0th node, 2 to 3rd node, 2 to 1st node & 3 to 2nd node

i.e.



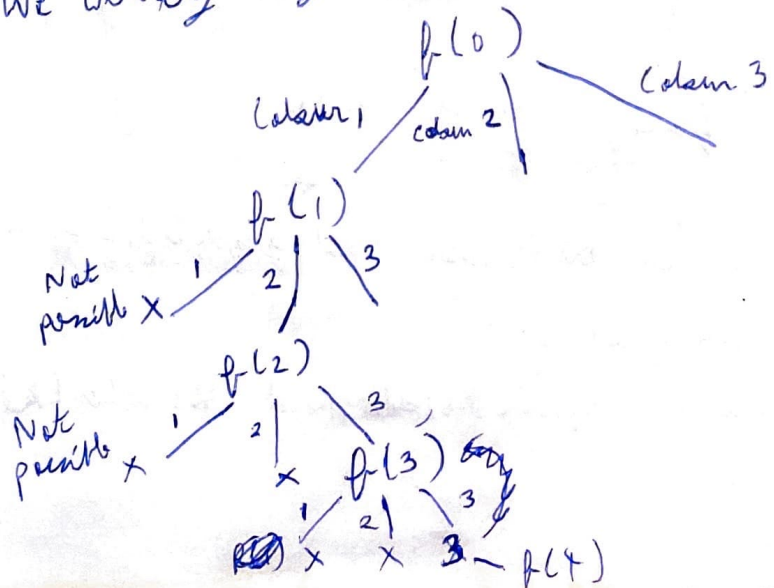
This way all nodes/vertices are colored and no adjacent vertices have same color.

Approach:

Our state: whether we can colour a graph with at most M colors.

So, we will try every possible way.

i.e. We will try every colour on every node.



→ No. of nodes were n, wherever we reach n in recursion call, it will be our base case.

→ Before assigning a colour to any vertex, we will check its adjacent vertices whether they have same colour or not.

→ If they don't have same colour then we will colour the node and then I move to next vertex.

→ Whenever at each function call we will check for every possibility of 1 to M colors, as we are using a loop for this purpose.

→ Whenever we reach base case we will return true. And whenever we get our first true, we will not make further function calls and return ~~true~~ true as the ans.

Pseudo code →

TC → N^M

SC → $O(M) + O(N)$

```
f(node)
{
  if(node == N)          // Base Case
    return true;

  for (col = 1 → M)
  {
    if ( possible → V )
      {
        color[node] = col;
    if ( f(node+1) == T)
        return T;
        color[node] = 0;
      }
  }
return false;
}
```

```
bool possible (int node, int[] color, bool graph[][], int n, int col)
{ for (k=0; k<n; k++){
    if(k != node && graph[k][node] == 1  && color[k] == col)
      return false;
}
return true; }
```

Rat in a Maze Problem - 1

Rat placed at (0,0) in a square matrix of order N * N.
It has to reach destination (N-1, N-1).
Find all possible paths that rat can take to reach from Source to Destination.

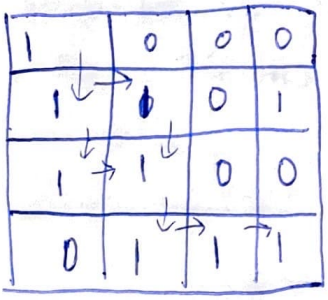
Directions → U D L R

0 means blocked cell
1 means not blocked

Note: - No cell can be visited more than once.
- If source cell is 0, the rat cannot move to any other cell.

Eg - N=4

m[][] = {{1, 0, 0, 0},
{1, 1, 0, 1},
{1, 1, 0, 0}
{0, 1, 1, 1}}



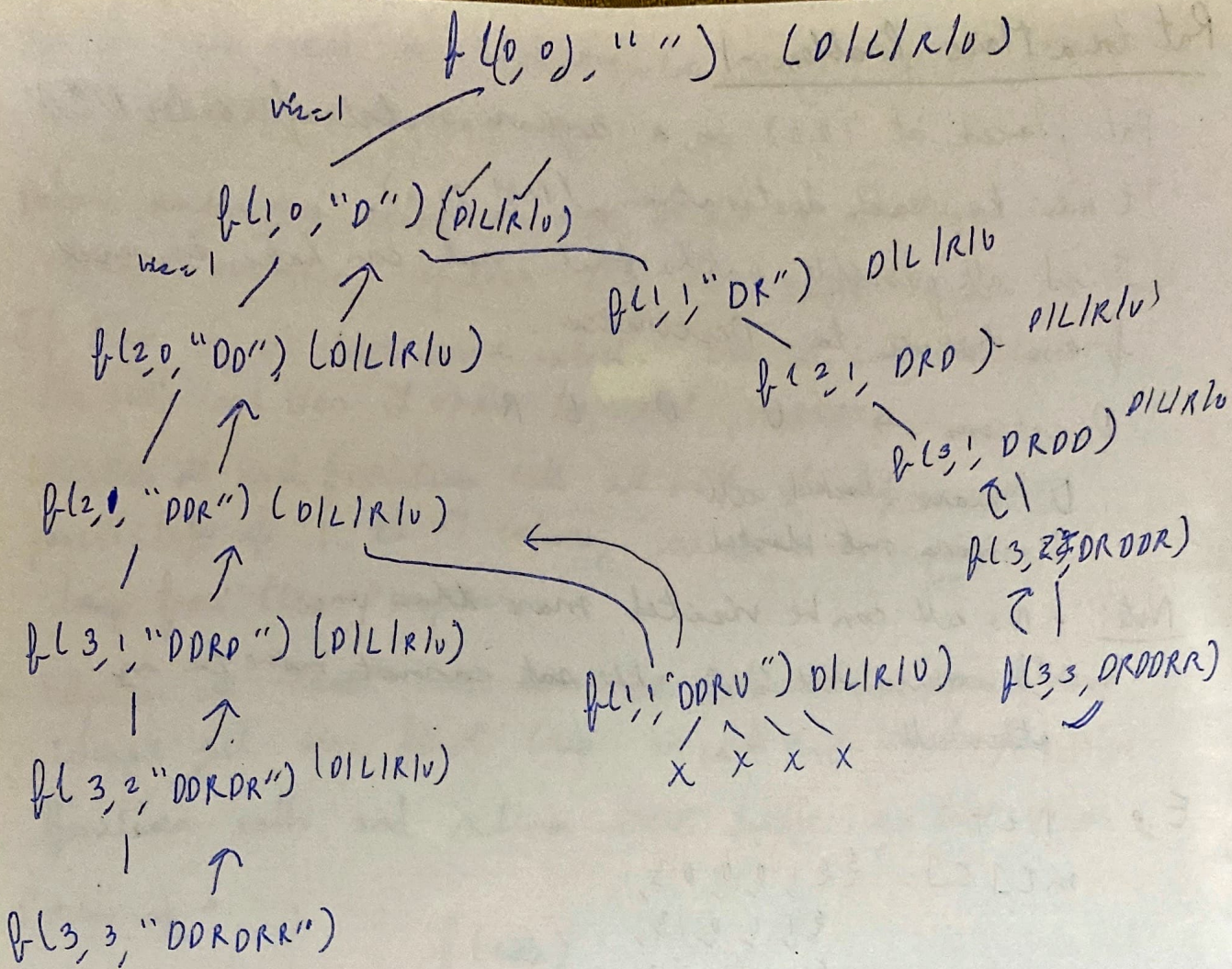
O/P → DRDDRR
DDRDRR

DDRDRR
DRDDRR ✓

(Answer should be in Lexicographical order)

Approach -

- We start from Source i.e. (0,0) and we try every possible path that we can go.
- Since answer should be in lexicographical order, then we will also try directions in this order only i.e. D L R U
- We will also use a visited array. We will mark visited as 1 wherever we leave that cell and move to next recursion call. This is to ensure that we do not visit same cell twice.



We will print DDRDRR, wherever we reach dest.
Then we will return & unmark visited (i.e vis=0.

Pseudo code →

```
f()
{
  down
  f()
  Left
    f()
  Right
    f()
  Upper()
```

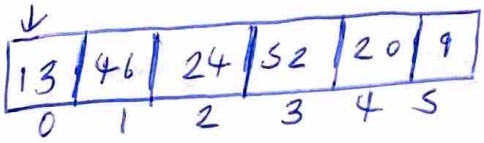
TC - $4^{n \times m}$ because at every call we are trying 4 possibilities

SC - Depth of recursion tree

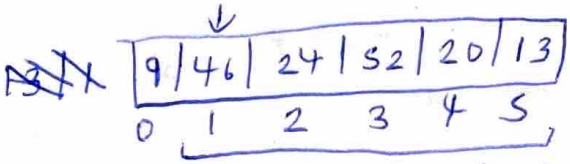
Selection Sort

As the name suggests, we select minimum

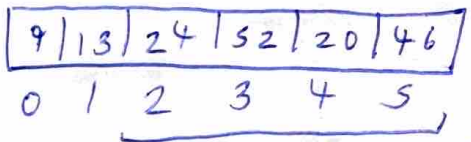
1) Traverse the entire array, determine the min element
2) Place the min in the first index



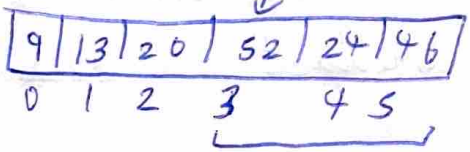
We find 9 is minimum, place 9 at index 0 & place 13 in 9's index. (select min & swap)



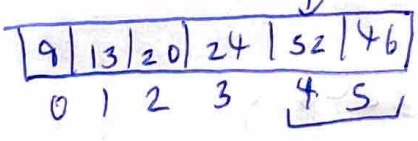
Now in 1 – 5 part of the array find min.
13 is the minimum, swap it with first index in this array



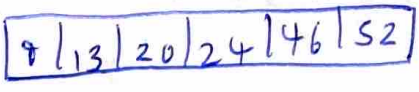
This part of array is still unsorted, so we will perform the same steps as above.
20 is min., swap with first index element



24 is min. now



46 is min. now



So, in 5 steps our array got sorted.

6 elements → 5 steps

So, basically what we are doing is →

- In the entire array figure out the minimum
- On whichever index the min appeared, swap it with the first index

Pseudo code →

```
for (i = 0 → n-1) {
    mini = i;
    for (j = 0 → n) {
        if (a[j] < a[mini]) {
            mini = j;
        }
    }
    if (i != mini)
        swap (a[i], a[mini]);
}
```

TC → $O(n^2)$ → Best
→ Worst
→ Avg.

Bubble Sort

* It pushes the maximum to the last, which is opposite of what selection sort was doing
  - Selection sort was pushing minimum to the front, So, this pushes maximum to the last
- It uses adjacent swapping

[13, 46], 24, 52, 20, 9

(compares first 2 elements, whether 13 < 46 ?

Yes it is ✓

So, we do not do anything

Go to next two elements

13, [46, 24], 52, 20, 9

46 < 24 ?, No, so swap them and move to next 2 elements

13, 24, [46, 52], 20, 9

13, 24, 46, [52, 20] 9
Swap

1st pass

13, 24, 46, 20, [52, 9]

13, 24, 46, 20, 9, 52 This is sorted

After adjacent swaps, we can see that max element reached the last.

Now, perform the same steps

2nd pass

[13, 24], 46, 20, 9, 52

13, [24, 46], 20, 9, 52 → 13, 24, [46, 20], 9, 52
Swap

13, 24, 20, [46, 9] 52 → 13, 24, 20, 9, 46, 52
Swap This is sorted now

Again perform same steps

[13, 24], 20, 9, 46, 52 → 13, [24, 20], 9, 46, 52 → 13, 20, [24, 9], 46, 52
Swap Swap

3rd pass

13, 20, 9, 24, 46, 52
This is sorted now

Again perform same steps

[13, 20] 9, 24, 46, 52 → 13, [20, 9], 24, 46, 52 → 13, 9, 20, 24, 46, 52
Swap This is sorted now

4th pass

Again perform same steps

[13, 9] 20, 24, 46, 52 → 9, 13, 20, 24, 46, 52
Swap Full array sorted now

5th pass

In 1st pass → 0 - n-1
2nd pass → 0 - n-2
3rd pass → 0 - n-3
4th pass → 0 - n-4

So, j loop runs from 0 to n-i-1

```
for(i=0;i<n;i++){
    for(j=0;j<n-i-1;j++){
        if(arr[j]>arr[j+1]){
            swap(arr[j], arr[j+1]);
        }
    }
}
```

TC — $O(n^2)$ — Worst complexity
— Avg complexity

$O(N)$ - Best case if we do some optimisation

Optimised →

We will check whether swap was performed or not,
e.g - 1 2 3 4 5 is sorted,
No swap will be performed in further iterations also
So, we can maintain a bool swap = false earlier
Whenever we do swap, we do swap = true
And after getting out of inner loop, we check
if(swap == false) break;
Means no swap performed.

Insertion Sort →

→ Takes an element & places it in its correct position.

- The algorithm starts with looking at the first element as an array.

[14] 9 15 12 6 8 13

We will check whether 14 is at correct position in this array,
Yes it is, so we include next element.

[14 9] 15 12 6 8 13

Now, is 9 at correct position in this array?
No, so we do swap

9 14 15 12 6 8 13

Now add 15 in array

9 14 15 12 6 8 13

15 is at correct pos, so no swap

Add 12 in array

9 14 15 12 6 8 13

12 is not at correct position

Right shift 14 & 15 & insert 12 after 9

9 12 14 15 6 8 13

Include 6

9 12 14 15 6 8 13 → 6 9 12 14 15 8 13

6 9 12 14 15 8 → 6 8 9 12 14 15 13

6 8 9 12 14 15 13 → 6 8 9 12 13 14 15

∴ The array is sorted

- We are doing right shift and then inserting the element in sorted array

```
for (i = 0; i <= n-1; i++)
{
  j = i
  while (j > 0 && a[j-1] > a[j])
  {
    swap (a[j-1], a[j])
    j--;
  }
}
```

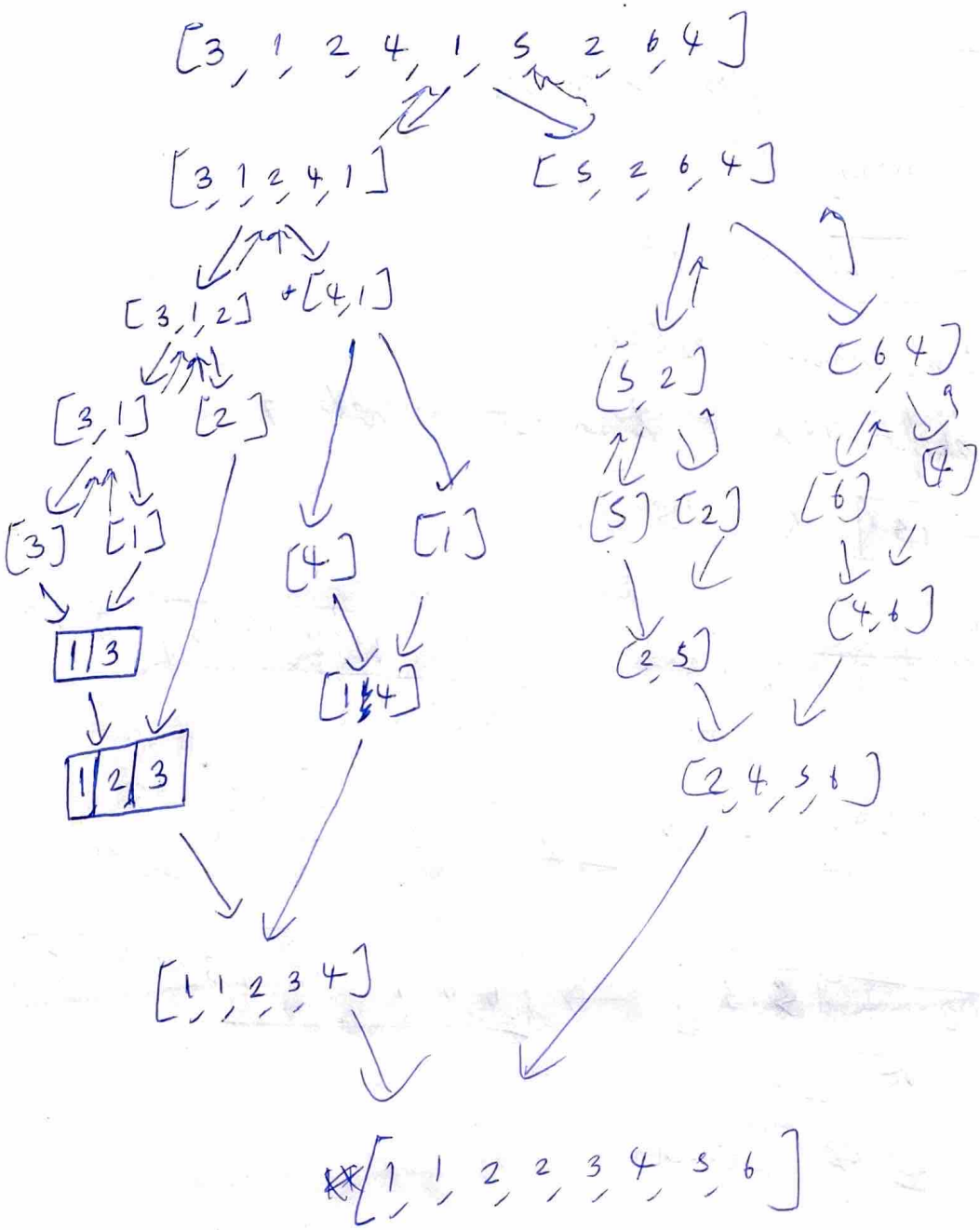
T(n) - $O(n^2)$ - Worst case, Avg case

$O(N)$ - Best case

Slt

Merge Sort

- Much more optimised sorting algo
→ Divide & Merge



TC – O(n log n)

Best Avg Worst

Pseudocode →

```
mergeSort ( arr, low, high )
{
    if (low >= high) return;
    mid = (low+high)/2
    mergeSort(arr, low, mid);
    mergeSort(arr, mid+1, high);
    merge (arr, low, mid, high);
}
```

Quick Sort →

arr[] = [4 6 2 5 7 9 1 3]

TC - O(N log N)
SC - O(1)

Steps:

1. Pick a pivot from the array & place it in its correct place in the sorted array.

Pivot can be: - 1st element in the array
Last element in the array
Median of the array
Random element of the array

Pivot

↓
[4] 6 2 5 7 9 1 3

Place it its correct place in sorted array

2. Smaller on the left and larger on the right

[4] 6 2 5 7 9 1 3

2 1 3 4 6 5 7 9

We put 4 in its place, then we check its further elements if they are smaller we place them in left otherwise right ~~2 1 3~~ & We place them according to how they occur in the array.

So, after 1 step 1 element is at correct place in the array. Left & right portion are still unsorted

Now repeat these steps

[2 1 3] 4 [6 5 7 9]

Lets solve left portion first

pivot
↓
[2] 1 3

[1] 2 [3]
left right

Now, again two smaller elements, now array have one element we don't need to do anything.

[1 2 3] is sorted.

Now, [6 5 7 9]
↑ pivot

[5] 6 [7 9]
left right

5 is a single element, so we don't need to do anything

[7 9]
↑ pivot

Place it in correct position

[7 9]

Now, at the end

1 2 3 4 5 6 7 9

Pseudo code →

low = 0 & high = n - 1

low
↓ i
4 6 2 5 7 9 1 5
↓ high
j

pivot = a[low]

Now to place pivot at its correct position, take i = low & j = high

Now, let's find 1st element which is greater than pivot

if (a[0] > pivot) → No
if (a[1] > pivot) → Yes i.e. 6 > 4

i pointer stops here.

Now, we'll find first element smaller than pivot using j

if (a[j] < pivot) → Yes i.e. 3 < 4

Now, swap both of them

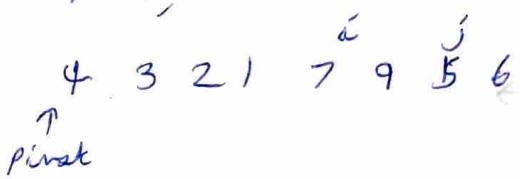
pivot
↓
4 3 2 5 7 9 1 6
i j

Now, again figure out the 1st element > pivot & < pivot, Swap them

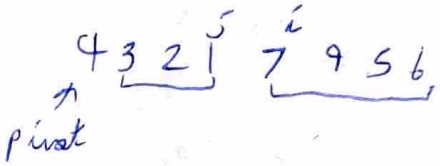
4 3 2 1 7 9 5 6
i j

Now, again find a[i] > pivot

7 > 4, so i reaches 7



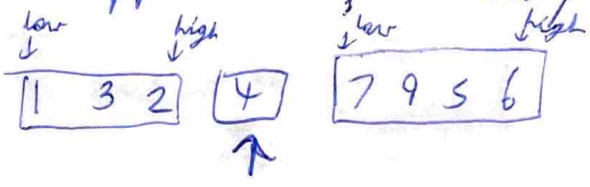
Now find a[j] < pivot



But here j > i, we'll stop we do not swap

Now j is in the territory where everything is smaller than pivot & i in " " " " greater,

Since we stopped at 1 we swap with pivot



This is the partition index

~~Now, high = partition~~,

So, one array (low, partition - 1)

(partition + 1, high)

Perform quicksort again on this

Pseudo code → qs (arr, low, high)

```
{
  if (low < high)
  {
    partition = fp (arr, low, high)
    qs (arr, low, partition - 1);
    qs (arr, partition + 1, high);
  }
}
```

```
fp (arr, low, high)
{
  pivot = arr[low];
  i = low, j = high
  while (i < j)
  {
    while arr[i] <= arr[pivot]
      && i <= high) i++;
    while arr[j] > arr[pivot]
      && j >= low) j--;
    if (i < j) swap (a[i], a[j]);
  }
```

Radix Sort

- Linear sorting algo
- We do digit by digit sorting
-

Implement Queue using Arrays →

front & rear ptr

```
push(x){
    if(cnt == n) return -1;
    a[rear % n] = x;
    rear++;
}
```

```
pop(x) {
    if(cnt == 0) return;
    a[front % n] = -1;
    front++;
    cnt--;
}
```

```
top(){
    if(cnt == 0) return -1;
    return a[front % n];
}
```

Implement Queue using stack

We need min 2 stacks for this purpose

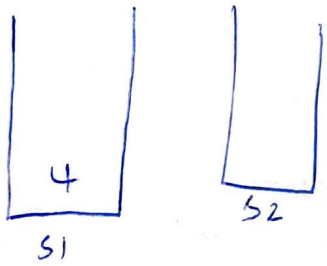
push(x)

- s1 ↔ s2
- x → s1
- s2 → s1

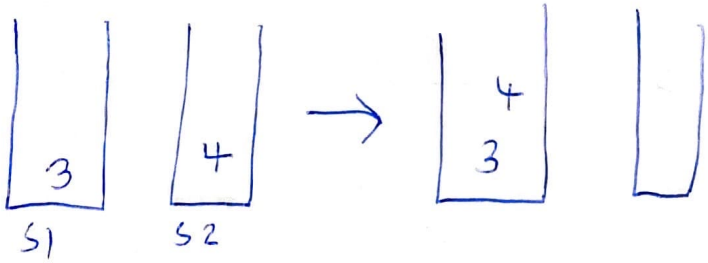
pop(x)

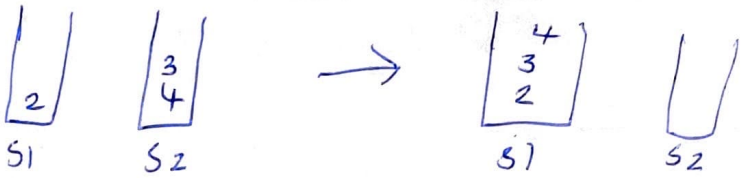
s1.pop()

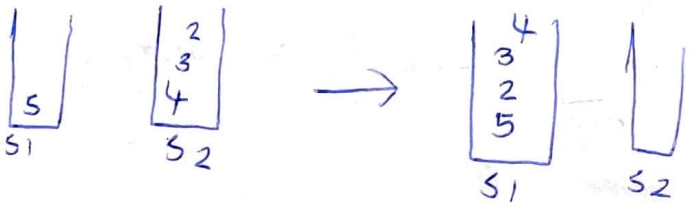
Push(4)
Push(3)
Push(2)



Push(3)



Push(2) → 

Push(5) → 

top() → 4 (Follows FIFO now)

pop() → 4 gets popped

top() → 3

TC → O(N)

SC → O(2N)

Optimal Soln:

In order to decrease TC of push operation, we will take 2 stacks named Input & Output

Push(x)

- Add x → input

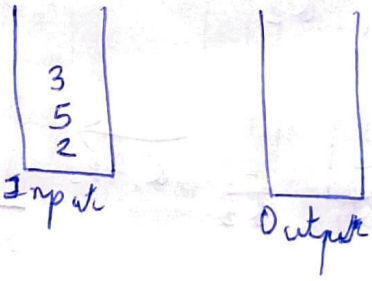
pop()

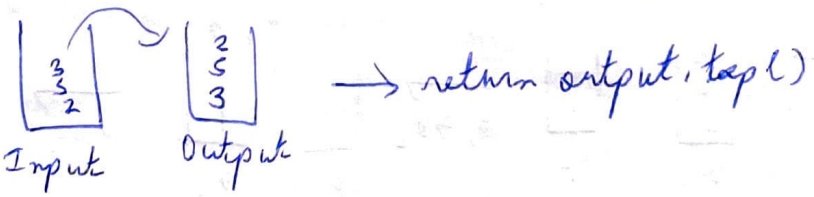
if (output not empty)
  output.pop()
else
  input → output
  output.pop()

top()

if (output not empty)
  return output.top
else
  input → output
  output.top

Push(2)
Push(5)
Push(3)



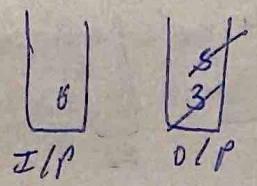
O(N) top() → 

→ return output.top()

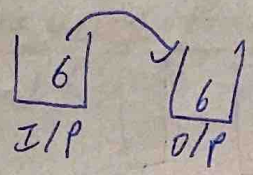
pop() → Output not empty, so pop it. O(1)

push(6)

pop()
pop()



top() →



→ Return output. top()

O(N)

TC → O(1) amortized complexity

SC → O(2N)

Implement Stack Using Queue →

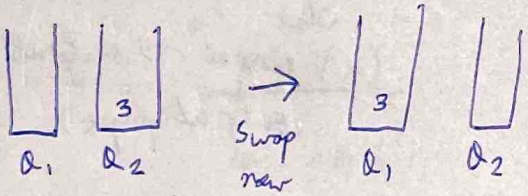
Take 2 queues

Steps →

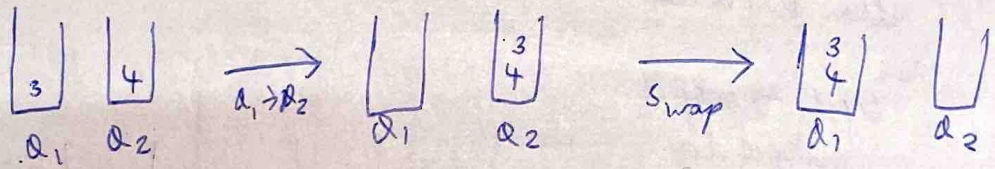
Push(x)
- Add x → Q2
- Q1 → Q2 (elementby element)
- Swap (Q1 ⇄ Q2)

Pop()
- Remove top of Q1

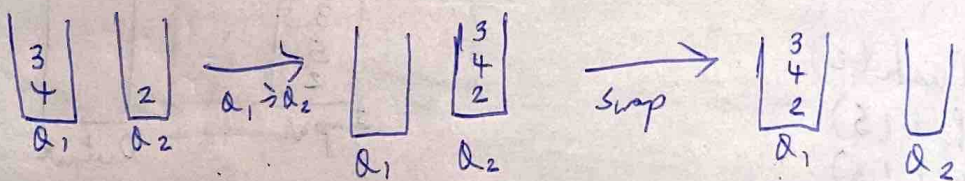
Push(3) →



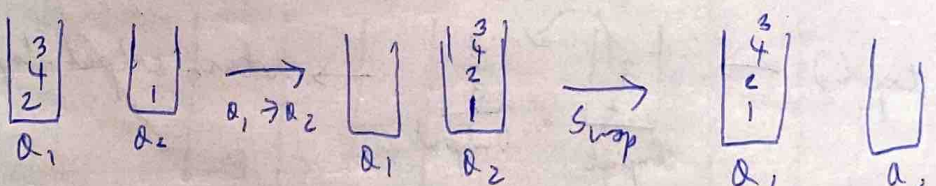
Push(4) →



Push(2) →

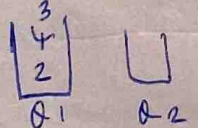


Push(1) →



top() → 1 i.e. LIFO

pop() →



top() → 2

TC → O(N)
SC → O(N)

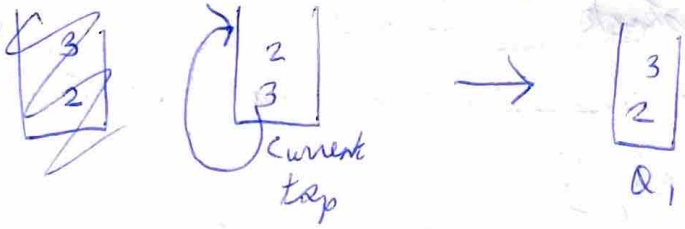
Optimal - Using Queue (Single)

Push(3) → 

Push(2) → 

size = 2

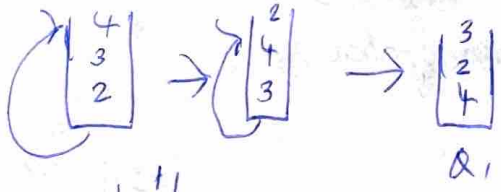
Now, as size = 2, one lesser than that, i.e. for 1 time Take the top most element from queue & push it again.

i.e. 

Push(4) → 

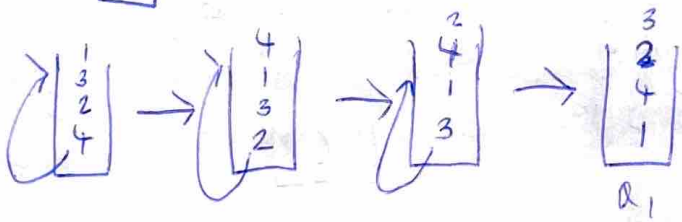
size = 3

So we will do above operation 2 times i.e. size-1 times

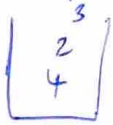


Push(1) → 

Now size = 4, do 3 times



top() → top is printing 1, i.e. it is following LIFO

pop() → 

top() → 4

Valid Parentheses

s = ( ) [{ } ( )] ✓

s =] ( ) { } X

s = [ ) ( ] X

Approach:

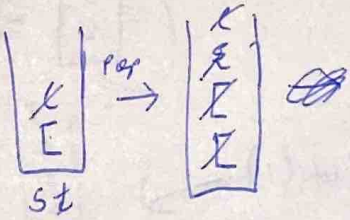
- We will use stack
- Whenever we encounter a opening bracket, push it into the stack [({
- Whenever we encounter a closing bracket, check if stack is not empty.

~~If~~ If it is not empty, then for the closing ~~tag~~ bracket, there might be a opening bracket in stack

- Take the top element out, if they are corresponding, then pop it otherwise return false.

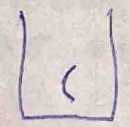
e.g - [( ) [{ ( ) }]]

↑ ↑ ↑ ↑ ↑ ↑



if (st.empty()) return true;

( ) [ ] { }



TC - O(N)

SC - O(N)

Next Greater Element

In an array of N integers, we have to figure out for every element which is the next greater element

3	10	4	2	1	2	6	1	7	2	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	10
10	-1	6	6	2	6	7	7	9	9	

nge[] => 10 -1 6 6 2 6 7 7 9 9 10

N = 11

For last element i.e. 9, we will look from starting for its next greater

Brute Force-

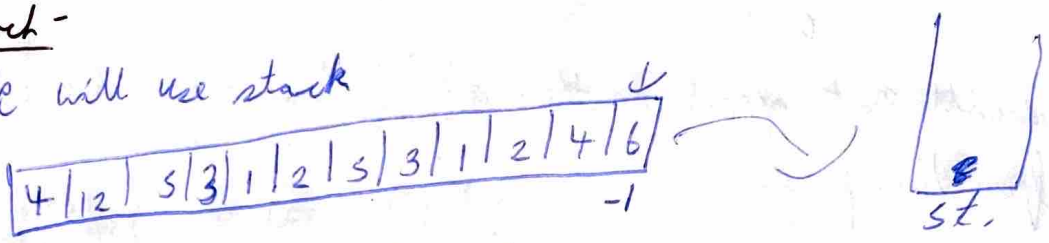
Using 2 loops

```
for (i=0 to n-1)
    for(j=i+1 to n-1)
        if (a[j] > a[i])
            nge[i] = a[j];
            break;
        else
            nge[i] = -1;
```

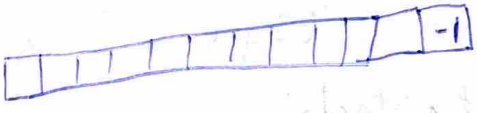
TC- $O(N^2)$

Optimal Approach-

We will use stack



We will start from the last element.
- We will see whether there are elements < 6 in the stack
- Right now, no element < 6, so we push 6 in stack and put nge[i] = -1
  Then do i--;



Now ele = 4.

4 12 5 3 12 5 3 1 2 4 6

elements < 4 in stack, No
Since there are no lesser elements in the stack, take the topmost guy, which will be our next greater element
Then push 4 in stack

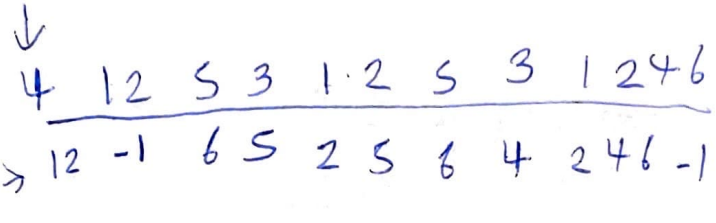
4 12 5 3 12 5 3 1 2 4 6

2 4 6 -1

nge →

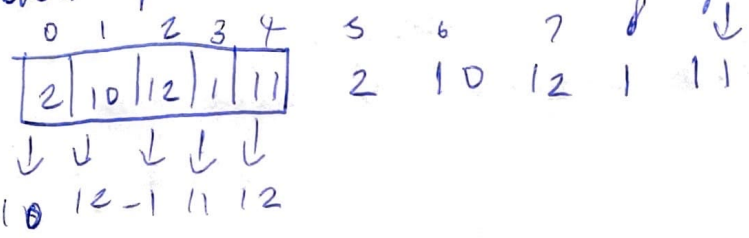
Now, ele = 3, there are elements < 3 in stack
pop them until st.top < 3.

Similarly, we will proceed in this way



Now for circular array →

We copy paste the portion once again, and then we will just solve the problem like above



We will not append another array, but we will use i % n for this purpose.

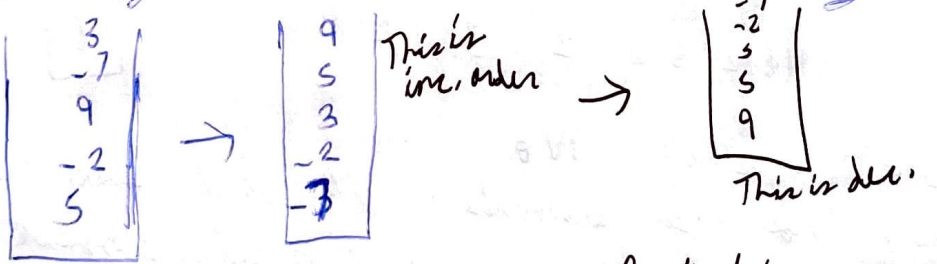
TC - $O(2n + 2n)$

$\approx O(N)$

SC - $O(N)$

Sort a Stack

N integers Sort stack in decreasing order



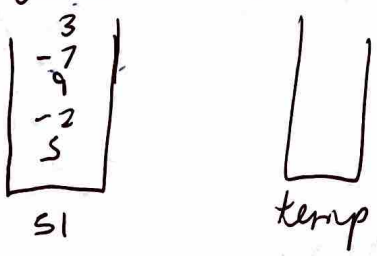
Brute Force → We copy the contents of stack in an array then sort the array, then again store the ~~arr~~ array in stack

TC - $O(N) + O(N \log N) + O(N)$

SC - $O(N)$

Better- What if we use two stacks

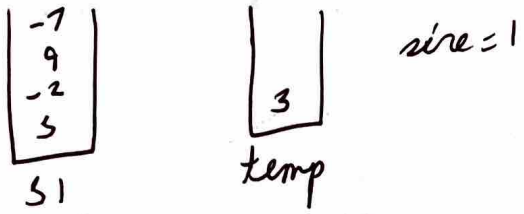
We will create a temp stack



If temp.empty() → Push s1.top() & s1.pop

In the temp array we will sort elements in increasing order by doing a sorted insert

Before inserting we will compare st.top()



If st.top() ≤ temp.top()

{

while (size){ rather than temp array

//Store elements in

size--;

temp.pop();

}

Now push st.top() in temp & do s1.pop()

Now our stacks are

Follow above steps

Now again -2 < 9

an

9 3

Implement LRU Cache

Design a data structure that follows the constraints of a LRU cache.

- LRUCache(int capacity) initialize the LRU cache with positive size capacity.
- int get(int key) Return the value of the key if key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists.

Otherwise, add the key-value pair to the cache. If no. of keys exceeds the capacity from this operation, evict the least recently used key.

E.g:

size = 2

put(1,1) {{1,1}}

put(2,2) {{1,1}, {2,2}}

get(1) → 1

put(3,3) → But here our capacity is full, so we will remove Least Recently Used pair and insert this.

i.e, {2,2}

{{1,1}, {3,3}}

get(2) → -1

put(4,4) → {{3,3}, {4,4}}

get(1) → -1

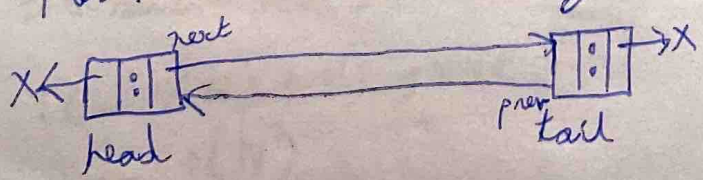
get(3) → 3

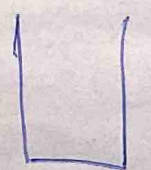
get(4) → 4

Do these functions in O(1) complexity

Approach

For this we need a doubly linked list & a hashmap

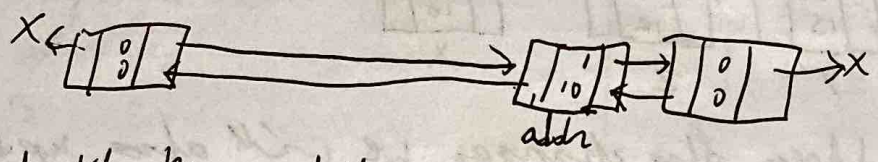




(Key, node)

- Put(1, 10) We will check whether (key, node) exists in hashmap or not, If it doesn't then we will insert it right after head

- Here, we are using doubly linked list to maintain order of how elements are coming, with this we can determine which is LRU element.



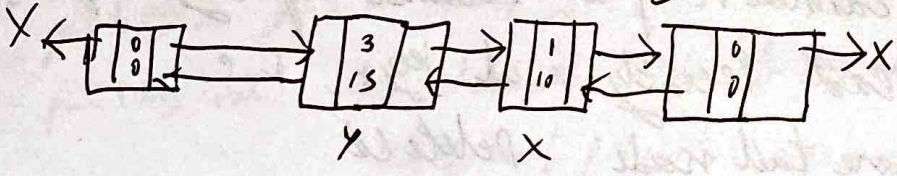
Take the key, and its addr. and store it in hash map

X - addr

(key, node)

Put(3,15)

Perform same steps as above, If it ~~exists~~ does not exist in map, then insert it right after head



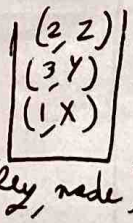
Store key, addr in map

(key, node)

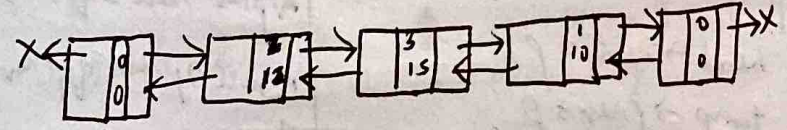
Similarly, now

Put(2, 12)

Perform same steps & insert at addr. Z



(Key, node)



As we can see, elements are being inserted in the way they were used, i.e. least recently or most recently

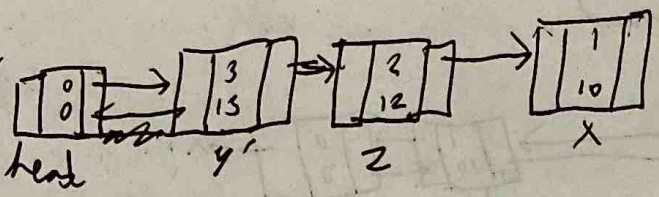
Now, get(3), We will check whether it exists in hashmap or not.

Yes, it exists, we will go to the corresponding address and fetch.

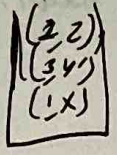
Now, as we have used 3 most recently, we will also have to move it in linked list. To move it

- Delete the node first, then
- Insert it right after head.

Now,



As its address also changes, we will also update in hashmap.



But max size = 3

Put(4,20)

Step 1 - Check in hashmap if it exists or not, No it doesn't

Step 2 - Check the size of current the cache, i.e. 3 & max capacity also 3.

So (4,20) cannot be put, because capacity is full

Step 3 - Remove Least Recently used key, i.e. right before tail node. Delete it & do tail->prev = tail->prev->prev

Code → Similarly we will proceed

Make a doubly LL →

```
class Node {
    int key, val;
    Node *next, *prev;
    Node(int k, int v) {
        key = k;
        val = v;
    }
};
```

```
void addNode(Node *p) {
    Node *temp = head->next;
    p->next = temp;
    p->prev = head;
    head->next = p;
    temp->prev = p;
}
```

```
void deleteNode(Node* q) {
    Node *qprev = q->prev;
    Node *qnext = q->next;
    qprev->next = qnext;
    qnext->prev = qprev;
}
```

```
int get(int key){
    if(mp.find(key) != mp.end()){
        Node *resNode = mp[key];
        int res = resNode->val;
        mp.erase(key);
        deleteNode(resNode);
        addNode(resNode);
        mp[key] = head->next;
        return res;
    }
    return -1;
}
```

```
void put(int key, int val){
    if(mp.find(key) != mp.end()){
        Node *existing = mp[key];
        mp.erase(key);
        deleteNode(existingNode);
    }
    if(mp.size() == cap){
        mp.erase(tail->prev->key);
        deleteNode(tail->prev);
    }
    addNode(newNode(key, value));
    mp[key] = head->next;
}
```

LFU Cache (Least Frequently Used) Cache

We have to design a data structure with 2 functionalities i.e. get() & put()

get(key) → get the value of key if it exists, else -1

put(key, val) → update the value of key if present, or insert key if not present.

When the cache is full, it removes LFU one & if tie, then it removes LRU one.

Design in O(1) complexity

E.g- size = 2

put(1,10)

put(2,20)

get(1) → 10

freq

1 → (1,10) (2,20)

2 → (1,10)

→ (1,10) & (2,20) has freq = 1 for now.

→ As get(1) is called, its frequency increases

put(3,30) Now our cache is full, as size == 2, so we will remove the LFU guy. Here, LFU element is with freq == 1.

i.e. we will remove (2,20)

Now,

1 → (3,30)

2 → (1,10)

get(2) → -1 → 2 is not there in our cache

get(3) → 30 → freq of 30 will increase

① → ×

2 → (1,10)(3,30)

put(4,4) → Now cache size == 2, we will have to remove the LFU.

Here LFU is 2 in this case, and with freq == 2, we have 2 elements.

So, here is a tie, now we will remove the LRU.

(1,10) is LRU, so we will remove it

① 1 → (4,4)

2 → (3,30)

get(1) → -1

get(3) → 30

get(4) → 4

1 → (4,4)

3 → (3,30)

2 → (4,4)

3 → (3,30)

Implementation →

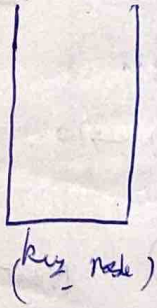
In order to implement LFU, we will use 2 hashmaps and a doubly linked list

map<freq, list> freqList ↗ DLL

map<key, Node*> KeyNode

capacity = 0

freq = 0



freqList maintains freq of List

keyNode will store the key & its address

freq variable tracks which is the least frequency guy as of now.

size = 3

Put(1,10)

Steps: 1) Check whether I exists previously in keyNode map
2) Check is there space in cache
3) Now put it in cache

1 → [(1,10)] DLL just like LRU
x

4) Update least frequency, i.e. freq = 1
& capacity = 1

(1,x)

5) Store (key, addr) in keyNode map

Put(2,20) perform same steps

1 → [(2,20) (1,10)]
y x

(2,y)
(1,x)

capacity = 2 & least freq = 1

Put(3,30) perform same steps

1 → [(3,30) (2,20) (1,10)]
z y x

(3,z)
(2,y)
(1,x)

capacity = 3 & freq = 1

Put(4,40) Now capacity == size
So, we will have to remove LFU
We will check what is the least freq using freq variable.
Here, freq = 1, which has 3 elements
So, we will remove LRU one, which is the same way we did in previous ques. i.e. remove tail's prev.

1 → (3,30) (2,20) (1,10)

Now put (4,40)

1 → [(4,40)(3,30)(2,20)]
W Z Y

(4,W)
(3,Z)
(2,Y)

(key, node)

get(3) → 30 Yes it exists in key node map
update its freq

1 → [(4,40) (2,20)]
W Y

2 → [(3,30)]
Z

Its address will not change

get(2) → 20
Change freq

2 → [(2,20)(3,30)]
Y Z

get(4) → 40

2 → [(4,40)(2,20)(3,30)]

Now our freq → 1 list becomes empty
We will update freq = 2

Put(5,50) 1 → [(5,50)] freq = 1 again
2 → [(4,40)(2,20)]

Put(2,25) 1 → (5,50)
2 → (4,40)
3 → (2,25)

Code implementation is in LC 460. LFU Cache

Largest Rectangle in Histogram

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

E.g 1 heights = [2, 1, 5, 6, 2, 3]

O/P → 10

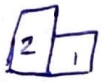
Largest rectangle is highlighted

height = 5 width = 2

So, area = 5 × 2 = 10

Brute Force -

We will try every histogram block as a particular rectangle.

If we take 2,  we cannot extend its width towards right, as 1 < 2, so we will take width = 1 & height = 2

Area = 2, we will maintain this in max variable

If we take 1 as height, we will try to extend its width in both left & right directions.

As we can see max width we can achieve with height = 1, is 6, So, 6 × 1 ⇒ Area = 6

~~To compare~~ To check till where we can extend width we will check if height[j] <= height[i] → width++;

where j is index of elements to i's left & right.

Similarly, we will proceed and calculate max area

We will use 2 loops to find leftSmaller & rightSmaller

width = rightSmallerIndex - leftSmallerIndex + 1;

TC - $O(n^2)$

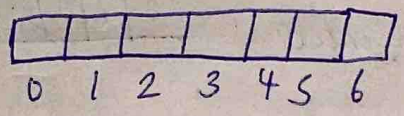
SC - $O(1)$

Better

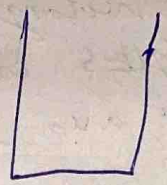
Now to ~~calculate~~ avoid finding left Smaller & rightSmaller for each index, we will create a

leftArray of size n

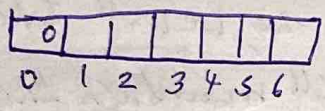
Also create a stack which will store indexes

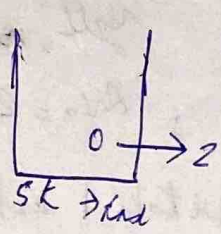
leftArr → 

Arr → [2, 1, 5, 6, 2, 3, 1]



Start iterating from 0th index, check in stack is there any greater element. Since, there is not, put index in stack & leftArr.

leftArr → 

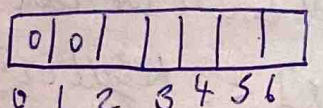


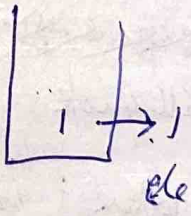
Now, ind = 1,

element ≤ value in stack.

We don't want to store larger elements in stack, so we will pop out greater value, and empty our stack.

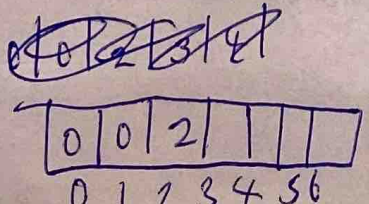
Now, on the left we don't have any smaller element < 1, so push this in stack & update leftArr

leftArr → 

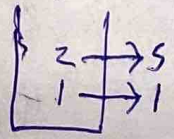


Now, ind = 2, value = 5

check in stack, yes ele ≤ value, so we will add 1 to the index in stack and ~~store~~ store it in leftArr

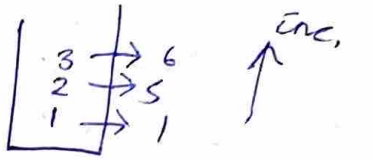


Our leftArr is representing boundary



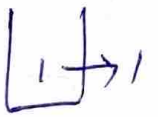
Now, ind = 3, smaller will become 5, which is on its left

leftSmall → | 0 | 0 | 2 | 3 | | | |
0 1 2 3 4 5 6



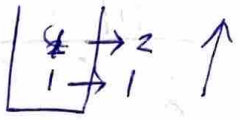
Now, ind = 4, val = 2, we will find next smaller by comparing in stack

Now, the element in stack is smaller than curr val



So, update leftArr & stack

leftArr → | 0 | 0 | 2 | 3 | 2 | | |
0 1 2 3 4 5 6



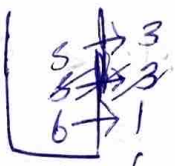
Similarly,

leftArr will become

| 0 | 0 | 2 | 3 | 2 | 5 | 2 |
0 1 2 3 4 5 6

We will do similar thing to compute rightSmaller Array

rightArr → | | | | | 5 | 5 | 6 |
0 1 2 3 4 5 6



st (ind → val)

We will store ind - 1 in rightArr & compare whether elements in stack are greater or lesser, If greater we will pop out. Because we have to find smaller element in right. The moment we find smaller element, we will store its ind - 1 in rightArr.

So, our arr will become

rightSmall → | 0 | 6 | 3 | 3 | 5 | 5 | 6 |

Now, calculate area for each index, and maintain a large, element.

$TC \rightarrow O(N) + O(N) \approx O(N)$

$O(N) + O(N)$

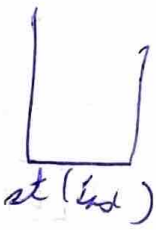
$SC \rightarrow O(N) \times 3$

Optimal

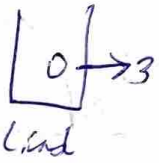
We will convert previous 2 pass soln to 1 pass soln in this ~~str~~ method.

We were using 2 separate passes to calculate leftSmaller & rightSmaller.

maxA = 0



At ind=0, ele=3, st is empty, so store it in stack



At ind=1, ele=1, st is not empty,

compare it with element in stack.

ele < element in stack

So, for 3, right smaller is index 1

Pop 3 from stack,

For index=0, right Smaller=1

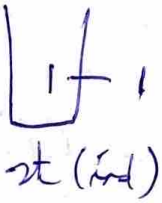
left Smaller will ~~~~ be 0,

because we don't have anything on left.

So, whatever is right Smaller, for total will become width.

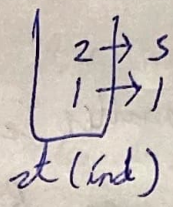
So maxA = 1-0 * 3 = 3

put ind=1 in stack



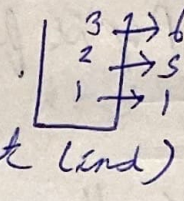
At ind = 2, ele = 5

Compare with ele in stack, it is greater, so put it in stack



At ind = 3, ele = 6

Compare with ele in stack, it is greater, so put in stack



Inc. order

At ind = 4, ele = 2

which is smaller than top of stack, in that case

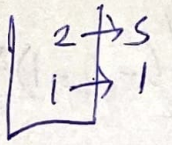
take the ~~pre~~ index 3, and say its rightSmall = ~~6~~ 4

& leftSmall is the element below top which is ind = 2

So, max area = 4 - 2 - 1 = 1 × 6 = 6

Remove 6,

compare a[ind] with top of stack again



top is greater, which means for ind = 2, its rightSmaller = 4

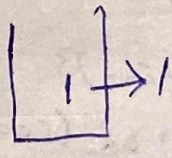
& leftSmaller = 1

RS LS

max A = 4 - ~~1~~ - 1 * height =

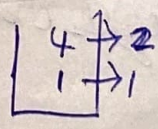
= 2 * 5 = 10

Pop it from stack



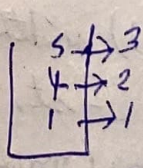
Now, compare it again with ind = 4,

top is lesser, so push in stack



At ind = 5, ele = 3

top is lesser, so push in stack



Now, we will do one more iteration with ind = 6

We will take top of stack,

[5] ⑤ its right smaller is 6 itself

& left smaller is the element below top.

so, RS = 6 & LS = 4

area = 6 - 4 - 1 = 1 * 3 = 3

Pop it from stack

[1 4 → 1, 2 → 1]

Its right small is also 6 & left small = 1

area = 6 - 1 - 1 * 2 = 8

Pop it from stack

Its. [1 → 1]

Its right small is also 6 & left small is 0

(6 - 0 - 1) * 1 = 5

T.C - O(N) + O(N)

S.C - O(N)

Code →

```
stack<int> st;
int maxA = 0;
for(i = 0; i <= n; i++){
while(!st.empty() && (i == n || heights[st.top()] > heights[i])){
    int h = heights[st.top()];
    st.top();
    int width;
    if(st.empty()) width = i;
    else width = i - st.top() - 1;
    maxArea = max(maxArea, width * h);
}
st.push(i);
}
```

Sliding Window Maximum

array of integers → nums sliding window of size → k

Which is moving from the very left of the array to the very right. You can see only k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

E.g:-

[1 3 -1 3 5 3 6 7] k=3

Window	Max
[1 3 -1]	3
[3 -1 3]	3
[-1 3 5]	5
[3 5 3]	5
[5 3 6]	6
[3 6 7]	7

O/P → [3, 3, 5, 5, 6, 7]

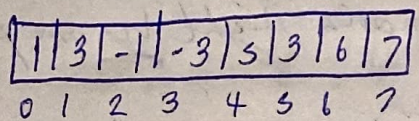
Brute force -

Using 2 loop :

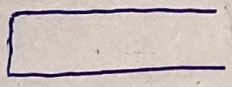
```
for(i = 0 to n-k+1)
  { max = a[i]
    for(j = i → i+k-1)
      {
        max = max(a[j], max)
      }
  }
```

TC- O(N*K)

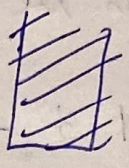
Optimal- Here also we will use the concept of nge
i.e. next greater element,
But here we will store elements in decreasing order
In order to do so, we will use deque



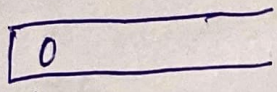
k = 3


deque

(We are storing index in deque)

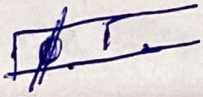


At ind = 0, deque is empty, so push it in deque



deque

At ind = 1, val = 3, as we have got 3, we will remove 0th index from deque & push 1 index, because we need to store max element. So storing 1 is useless, so pop it

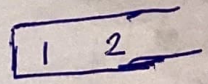

deque

Pop

At ind = 2, val = -1

Now, val < element in deque

so, push it in deque

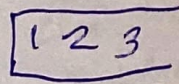


deque

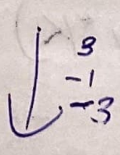
Value is in decreasing

At ind = 3, val = -3 which is less than -1

so, push it in deque

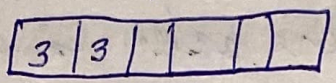


deque



decreasing order, so maintained

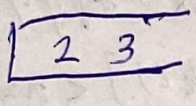
Now, everytime we reach k=3, we will keep storing max by looking at the front of the queue.

So, till now maxAns 

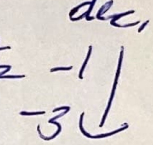
Now, ind=4, val=5,

if we insert it in deque, it will disturb the decreasing order of our deque.

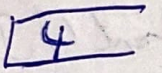
Also, in the boundary
2 to 4, index 1 should not be stored in queue so we can pop it



this is our deque with elements

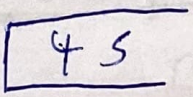


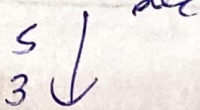
Now, to not disturb the decreasing order, we will pop ind 2 & 3 from deque at push ind=4

deque 

store in maxAns → 3 3 5

Now, ind=5, val=3



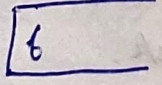


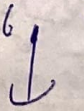
decreasing order is maintained

Store in maxAns → 3 3 5 5

Now, ind=6, val=6

If we insert it decreasing order will not be maintained, so pop elements from deque then insert

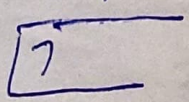


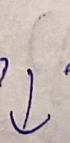


maxAns → 3 3 5 5 6

Now, ind=7, val=7

Pop ind=6, then push 7





maxAns → 3 3 5 5 6 7

Intuition- We keep storing max elements in the beginning of the deque, and maintain decreasing order in the deque. If anywhere order is disturbed by a new element, we will pop previous elements in deque.

TC → O(N) + O(N) ≈ O(N)

SC → O(K)

Code →

```
for (i=0 → n-1) {
if(!dq.empty() && dq.front() == i-k) dq.popfront();
while(!dq.empty() && nums[dq.back()] < nums[i]){
    dq.pop_back();
}
dq.push_back(i);
if(i >= k-1) ans.push_back(nums[dq.front()]);
}
return ans;
```

Min Stack

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

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

TC - O(1)

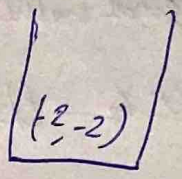
Approach 1 - We will store pairs in the stack

where (val, min)

val represents current value & min represents minimum element till now

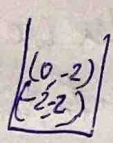
Push(-2)

Stack is empty so directly push



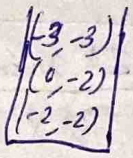
Push(0)

Push 0, but min remains same



Push(-3)

Push -3, min gets updated



getMin() → return top().second

pop() - pop out top

getMin() return top().second

T(-O(1)

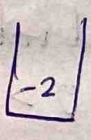
S(-O(2N)

Approach 2 We will not store pair in stack

Maintain a min variable = INT_MAX

Push(-2) → Stack is empty, push -2

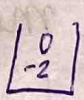
min = -2



Push(0) → 0 > -2, don't do anything

push 0

min = -2

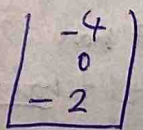


Push(-3) → -3 < -2, so now instead of pushing original value, we will push (2 X val - prevmin) *

i.e. (2X-3 - (-2)) = -4

push -4 into stack

min = -3 now



getMin() → return mini element
which is -3

top() → -4 is not original value, this is modified value, so to retrieve original value we will check if -4 < mini i.e. -4 < -3

-4
0
-2

Yes, it is then we will simply return -3 as top

pop() → Now, when we pop, mini should go back to -2
In order to that, ~~xxx~~ update mini

$$\text{mini} = (2 \times \text{mini}) - (\text{st.top}())$$
$$= 2 \times -3 - (-4)$$
$$= -6 + 4$$
$$= -2$$

mini returns to -2
We will have to update mini before popping

Before doing this we will check if -4 i.e. modified value < mini, then yes it confirm that it is a modified value.

top() → 0, we know this is not modified value, because
0 > -2 i.e. mini

0
-2

Intuition - Why is modified value always < mini ?

We know the value we were trying to insert before modifying was -3, which was < -2

So, we are inserting 2X val - mini

val < mini

val - mini < 0 (Add val on both sides)

val + val - mini < 0 + val

Modified value ← (2val - mini) < val

∴ Modified value < mini

Intuition → Why does mini = 2 × mini − st.top() works?

We inserted (2 × val − mini) in stack

So, mini = 2 × mini − st.top() can be written as

= 2 × mini − (2 × val − prevmini)

We know that val == mini

= 2 × val − 2 × val + prevmini

= prevmini

∴ This reverts back to previous min

Rotten Oranges given grid

0 represents empty cell

1 represents fresh orange

2 represents rotten orange

Every minute, any fresh orange that is 4 directionally adjacent to a rotten orange becomes rotten.

Return the min no. of minutes that must elapse until no cell has a fresh orange.

If it is impossible return −1.

Eg- grid = $\begin{bmatrix} [2,1,1], \\ [1,1,0], \\ [0,1,1] \end{bmatrix}$ o/p → 4

1 minute $\begin{bmatrix} 2,2,1 \\ 2,1,0 \\ 0,1,1 \end{bmatrix}$ 2 minute $\begin{bmatrix} 2,2,2 \\ 2,2,0 \\ 0,1,1 \end{bmatrix}$ 3 minute $\begin{bmatrix} 2,2,2 \\ 2,2,0 \\ 0,2,1 \end{bmatrix}$ 4 minute $\begin{bmatrix} 2,2,2 \\ 2,2,0 \\ 0,2,2 \end{bmatrix}$

$E_2 \rightarrow \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$

↑
This orange can never become rotten

So, o/p → -1

Approach-

We will use a BFS technique

1) We will start counting, what is the no. of oranges we are currently having

$\begin{bmatrix} 2 & 0 & 0 & 2 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ cnt = 8

4×4

2) We will take a queue which will store (row, col) no.

Initially store the rotten oranges index in the queue.

cnt = 8

In first iteration, we perform BFS on elements in queue.

* Try to go in all 4 directions

x ← (0,0) → x empty

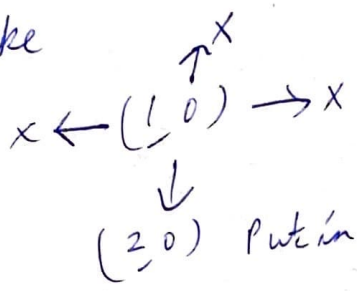
↓
(1,0) put it in queue

empty x ← (0,3) → x

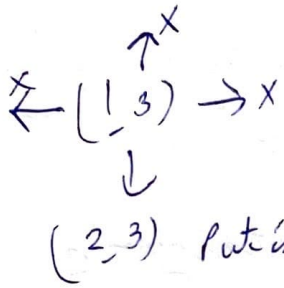
↓
(1,3) Put it in queue

minutes = minutes + 1;

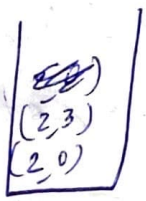
Now take



(2,0) Put in queue

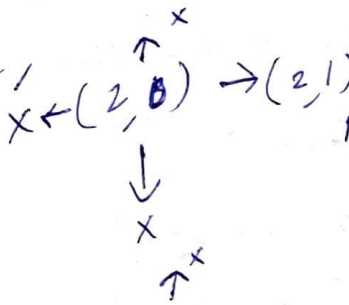


(2,3) Put in queue

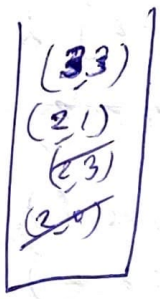


if (!q.empty) minutes ++;

Now take

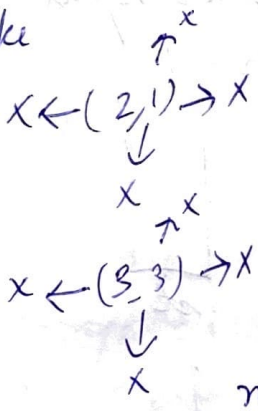


(3,3) Put in q



minutes ++;

Now take



minutes = 3

minutes will remain same

We will also keep a track of no. of elements we inserted in queue. Here, we put 8 oranges in queue, which means we have made all oranges rot.

TC - O(N*N) X4

SC - O(NXN)

Online Stock Span

Collect daily price quotes for some stock and return the span of that stock's price for the current day.

Span of stock's price in 1 day is max no. of consecutive days (starting from that day and going backwards) for which stock price <= curr price

Ex 1: In [7, 2, 1, 2] & price of stock today = 2

Then span = 4, because starting from today total 4 days stock price <= curr price

In [7, 3, 4, 1, 2] & price of today = 8

Then span = 3, because ~~[1, 2]~~ [1, 2] < 8

StockSpanner() - Initialize object of the class

int next(int price) - Returns the span of the stock's price given today's price

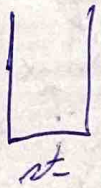
Approach:

~~next(100)~~ We will use a stack & maintain a span variable

Initially stack is empty & span = 1

next(100)

As stack is empty, push 100 & return 1

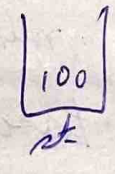


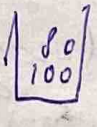
next(80)

We will compare with top of stack, if top < curr price then we will pop & increment span

Here, span will remain 1

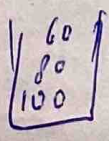
push 80





next(60)

push 60 return 1



next(70)

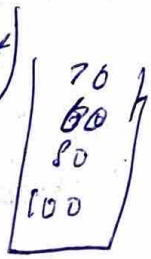
st.top() < curr price (Remember we don't have to pop permanently)

increment span

keep doing this until top < curr

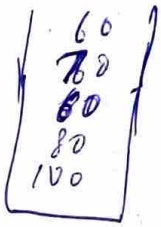
at last span = 2 & push 70

& return span



next(60) reinitialize span = 1,

return span, push 60



next(75) span = 4

This solu will give TLE because when we will keep taking pop ele < curr, then we will pop & store it in some temp stack.

We will have to use another loop to push all the elements again.

TC - O(N)+O(N)

SC - O(N)+O(N)

Optimal -

We will store price, span in stack

E.g next(100) span = 1

next(80) span = 1

compare it with top just like prev approach

next(60) span = 1

next(70) Now here span = 1

st.top() < curr price, So, add st.top() span with curr span. Span becomes 2

Pop (60,1) & Push (70,2)

(70,2) represents there were 2 days at which price <= 70.

(70,2)
(80,1)
(100,1)

next

~~Pop~~ (60,1)

(60,1)
(70,2)
(80,1)
(100,1)

next

~~Pop~~ (75,) span = 1

st.top() < curr.

Pop elements from stack until st.top <= curr & add span

Store (75,4) in stack

SC - O(2N)

TC - O(N)

Celebrity Problem

Given N, which represents N people in a party

Each person has been assigned a unique id b/w 0 to N-1

Identify the celebrity.

A celebrity is a person whom everyone knows

But celebrity doesn't know everyone

Note: There is a knows(A,B) func which tells whether A knows B, returns true/false.

A NXN matrix is given which tells whom knows whom

Eg:- N=3 M[][] = {{0,1,0},
{0,0,0},
{0,1,0}}

0 || 1 → 1

Explanation: 0 & 2nd person both knows 1 ∴ 1 is celebrity

Brute Force Approach:- As we can see celebrity knows no one, so celebrity row will have all 0's

i.e. 0 0 0

And all celebrity column will have all 1's except its same row, because (1,1) = 0.

```
for( i=0 to N-1)
{
    [Row loop]  O(N)
  + [Col loop]  O(N)
}
```

T.C. - O(N²)
S.C. - O(1)

Optimal Approach:- We will use a stack

Stack will have celebrity element in the end.

Algo: 1) Put all elements in stack

2) While (st.size() != 1)

A → st.top() → st.pop()
B → st.top() → st.pop()

if (A knows B)

true, then A is not a celebrity for sure, we will discard A & push B

Or → if (B knows A)

B discard, A → push in stack

We will keep doing this, i.e. we will take 2 elements whether either of them know each other, then we will pop accordingly. At last, we will be only left with celebrity.

But, we still cannot say it is a celebrity.

So, to confirm, we will check its row & col

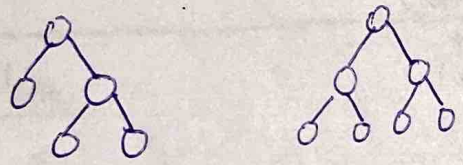
TC - O(N)

SC - O(N)

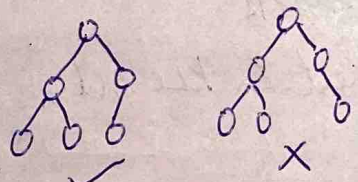
Binary Trees

Types of Binary Trees →

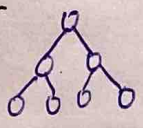
1) **Full Binary Tree** - either has 0 or 2 children



2) **Complete Binary Tree** - All levels are completely filled except the last level.
The last level has all nodes as left as possible



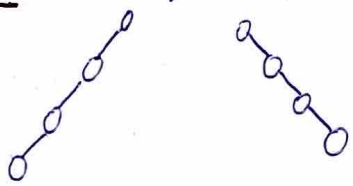
3) **Perfect Binary Tree** - All leaf nodes are at same level



4) **Balanced Binary Tree** - Height of tree at max $\log N$ where N is no. of nodes

$n = 8$, $\log_2 N = 3$

5) Degenerate Trees - If n = 4, then these trees will be like

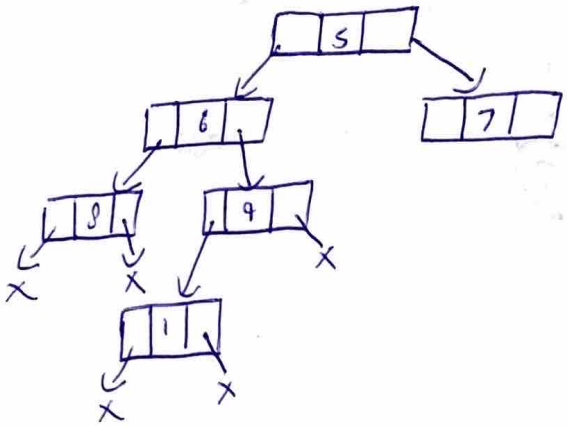


Every node has a single children

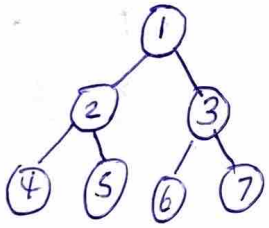
(Skewed trees)

Binary Tree Representation

We will create a struct Node with val, left & right ptr



Traversal Techniques (BFS|DFS)



→ Inorder Traversal (Left Root Right)

4 2 5 1 6 3 7

→ Pre-Order Traversal (Root Left Right)

1 2 4 5 3 6 7

→ Post-Order Traversal (Left Right Root)

4 5 2 6 7 3 1

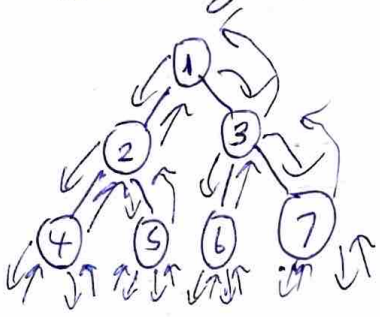
→ BFS → Goes Level wise

1 2 3 4 5 6 7

Pre-order Traversal →

```
void preorder(node){
  if(node == NULL) return;
  print(node->data)
  preorder(node->left)
  preorder(node->right)
}
```

T <- O(N)
S <- O(N) ↑ worst case



1 2 4 5 3 6 7

Inorder Traversal →

```
void inorder(node){
  if(node == NULL) return;
  inorder(node->left)
  print(node->data)
  inorder(node->right)
}
```

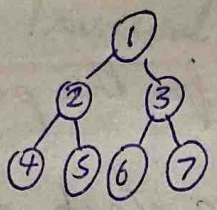
T <- O(N)
S <- O(N) ↑ worst case

Postorder Traversal →

```
void postorder(node){
  if(node == NULL) return;
  postorder(node->left)
  postorder(node->right)
  print(node->data)
}
```

T <- O(N)
S <- O(N) ↑ worst case

Level Order Traversal



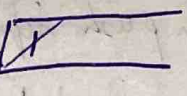
~~1 2 3 4 5~~ 1 / 2 3 / 4 5 6 7

We need a queue data structure, which will be initially having the root of the tree.

In order to store the traversal level wise we will be requiring same data structure.

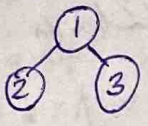
Let's take vector<vector<int>>

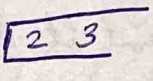
Initially 1 is present in queue



Take this 1 out

Check whether its left / right exists, if yes then put them in queue





Now, we have done it for 1, since there was only 1 element in this level, ~~what~~ we will put this into a vector.

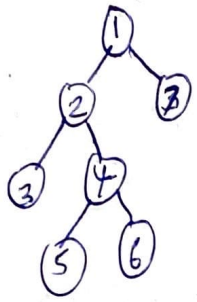
Now, in next iteration pop out elements from the queue & perform same task as above.

Code →

```
q.push(root);
while(!q.empty()){
    int sz=q.size();
    vector<int> level;
    for(i=0;i<sz;i++){
    TreeNode *node=q.front();
    q.pop();
    if(node->left) q.push(node->left);
    if(node->right) q.push(node->right);
    level.push_back(node->val);
}
ans.push_back(level);
}
```

Iterative Preorder Traversal →

We will be using stack for this purpose



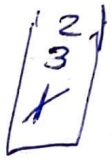
1) We will take a stack, push the root in stack

1

& print it, then

2) Pop out 1, & check for its right & left children.

First insert right children in stack then push left children

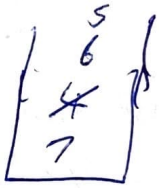


1 2

Now pop out (2), ~~check~~ print it & then push its right & left children

Now pop out (3), print it & insert its children

1 2 3



Then (4), print it & insert children

1 2 3 4

Then (5), print & check children

Then (6), print & check children

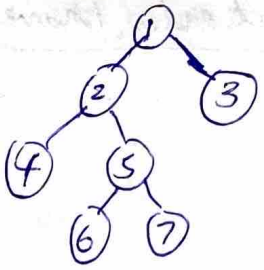
Then (7) " " "

1 2 3 4 5 6 7

TC - O(N)

SC - O(N)

Iterative Inorder Traversal

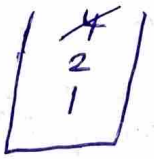


4 2 6 5 7 1 3

In this approach also, we will use stack.

1) Push root into stack

2) Initially traverse in complete left until left != NULL & keep pushing left children in stack.



Now ④ → left ~~& right~~ == NULL,

Now ~~check~~ print ④, now move to 4's right, which is also NULL

Now come back for next iteration

Now go to ②'s right i.e. 5

Now move to 5's left which is 6, push in stack

Move to 6's left, which is NULL

if (node == NULL), then take top of stack, print it and go to top's right, 6 → right is also null

~~Go to pop to Now take & because~~

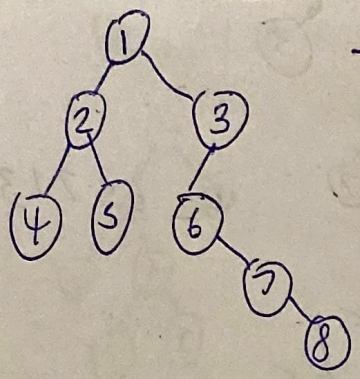
Now also if (node == NULL) then take top of stack, print it & go for its children

So, whenever we see node == NULL, we print st.top()

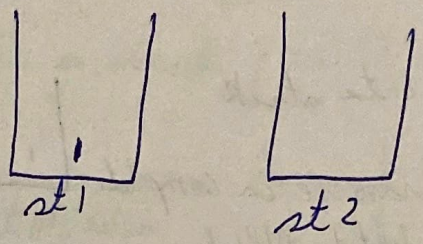
```
Treenode * node = root;
while (true) {
  if (node != NULL) {
    st.push(node);
    node = node -> left;
  }
  else {
    if (st.empty()) break;
    node = st.top();
    st.pop();
    ans.push_back(node->val);
    node = node -> right;
  }
}
return ans;
```

Iterative Postorder Traversal using 2 stack

TC- O(2N)
SC- O(N)

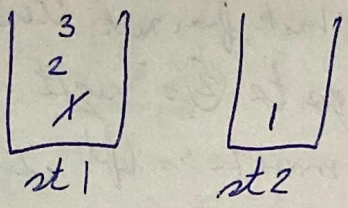


→ 4 5 2 8 7 6 3 1

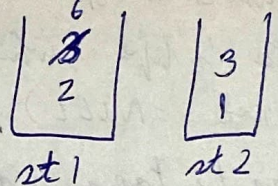


1) Take the root of tree and insert in st1.

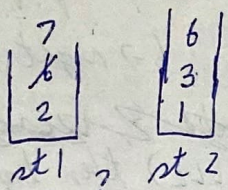
2) Take st1.top() out and insert in st2.
And if top has left & right then insert it in st1, ✗
Insert left first then right.



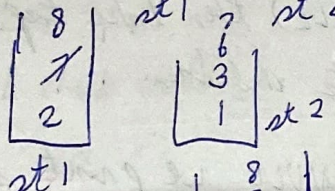
3) Take st1.top() put it in st2, push left & right in st1



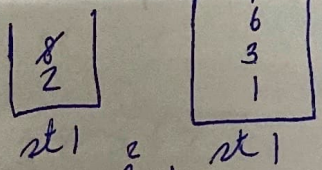
4) Now 6,



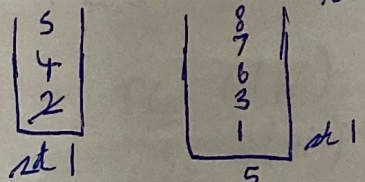
5) Now 7,



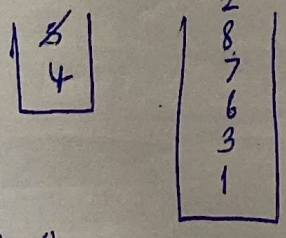
6) Now 8,



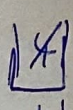
7) Now 2,

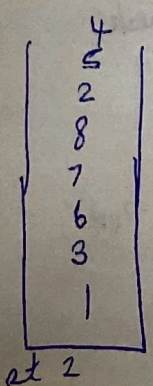


8) Now 5,



9) Now 4,





✗ st2 stores the postorder traversal