Striver’s SDE Sheet With It’s Short Algo and Complexities

In this valuable contribution, we present a meticulously curated PDF file designed to assist college students aspiring to excel as software developers. Our comprehensive guide includes a compilation of questions, problem-solving approaches, concise algorithms, and the associated time and space complexities, all based on Striver's SDE sheet. By following this guide, students can efficiently prepare for interviews within a short time frame of 3 to 4 hours. The questions presented have been handpicked by Striver bhaiya, making them particularly significant and beneficial for interview preparation.

Structure of the Study Guide:

1) Question Name: Each question is clearly labeled, allowing easy navigation and reference.
2) Approach to Solve: Step-by-step guidance is provided, offering a systematic and structured methodology to tackle each problem.
3) Small Algorithm: Succinct algorithms are included to present a concise yet comprehensive solution to each question.
4) Time & Space Complexities: The complexities associated with each solution are outlined, giving students a holistic understanding of performance considerations.

Note: Please ensure to credit the original source and author, Raj Vikramaditya (striver Bhaiya) , for the questions included in the study guide.

1. Set Matrix Zeroes.

Approach 1:
Algo:
Make two dummy arrays a and b with size of m and n respectively.
If matrix[I] [j] == 0
a[I] =0
b[j] = 0
Now traverse again this matrix
ifa[i] == 0 || b[j] == 0
matrix[I] [j] = 0
Time complexity = (m*n) + (m*n)
Approach 2:
Use first row and first column of matrix as dummy arrays.
Algo:

```
Bool condition = true
for(int I to m){
        if(matrix[I] [0] == 0) condition = false
                for(int j=1 to n){
                if(matrix[I][j] == 0){
                        matrix[I][0] = [0][j] = 0
}
For(I=m-1 to 0){
        for(j =n-1 to 1){
                if(matrix[I] [0] ==0 || matrix[0] [j]==0) matrix[I][j]=0
        }
if(condition is false) matrix[I][0] = 0;
}
```

Time complexity = 2*(m*n)

2. Pascal's Triangle

Algo:

```
vector<vector<int>> ans;
       for(int i=0;i<numRows;i++){
          ans.push_back(vector<int>(i+1,1));
          for(int j=1;j<i;j++){
             ans[i][j] = ans[i-1][j-1] + ans[i-1][j];
          }
       }
```

Time complexity is O(n^2).

3. Next Permutation

Approach: Remember that according to problem statement next permutation will always sorted in decreasing order.here logic is that, first find the inflation

point which means number which is greater then its previous element(break the sorted order of decreasing order) then find the next bigger number from that element and swap it then after inflation point sort the remaining elements in increasing order.

Algo:

```
Int I = nums,size() - 2;
Int b_p, n_g;
while(I >= 0){
if(nums[I] < nums[i-1])
b_p = I; break;}
I - -; }
if(I <0){
reverse(nums.begin(), nums.end());
Return; }
I = nums.size() - 1;
while(I >= b_p-1){
if(nums[b_p] < nums[I]) n_g = I; break;
}
I - -;
}
swap(nums[b_p], nums[n_g]);
reverse(nums.begin() + b_p+1 , nums.end();
```

Time complexity is O(n).

6) Best Time to Buy and Sell Stock

```
Int mx = 0, mn = INT_MAX;
for( I to n){
Mn = min(mn, prices[I])
Mx = max(mx, prices[I] - mn);
}
Return Mx;
```

Time complexity is O (N)

48) Rotate Image

Approach: first transpose the matrix and then swap all the element like first with last, second with second last like wise with each inner array of matrix.

Algo:

```
Void rotate(vector<vector<int>> & matrix){
Int n = matrix.size();
// first transpose the matrix.
for(int I=0;i<n;i++){
        for(int j=I;j<n;j++){
        swap(matrix[I][j], matrix[j][I]);
} }
```

```
// now swap all the element in inner array of matrix
for(int I=0;i<;I++){
Int left = 0, right =n-1;
while(left < right){
swap(matrix[I][left], matrix[I][right]);
left++; right - -;
} }
```

Time complexity is O(n^2)

56) Merge Intervals

Approach : Two pointers and stack
Algo:
First sort the matrix.
Take start = intervals[0][0], end = intervals[0][1];
Take 2d vector ans.
Check this three conditions:

```
 for(auto it: intervals){
        if(it[0] > end && it[0]>start){
            ans.push_back({start,end});
            start = it[0];
            end = it[1];
        }
        else if(it[0]<end || it[0]==end){
            if(it[1]>end){
            end = it[1];
            }
        } ans.push_back({start, end});
```

Time complexity in both approaches is O(n log n).

88. Merge Sorted Arrays

Approach: Use map
Algo;
First store the all the frequencies in map.
Clear the nums1 vector.
Iterate map and while(mp.second —) store all the mp.first in nums 1.
Time complexity O ((m + n) log(m + n)).

287. Find the Duplicate Numbers.

Approach: Use the map
Algo:

```
Map<int,int>mp;
for(auto it: nums){
mp[it]++; }
for(auto it: mp){
if(it.second>1) return it.first;
}
```

Time complexity is : O (n log n).

74. Search a 2D Matrix:

Approach: Binary Search
Algo:

```
Int n = matrix.size(), m = matrix[0].size();
If ( m == 0) return false;
Int low = 0, high = (m * n ) -1;
while( low <= high){
Int mid = (low + high) /2;
if(matrix[mid/m] [mid%m] == target) return true;
if(matrix[mid/m] [mid%m] > target) high = mid-1;
Else{
Low = mid+1; }
}
Return false;
```

Time complexity is: O (log (m * n))

50. Pow(x,n)

```
Double res = x;
Long long p =abs(n);
while(p){
if(p & 1) res = res*x;
X *= x;
P = p/2;}
ir(n<0) return (double)1.0/res;
Return res;
```

Time complexity is: O(log n)

169. Majority Element

Approach: Use the Map
Algo:

```
int factor = nums.size()/2;
unordered_map<in, int>
for(int I=0;i<nums.size();I++){
mp[nums[I]]++;
if(mp[nums[I]] > factor) return nums[I];
}
Return -1;
```

Time complexity is : O(n).

229. Majority Element II.

Approach: Map
Algo:

```
Vector<int> ans;
unordered_map<int, int> mp;
for(int I=0;i<nums.size();I++){
mp[nums[I]]++ }
for(auto it: mp){
if(it.second > nums.isze()>3){
ans.push_back(it.first);
}
Return ans;
```

Time complexity is : O(n)

62. Unique Paths

Approach 1: First approach is that use dynamic programming. We know that 2 paths are from starting point to go the right until you reach the m-1 and then go down side. Similar to opposite first go the down side until the n-1 and then go right side until you reach the destination. Now go in the matrix from I =1 to m and j = 1 to n and then the fill the matrix matrix[I] [j] = matrix[I] [j-1] + matrix[I-1] [j] then return the matrix[n-1][m-1];

Algo:

```
Vector<vector<int>> ans (n,m);
for(int I=0;i<n;i++){
ans[0][I] = 1; }
for(int i=1;i<m;i++){
ans[I] [0] = 1; }
for(int I=1;i<n;i++){
        for(int j=1;j<m;j++){
        ans[I] [j] = ans[I-1][j] + ans[I] [j-1];
} }
Return ans[n-1] [m-1];
```

Time Complexity is: O (m * n)

Approach 2: Use the Ncr formula. Use any of them where N = n+m-2; r is either n-1 or m-1;

```
Int N = n+m-2, r = n-1;
Long long res = 1;
for(int I=1; I<=r ;I++){
Res = res * ( N - r + I ) /I;
}
Return (int) res;
```

Time complexity is O (n)

493. Reverse Pairs

Approach: Merge sort Approach.

first divide whole array like merge sort and now when you will merge the array compare the given condition and increment the count according to that

condition.
Algo: https://leetcode.com/problems/reverse-pairs/submissions/911568268/
Time complexity is: O (n log n) because merge sort's time complexity is also o (n log n).

1. Two Sum

Approach: first is brute force with time complexity of O (n ^ 2). And second approach is use map with time complexity of O (n).
Algo:

```
unordered_map<int, int> mp;
vectoir<int> ans;
for(int I=0 to nums.size()){
if(mp.find(target - nums[I]) != mp.end()){
Ans.push_back(mp[target-nums[I]);
Ans.push_back(I);
}
mp[nums[I]] = I;
}
Return ans;
```

Time complexity is O (n).

18. 4 Sum

Approach : Use 2 for loop and 2 pointer approach.
Algo:

```
sort(nums.begin(), nums.end());
Int n = nums.size(); vector<vector<int>> ans;
for( I to n){
        for(j =I+1 to n){
                long long temp = (long)target - nums[j] - nums[I];
                int left = j+1, right = n-1;
                while(left < right){
                int sum = nums[left] + nums[right];
                if(temp == sum){
                ans.push_back({nums[I],nums[j],nums[left],nums[right]});
                        while(left<right && nums[left] == nums[left+1]) left++;
                        while(left<right && nums[right] == nums[right-1]) right- -;
                }
                else if(sum < temp) left++;
                else right - -;
while(I > n-1 && nums[I] == nums[I+1]) I++; }
while(j > n-1 && nums[j] == nums[j+1]) j++; }
Return ans;
```

Time complexity is O (n ^ 3)

108. Longest Consecutive Sequence

Approach: use Hashmap
Algo:

```
set<int> st;
for(auto it: nums) st.insert(it);
Int streak = 0;
for(auto it: nums){
if( ! St.cont(it-1)){
Int curr_streak = 1, curr_num = it+1;
while(st.count(curr_num+1){
curr_num++, curr_streak ++; }
Streak = max(streak, curr_streak) }
Return streak;
```

Time complexity is O (n).

(GFG) Largest subarray with 0 sum.

Approach : Use the map and store the prefix sum as key and index as values. Then find the sum in map if sum found in map and that index is bigger then previous index then ans = max(ans, i-mp[sum]).
Algo:

```
unordered_map<int, int> mp;
Int prefix = 0, ans =0;
for(int I=0;i<n;i++){
Prefix += A[I];
if(mp.find(prefix) != mp.end()){
Ans = max(ans, i-mp[prefix]) ; }
Else mp[prefix] = I;
}
Return ans;
```

Time complexity is O (n).

(Interview Bit) Subarray with given XOR.

Approach : Use Unordered map to solve this problem.
Algo:

```
Int ans = 0, x=0;
unordered_map<int, int> mp;
for(int I=0; I<A.size();I++){
x = x ^ A[I];
if( x == B) ans++;
if(mp.find( x ^ B) != mp.end()){
Ans += mp[x ^ B];
}
mp[x ^ B]++;
}
```

Time complexity is O (n). May be it will O (n log n) because in worst case unordered map will takes the O(n) for search a variable in map.

3. Longest Substring Without Repeating Characters

Approach: Use the Set

Let's Understand with iterating one string called "pwwkew:
Initialize the set st, the variable ans to 0, and the variable start to 0.
Loop through each character in the string:
For the first character 'p', it is not present in the set. So, 'p' is added to the set. The current substring is "p" with a length of 1. Update ans to 1.
For the second character 'w', it is not present in the set. So, 'w' is added to the set. The current substring is "pw" with a length of 2. Update ans to 2.
For the third character 'w', it is present in the set. This indicates the start of a repeated substring. We need to update the start variable and remove the characters between the current start and the position of the repeated character ('w'). In this case, we iterate from start to the position of 'w' ('p' in this case), erase 'p' from the set, and increment start to the next index. The current substring is now "w" with a length of 1.
For the fourth character 'k', it is not present in the set. So, 'k' is added to the set. The current substring is "wk" with a length of 2. Update ans to 2.
For the fifth character 'e', it is not present in the set. So, 'e' is added to the set. The current substring is "wke" with a length of 3. Update ans to 3.
For the sixth character 'w', it is present in the set. Again, this indicates the start of a repeated substring. We update the start variable and remove the characters between start and the position of the repeated character ('w'). In this case, we iterate from start to the position of 'w' ('k' in this case), erase 'k' from the set, and increment start to the next index. The current substring is now "ew" with a length of 2.

Algo:

```
Int n = s.size(), ans=0, start=0;
Set<char> st;
for(int I=start; I<n; I++){
if(st.find(s[i]) != st.end()){
Ans = max(ans, i-start);
while(s[start] != s[I]){
St.erase(st[start);
s++;}
s++;}
St.insert(s[I]);
}
Return max(ans, n-start);
```

Time complexity is O (n log n)

206. Reverse Linked List

Approach: for the given example: 1->2->3->4->5 we just need to change the direction of arrows. So for this we use another linked list for store the reverse

Algo:

```
ListNode* reverseList(ListNode* head) {
ListNode* dummy = NULL;
while(head != NULL){
ListNode* next = head->next;
```

```
Head->next = dummy;
Dummy = head;
Head = next;
}
Return dummy;
```

Time complexity is O (n)

876. Middle Of the Linked List:

Approach : take two pointer and iterate over linked list increment first pointer by 1 ans second pointer by 2 if 2nd pointer reaches the last node of list then return first pointer.

Algo:

```
ListNode* middleNode(ListNode* head) {
        ListNode *slow = head, *fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
```

Time Complexity is O (n).

21. Merge two sorted Lists.

```
 ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
ListNode* ans = NULL;
         if(list1 == NULL) return list2;
         if(list2 == NULL) return list1;
if(list1->val <l ist2->val) {
Ans = list1;
Ans->next = mergeTwoLists(list1->next, list2); }
Else {
Ans = list2;
Ans->next = mergeTwoLists(list1, list2->next);
}
Return ans;
```

Time complexity is : O (m + n)).

19. Remove Nth Node From End of List

Approach : take another dummy empty linked list, then take two ListNode pointer slow and fast as pointing to dummy ListNode, now iterate the fast pointer from 0 to n. Now take another while loop and iterate until fast-next become NULL ans take slow = slow-next as well. Now make slow->next = slow->next->next. Finally return the dummy.

Algo:

```
*ListNode dummy = new ListNode();
*ListNode fast = dummy;
```

```
*ListNode slow = dummy;
for(int I = 0;i<n; I++) fast = fast->next;
while(fast->next != NULL){
Fast = fast->next;
Slow = slow->next; }
Slow->next = slow->next->next;
Return dummy;
```

Time complexity is O (n).

2. Add Two Numbers

Algo:

```
ListNode *dummy = new ListNode();
ListNode *l = l1, *r = l2, *curr = dummy;
Int carry = 0;
while(l != NULL || r!= NULL){
Int x = 0, y= 0;
if(l != NULL) x = l->val;
if(r != NULL) y = r->val;
Int sum = carry + x + y;
Carry = sum / 10;
Curr->next = new ListNode(sum % 10);
Curr curr->next;
if( l != NULL) l = l->next;
if(r != NULL) r = r->next;
if(carry > 0){
Curr->next = new ListNode(carry);
} }
Return dummy->next;
}
```

Time complexity is O(Max (n , m)). Where n is length of list1 and m is length of list2.

237. Delete Node In a Linked List

Approach: it is a very tricky question answer of question is in the question itself. It is clearly given in the question is that the node value is not at the last of the linked list.

So we can take the node->next value in the dummy node and then put the dummy-> val into the node->val and dummy->next into the node->next.

Algo:

```
Void deletenode(ListNode* node){
ListNode *dummy = node->next;
Node->val = dummy->val;
Node->next = dummy->next;
}
```

160. Intersection of Two Linked List.

As per the problem we can find to same address of a node not the value of tat node. And the one possible solution is first make the two linked lists same size before they are intersected.

Algo:

```
Int l1 = 0, l2 = 0;
ListNode *dummy = headA;
while(dummy) l1++;
while(dummy) l1++; dummy = dummy->next;
Dummy = headB;
while(dummy){
l2++;
Dummy = dummy->next;
Int diff = abs(l1 - l2).
if(l1 < l2){
while(diff — ){
headB = headB->next; } }
Else{
while(diff —){
headA = headA->next;
} }
while(headA && headB){
if(headA == headB) return headA;
headA = headA->next;
headB = headB->next;
}
Return NULL;
```

Time complexity is O(m+n).

141. Linked List Cycle

Approach: Use the set and iterate the linked list if head is found in set then return true otherwise return false;

Algo:

```
unordered_set<ListNode*> visited;
while(head != NULL){
if(visited.count(head) > 0) return true;
Visited.insert(head);
Head = head->next;
}
Return false;
}
```

Time complexity is O(n). Space complexity is also O (n).

*Note that this solution will use extra o(n) space. So for that we use another approach called two pointer.

In this take slow pointer starting from head and fast pointer starting from head->next; now iterate over linked list and make 1 jump for slow pointer ans 2 jump for fast pointer at the same time. In this while loop if fast == null or fast->next == null return false otherwise at the end return true;

Algo:

```
if(head == NULL || head->next == NULL) return false;
Listnode *slow = head;
ListNode *fast = head->next;
while(slow != fast){
if(fast == NULL || fast->next == NULL) return false;
Slow = slow->next;
Fast = fast->next->next;
}
Return true;
```

Time complexity is O (n)

25. Reverse Nodes In K-group.

Approach : Use the Naive approach create a dummy node with value 0 and use the three pointer approach for the reverse the k group.

Algo:

```
if(head == NULL || k <=1) return head;
ListNode *Dummy = new ListNode(0);
Dummy->next = head;
ListNode *prev= dummy, nex = dummy, curr = dummy;
Int count = 0;
while(cure != NULL){
count++;
curr = curr->next
}
while(count > k){
Curr = prev->next;
nex = curr->next;
for(int I=1; I<k; I++){
Curr->next = nex->next;
Nex->next = prev->next;
Prev->next = nex;
nex = curr->next;
}
pre  = curr;
Count -= k;
}
Return dummy->next;
```

Time Complexity is O (n)

234. Palindrome Linked List

Approach: first find the middle element of linked list and then from middle to last element reverse the list. And then traverse the list from first and middle and then compare the list. if not equal then return false otherwise at last return true.

Algo:

```
if(head->next == NULL) return true;
// find the middle element.
ListNode *slow = head;
ListNode *fast = head;
while(fast ==NULL && fast->next==NULL){
Slow = slow->next;
Fast - fast->next->next;
}
ListNode *curr =slow, *prev = NULL, *nex = NULL;
while(curr !-= NULL){
Nex = curr->next;
Curr->nex = prev;
Prev = curr;
Curr->nex;
}
ListNode *temp = head;
ListNode *temp1 = prev;
while(temp != NULL && temp1 != NULL){
if(temp->val != temp1->val) return false;
Temp = temp->next;
Temp1 = temp1->next;
}
Return true;
```

Time complexity is O (n).

142. Linked List Cycle II

Approach: for solve this question you have to use the 2 pointer approach which can we used for find the cycle in the linked list. In that approach you have to move the slow pointer by 1 ans fast pointer by 2 and in that loop you also check the condition for the cycle if fast == slow then cycle is 1. Then you can assign the fast pointer to the head and iterate the slow pointer until the fast and then return any pointer from slow and fast.

Algo:

```
ListNode *slow = head;
ListNode *fast = head;
Int cycle = 0;
while(fast && fast->next){
```

```
Slow = slow->next;
Fast = fast->next->next;
if(slow == fast) cycle = 1; break;
}
if(cycle == 0) return NULL;
Fast = head;
while(slow == fast){
Slow = slow->next;
Fats = fast->next;
}
Return slow;
```

Time Complexity O (n).

Flattening a Linked List (GFG):

Approach: Use the Recursion and Mergesort approach:
Algo:

```
Node *solve(Node *a, Node *b){
Node *temp = new Node(0);
Node *res = temp;
while(a != NULL && b != NULL){
if(a->data < b->data){
Temp->bottom = a;
a = a->bottom;
Temp = temp->bottom;
}
Else{
Temp->bottom = b;
b = b->bottom;
Temp = temp->bottom;
}
if(a) temp->bottom = a;
Else temp->bottom = b;
}
Return res;
}
//main function;
Node *flatten(Node *root){
if(root == NULL || root->next == NULL) return root;
Root->next = flatten(root->next);
Root = solve(root, root->next);
Return root;
}
```

Time complexity is O (n).

61. Rotate List

Approach: we connect the last node to the first node to create a circular linked list. After that, we traverse the linked list from the head node and move a pointer to the node that will become the new head of the rotated linked list. Finally, we break the circular linked list and return the new head node.

Algo:

```
if( head == NULL) return NULL;
if(k == 0 || head->next == NULL) return head;
ListNode *curr = head;
Int len = 0;
while(curr->next != NULL){
Len ++;
Curr = curr->next;
}
Len ++;
Curr = curr->next;
Int rotate_pts = len - k % len;
Curr = head;
while(rotate_pts - - > 1){
Curr = curr->next;
}
Head = curr->next;
Curr->next = NULL;
Return head;
```

Time complexity O (n)

138. Copy List with Random Pointer

Approach: use hash map approach with o(n) extra space.

First make the new linked list and then iterate the given list and then make the copy of original list. Again iterate the original list and find all the nodes with random values and give random node on the random values of copied list.

Algo:

```
Node *copy = new Node(0);
Node *curr = head;
Node *ans = copy;
Unordered_map<Node*, Node*> mp;
while(curr){
        Node *temp = new Node(curr->val);
        mp.insert({curr, temp);
        ans->next = temp;
        ans = ans->next;
        curr = curr->next;
}
Ans = copy->next;
```

```
Curr = head;
while(curr){
Node *ra = curt->random;
Node *temp1 = mp[ra];
Ans->random = temp1;
Ans = ans->next;
Curr = curr->next;
}
```
Return copy->next;
Time complexity is O(n).

15. 3 Sum

Approach: Binary search
First sort the array and then iterate the nums array then apply binary search on it if condition was satisfy then insert it into set. And at the last convert set into vector.
Algo:

```
Set<vector<int> st;
Vector<vector<int>> ans;
sort(nums.begin(),nums.end());
for(int I=0;i<nums.size();I++){
Int j = I+1, n = nums.size()-1;
while(j < n){
if(nums[I] + nums[j] + nums[n] == 0){
Vector<int> v = {nums[I],nums[j],nums[n]};
St.insert(v);
j++; n- -;
}
Else if(nums[I] + nums[j] + nums[n] < 0) j++;
Else n- -;
}
for(Auto it : st) ans.push_back(it);
Return ans;
```
Time complexity is O (n ^ 2)

42. Trapping a Rain Water

Approach: Use the left right means two pointer approach. Here question is how we make sure that at position I what is the maximum at right or left?. For example we current at pos 3 and that value is 0 and we know that out left max is 2. So here we putted condition of if(height[left] <= height[right]. We iakensured that at right side there is something a right max >= our leftmax otherwise we never reach the position I.
Algo:

```
Int left = 0, right = height.size()-1, ans = 0;
Int leftmax = 0, rightmax = 0;
```

```
while(left <= right){
if(height[left] <= height[right]{
if(height[left] >= leftmax) leftmax = height[left];
Else ans += leftmax - height[left];
Left ++;
}
Else{
if(height[right] >= rightmax) rightmax = height[right];
Else
Ans += rightmax - height[right];
Right - -;
}
Return ans;
```

Time complexity is O (n).

26. Remove Duplicates from Sorted Array

Approach: it is given that our array was sorted. So we can able to use the two pointers approach.

Algo:

```
Int I=0, j=0;
while(j < nums.size()){
if(nums[I] != nums[j]{
I++;
nums[I] = nums[j];
}
j++}
Return I+1;
```

Time complexity is O (n).

485. Max Consecutive Ones

Approach: two pointer to store previous maximum continues 1's and current maximum 1's.

Algo:

```
Int ans = 0, dummy = 0;
for(int I=0;i<nums.size();I++){
if(nums[I] == 1) dummy++;
Else{
Ans = max(Ans,dummy);
Dummy = 0;
}
}
Ans = max(ans, dummy);
Return ans;
```

Time complexity is O (n).

N Meeting In One Room (GFG).

Approach : it is the same problem with activity selection problem in DAA😇.
Algo:
1. First Make the structure which contains three integers start, end, index;
We take index also in structure because it may happen that. There is a two similar activities are there in input.
2. Second step is make the boolean comparator for sort the array according to the end element.
3. Now, sort the array.
4. Make another array called answer and push the very first index of array into that.
5. Take another variable which hold the value of very first ending of meeting.
6. Iterate array from 1 to n:

```
        if(meet[I].start > limit){
        limit = meet[I].end;
        answer.push_back(meet[I].idx);
} }
```

7. Return answer.size()
Time complexity is: O(n * log(n)).

MiniMum Platforms (GFG).

Approach:
Here two arrays are given arr and dep. after carefully observed problem we say that the train are arrival and departure are two independent event so that's why we able to sort those two array independently. After sorting use the two pointer approach one for the arr array and second for dep array.
Algo:

```
sort(arr, arr+n); sort(dep, dep+n);
Int platform_needed = 1, ans = 1, I=1, j=0;
while(I<n && j<n){
if(arr[I] <= dep[j]){
platform_needed ++;
I++;}
Else if(arr[I] >= dep[j]){
platform_needed - -;
j++;}
Ans = max(ans, platform_needed);
}
Return ans;
```

Time complexity is: O (2n log n) for sorting and O(n) for original implementation.

Job Sequencing Problem (GFG).

Approach: Greedy Algorithm.
Sort the array in descending order according to it's profit. And then create a one array with size as maximum deadline with all the assigned values as -1. And now according to jobs deadline and arrays index fill the array and return it's size and maximum profit.
Algo:

```
Bool comparison(job a, job b){
Return profit > b.profit);
}
sort(arr, arr+n, comparison);
Int maxi = arr[0].dead;
For(int I=1;i<n;i++;) maxi = max(maxi, arr[i].dead);
Vector<int> slot(maxi+1, -1);
Int countjobs = 0, jobprofit =0;
for(int I=0;i<n;i++){
for(int j = arr[i].dead;j>0;j- -;){
if(slot[j] == -1){
slot[j] = I;
countjobs++;
jobprofit += arr[i].profit;
Break;
}
}
}
Return {countjobs, jobprofit};
```

Time complexity is: O(n ^ 2).

Fractional Knapsack (GFG):

Approach: Greedy
Algo: first find the profit for per weights and sort the array according to that using comp function. And apply the knapsack also.

```
Bool static comp{Item a, Item b){
Double r1 = (double)a.value / (double)a.weight;
Double r2 = (double)b.value / (double)b.weight;
Return r1>r2;
}
 double fractionalKnapsack(int W, Item arr[], int n)
   {
       sort(arr, arr+n, comp);
       int curr_weight = 0;
       double final_profit = 0.0;
       for(int i=0;i<n;i++){
```

```
            if(curr_weight+arr[i].weight <= W){
                curr_weight += arr[i].weight;
                final_profit += arr[i].value;
            }
            else{
                int remain = W-curr_weight;
                final_profit += (arr[i].value / (double)arr[i].weight) * (double)remain;
                break;
            }
        }
        return final_profit;
    }
```

Time complexity is: N log N + N

Number of Coins (GFG)

Approach : Three Approach 1) Recursion 2) Recursion+Memoization 3) Tabulation method.

We Use Third Approach because first two are gives the TLE.

Algo:

```
Int solvetab(int coins[], int M, int V){
        vector<int> dp(V+1, INT_MAX){
        dp[0] = 0;
for(int I=1; I<=V;i++){
for(int j=0;j<M;j++){
if(i-coins[j] >=0 && dp[i-coins[j]] != INT_MAX) {
dp[i] = min(dp[i], 1+dp[i-coins[j]];
}
}
}
if(dp[V] == INT_MAX) return -1;
Return dp[I];
}
int minCoins(int coins[], int M, int V) {
Return solvetab(int coins[], int M, int V);
}
```

Time complexity:

1) Recursion -> Exponential
2) Recursion+Memoization - > O (V * M)
3) Tabulation method -? O(V*M) space = O(V)

N meetings in one room (GFG)

Approach : Greedy Approach. Sort meetings according to their finish time and then use the greedy method to solve it.
Algo:

```
Struct meeting{
Int start;
Int end;
Int pos;
};
Bool comp(meeting m1, meeting m2){
If(m1.end < m2.end) return true;
Else if(m1.end > m2.end) return false;
Else if(m1.pos < m2.pos) return true;
Return false;
}
Int maxmeetings(int start[], int end[], int n){
Meeting meet[n];
for(int I=0;i<n;i++){
meet[I].start = start[I];
meet[I].end = end[I];
meet[I].pos = I+1;
}
Int limit = meet[0].end;
Int ans = 1;
for(int I=1;i<n;i++){
if(meet[I].start > limit){
ans++;
Limit = meet[I].end;
}
}
Return ans;
```

Time complexity is: O(n log n + n)

Subset Sum (GFG)

Approach: Bruit force
create all possible 2^n subsets from given array and store all the sums in final array.
Algo:

```
Vector<int> final;
for(int I=0; I < (1 << N);I++){
Int ans  = 0;
for(int j = 0;j<N;j++){
if(I & ( 1<<j){
```

Ans += arr[j]; } }
Final.push_back(ans);
Return final;
Here logic is Bit manipulation for creating all the 2^n powerset of given array.
Time complexity is (2^n * n) 2^n for outer loop and n for inner loop.

Approach 2: Optimal Solution. (Recursion).
At each position we have two option first is either pick that element into our sum or not. So for that we use recursion here.
Algo:

```
Void solve(int idx, int sum, vector<int>& arr, vector<int>&answer, int N){
        if(idx == N){
        answer.push_back(sum);
        return;
        }
solve(idx+1, sum+arr[idx], arr, answer, N);
//do not pick the element
solve(idx+1, sum, arr, answer, N);
}
```

Time complexity is 2^n + 2^nlog2^n for sort the answer at the end.

90. Subsets II

Approach: recursion
At each point check and do not take the duplicate elements in our data structure.
Algo:
First sort the nums array because it will helps to find out the duplicate elements.

```
Void solve(int idx, vector<int>&nums, vector<int>&d_s, vector<vector<int>>
ans){
        ans.push_back(d_s); // first empty array
for(int I=0; I<idx; I++)[
if(I != idx && nums[I] == nums[i-1]) continue;
d_s.push_back(nums[I]);
solve(I+1, nums, d_s, ans);
d_s.pop_back();
} }
```

Time complexity : 2^n * n // n is average size of all the data structures
Space complexity: 2^n * k // k for average length of data structures
Auxiliary space means depth of recursion is O (n)

39. Combination Sum I

Approach: recursion.
Algo:
Void solve(int idx, int sum, vector<int>&candidates, vector<int>&temp, vector<vector<int>>&ans, int target, int n){
if(sum == target) ans.push_back(temp);
Else if(sum < target){
for(int I=idx; I<n; I++){
Temp.push_back(candidates[I]);
solve(I, sum, candidates, temp, ans, target, sum_candidates[I]);
Temp.pop_back(); }
}
Time complexity is 2^t * k

39. Combination Sum II

Approach : Recursion
Algo:
First sort the input array.
Void solve(int idx, int target, int n, vector<int>&temp, vector<int>&candidates, vector<vector<int>>& ans){
if(target == 0) ans.push_back(temp); return;
for(int I=idx; I<n; I++){
if(I != idx && candidates[i-1] == candidates[I]) continue;
if(candidates[I] > target) break;
Temp.push_back(candidates[I]);
solve(I+1, target-candidates[I], n, temp, candidates, ans);
} }
Time complexity is 2^n * 2^nlog2^n //extra for sorting

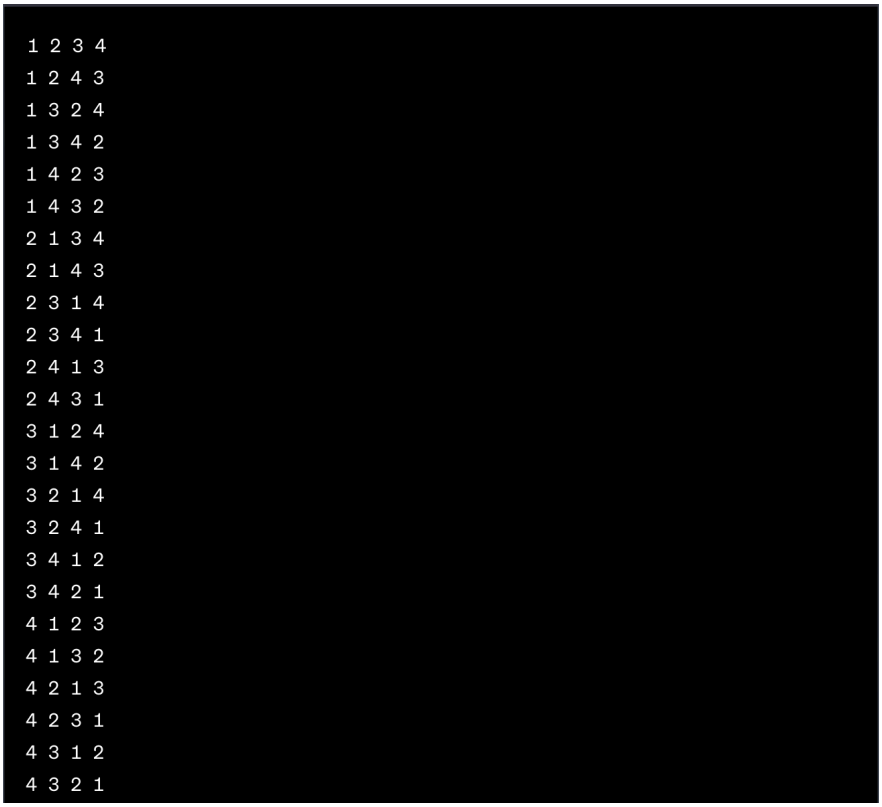
131. Palindrome Partitioning

Approach: Recursion
Algo:
Write one function which takes the string, start index and end index and return whether string is palindrome or not.
Void solve(int idx, string s, vector<string>& temp, vector<vector<string>>& ans){
if(idx == s.size()){
Ans.push_back(temp);
Return;
}
for(int I=idx; I<s.size();I++){
if(is_palindrome(s, idx, I) // if the above recursion string is palindrome
{

temp.push_back(s.substr(idx, i-idx+1));
solve(I+1, s, temp, ans);
} }
Time complexity is 2^n * n.

60. Permutation Sequence

Approach: Basic math. Here, thing is that ke bhai, first you can find the factorial of that number and then divide it with n. Here you can able to find the number blocks in which 1 number is first then 2 is first then 3 is first and like wise.



Here for n =4 there are total 4 blocks with size 6. So create a array with size n and store 1 to n elements on it.
Now we want to find 4th position and indexing start with 0 so make k = k-1.
Now the logic is for ans string first add the k/fact, and erase that added elements from the array then update k using k = k % fact and fact = fact / array size.
* and remember that find the factorial from 1 to only n-1.
Algo:
Vector<int> numbers; int fact = 1;
for(int I=1;i<n;i++) { fact = fact*I; numbers.push_back(i); }
Numbers.push_back(n);
K = k-1;

```
String ans = “ ”;
while(true){
Ans += to_string(numbers[k/fact]);
Numbers.erase(numbers.begin()+k / fact);
if(numbers.size() == 0) break;
K = k % fact;
Fact = fact / numbers.size();
} return ans;
```

Time complexity is O (n) + O (t) where t is number of blocks which can made by factorial.

46. Permutations

Approach : Recursion and Backtracking

Algo:

```
Void solve(int idx, vector<int>&nums, vector<int>&temp,
vector<bool>&choosen, vector<vector<int>>& ans){
if(temp.size() == nums.size()){
ans.push_back(temp);
Return;}
for(int I=0;i<nums.size();I++){
        if(choosen[I] == false){
        temp.push_back(nums[I]);
        choosen[I] = true;
        solve(I+1, nums, temp, choosen, ans);
        temp.pop_back();
        choosen[I] = false;
}
}
```

Time complexity is O (n! * n)

51. N- Queens

Approach: Recursion and Backtracking

Algo:

first write a is_safe function for check a valid cell in board.

Then write a recursive function

```
Bool is_safe(int row, int col, vector<string>&board){
        for(int I=0;i<board.size();I++){
                if(board[row][I] == ‘Q’) return false; }
        for(int I=row, j=col; I>=0 && j>=0; I—, j—){
                if(board[I][j] == ‘Q’) return false;}
        for(int I=row, j=col; I<board.size() && j>=0; I++, j- -){
                if(board[I][[j] == ‘Q’) return false;}
```

```
Return true;
}
Void solve(int col, vector<string>&board){
if(board.size() == col){
ans.push_back(board);
Return;
}
for(int I=0;i<board.size();I++){
if(is_safe(I, col, board){
board[I] [j] = 'Q';
solve(col+1, board);
board[I] [j] = '.';
}
```

*remember that for initialise new board use the vector<string>board(n, string(n, '.')); do not take directly vector<string>board(n, '.');

Time complexity is O(n ^ 2) because at each stage there are two option whether you put queen or not?

37. Soduku Solver

Approach: Recursion with backtracking.

Instead of other recursion here we write a different recursion with return type of boolean. Here we have to return true or false at each step after filling a cell whether that filled cell is right or wrong that's why we write a function as bool.

Algo:

First write a is_valid function to check all the conditions of valid soduku.

```
Bool is_valid(int col, int row, vector<string>&board, char c){
for(int I=0;i<9;i++){
        //for entire row
        if(board[row][I] == c) return false;
        //for entire column
        if(board[I] [col] == c) return false;
        // now special condition for that particular 3*3 box
if(board[3 *(row/I) + I/3] [3 * (col/3) + I%3] == c) return false;
}
Return true;
}
```

Second is let's write recursive function.

```
Bool solve(vector<string>&board){
for(int I=0;i<board.size();I++){
for(int j=0;j<board[0].size();j++){
        if(board[I][j] == '.'){
                for(char c = '1'; c<='9';c++){
```

```
                    if(is_valid(I, j, board, c){
                        board[I] [j] = c;

                        if(solve(board)) return true;
                        else board[I][j] = ‘ . ‘;
                        }
                    }
                return false;
} } }
Return true;
}
```

Worst case time complexity is O(9 ^ m) where m is number of empty cells in board.

Now let’s understands the line if(board[3 *(row/I) + I/3] [3 * (col/3) + I%3] == c) return false;
Suppose we want to check if the character c is present in the subgrid where the cell at (row, col) = (4, 7) is located.

(row/3) = 4/3 = 1, so the row group is 1.
(col/3) = 7/3 = 2, so the column group is 2.
3*(row/3) = 3*1 = 3, and 3*(col/3) = 3*2 = 6. Therefore, the starting index of the subgrid is (3, 6).
Now, we iterate i from 0 to 8 to check each cell within the subgrid:

When i = 0, i/3 = 0 and i%3 = 0. So, the cell at (3+0, 6+0) will be checked.
When i = 1, i/3 = 0 and i%3 = 1. The cell at (3+0, 6+1) will be checked.
Similarly, for i = 2, (3+0, 6+2) will be checked.
When i = 3, i/3 = 1 and i%3 = 0.

M-Coloring Problem (GFG)

How to store graph in c++?
There are two ways to store a graph in c++,
1)matrix
2)list representation
Approach: recursion and backtracking
Algo:
Write 2 separate function one is recursive and another is called is_safe to check whether we able to assign colour or not?

```
Bool is_safe(int n, int node, int color[], bool graph[101][101],int col){
for(int I=0;i<n;i++){
if(I != node && graph[I][node] == true && color[I] == col) return false;
```

```
Return true;
}
Bool solve(int node, int n, int m, int color[], bool graph[101][101]){
if(n == node) return true;
for(int I=1;i<=m;i++){
if(is_safe(n ,node, color, graph, I){
color[node] = I;
if(solve(node+1, n, m, color, graph)) return true;
} }
Return false;
}
```

Time complexity is O (n ^ m) where n is no of vertices and m is the number of colours.

Rat in a Maze Problem - I (GFG)

Approach: recursion plus backtracking

Algo: start from initial position (0,0) if it is 1 then from each cell write recursion which can move 4 side down, left, right, upper. Maintain the visited matrix also.

```
Void solve(int row, int col, int n, vector<vector<int>>& visited,
vector<vector<int>>&m, vector<string>& ans, string s){
if(row == n-1 && col == n-1){
Ans.push_back(s);
Return; }
//downward
if(row+1 >= 0 && !visited[row+1][col] && m[row+1][col] == 1){
visited[row][col] = 1;
solve(row+1, col, n, visited, m, ans, s+'D');
visited[row][col] = 0;
}
//left side
if(col-1 >= 0 && !visited[row][col-1] && m[row][col-1] == 1){
visited[row][col] = 1;
solve(row, col-1, n, visited, m, ans, s+'L');
visited[row][col] = 0;
}
//right side
if(col+1 < n && !visited[row][col+1] && m[row][col+1] == 1){
visited[row][col] = 1;
solve(row, col+1, n, visited, m, ans, s+'R');
visited[row][col] = 0;
}
```

```
//upward
if(row-1 >= 0 && !visited[row-1][col] && m[row-1][col] == 1){
visited[row][col] = 1;
solve(row-1, col, n, visited, m, ans, s+'D');
visited[row][col] = 0;
}
```

*remember that in main function first check if starting cell (0,0) have 1.
Time complexity will be O (4 ^ m*m) where 4 means at each cell we have 4 option left, right, up and down.

139. Word Break

Approach: 1) Recursion (TLE) , 2)Recursion with Memoization(TLE), 3)Dp with Tabulation (Accepted)
Algo:
Make a unordered_set to store all the dictionary words and one 2D dp array. Now fill the dp table according to conditions. And for answer if dp[0][n-1] contains true then return true otherwise false.

```
unordered_set<string> st(wordDict.begin(),wordDict.end());
Vector<bool<int>>dp(n, vector<bool>(false));
Int n = s.size();
for(int len = 1; len<=n; len++){
for(int I=0;i<n-len;i++){
Int j = I + len-1;
String sub = s.substr(I,len);
if(st.count(sub)){
dp[i][j] = true;
Continue;
}
for(int k=I;k<j;k++){
if(dp[I] [k] && dp[k+1[j]){
dp[i][j] = true;
Break;
} } } }
Return dp[0][n-1];
```

Time complexity is O (n ^ 3);

Word Break II (Coding Ninja)

Approach: 1) Only Recursion 2) Recursion With memoization
Algo:
First convert wordDict into a set.

```
Vector<string> solve(int idx, int n, string s, unordered_set<string>& st,
unordered_map<int, vector<string>> mp){
if(mp.count(idx)) return mp[idx];
if(idx == n){
Result.push_back(“”);
Return result;
}
Vector<string> result;
String word= “”;
for(int I=idx; I<n; I++){
Word += s[I];
if(st.count(word)){
Vector<string> suffixes = solve(I+1, n, s, st, mp);
For(string suffix : suffixes){
if(suffix.empty()){
Result.push_back(word);}
Else{
Result.push_back(word + “ “ + suffix);
} } } }
mp[dx] = result;
Return result;
}
```

Time complexity is O(N^3) n^2 for recursion and another n for check a valid string for each word.

Find Nth Root Of M (coding Ninja)

Approach : binary search

Algo:

```
Int low = 1, high = m;
while(low <= high){
Int mid = (low + high)/2;
if(pow(mid, n) == m ) return mid;
Else if(pow(mid, n) > m) high = mid-1;
Else low = mid+1;
}
Return -1;
```

Time complexity is O(log m)

Matrix Median (Interview Bit)

Approach : Binary search

Here logic is first store all the elements of matrix into 1D array and then return the median but constraint is we do not allow to take any extra space. So for

that we use binary search approach. Here logic is take the each element from each sorted row and apply binary search on it if all the elements <= that element is == elements >= that element then simply return that element.
Algo:
Int helper(vector<int>& a, int val){
Int l = 0, r = a.size()-1;
while(l <= r){
Int mid = (l+r)/2;
if(a[mid] <= val) l = mid+ 1;
Else r = mid-1;
}
Return l;
}
Int findmedian(vector<vector<int>>&A){
Int n = A.size() m= A[0].size();
Int low = 1, high = 1e9; // 1e9 because this constraint is given in question
while(low <= high){
Int cnt = 0, mid = (low + high)/2;
for(auto it: A){
Cnt += helper(it, mid);
}
if(cnt <= (n*m)/2) low = mid+1;
Else high = mid -1;
}
Return low;
}
Time complexity is O (n * log(m)) where n is number of rows in matrix and log (m) for binary search in each row.

540. Single Element in Sorted Array

Approach: 1) use map to keep track of occurrences of each element. 2) use bitwise xor and at the end return result 3) binary search.
Here we observe that at even index first instance is present of twice elements and at the odd index second instance is present so we use this observation to solve this question using binary search.
Algo:
Int low = 0, high = nums.size()-2 // -2. Because we move low on right side using mid+1.
while(low <= high){
Int mid = (low + high)/2;
Int adj;
if(mid %2 == 0) adj = mid+1;

```
Else adj = mid-1;
if(nums[mid] == nums[adj]) low = mid+1;
Else high = mid-1;
}
Return nums[low];
```

Time complexity is O (log n) because we used binary search.

33. Search in rotated sorted array

Approach: Binary search

Logic: here logic is at index I you have to check whether your left subarray is sorted or not if it is sorted then find your target on left side if target is not found then move to the right subpart of array.

Algo:

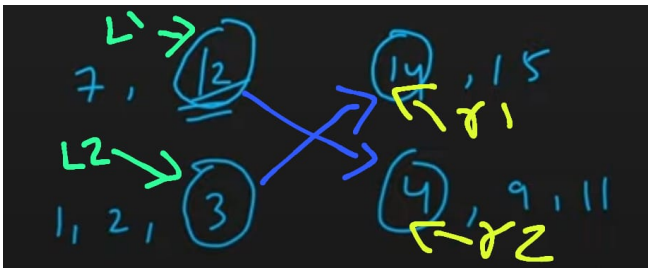
```
Int low= 0, high = nums.size()-1;
while(low <= high){
Int mid = (low+high) /2;
if(nums[mid] == target) {
Return mid;
}
Else{
if(nums[low] <= nums[mid]){
        if(target >= nums[low] && target<= nums[mid]) high = mid-1;
        else low = mid+1;
Else
        if(target >= nums[mid] && target<=nums[high]) low = mid+1;
        Else high = mid-1;
}
}
Return -1;
```

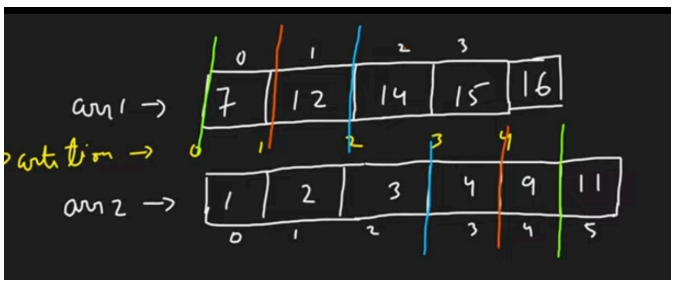
Time complexity is O(log n)

4. Median of two sorted arrays

Approach: Binary search

Logic is that you have to just make a valid partition of two arrays. In that l1 ans l2 will present in left side and r1 and r2 will present on right part of partition. And you have to make sure that l1<= r2 and l2<=r1. Like:





In ordere to decrease l1,l2 and increase the r1,r2 you have to use the binary search method.
Algo:

```
Int n1 = nums1.size(), n2= nums2.size();
if(n2 < n1) return findMedianSortedArrays(nums2, nums1);
Int low = 0, high = n1;
while(low <= high){
Int cut1 = (low+high) /2;
Int cut 2 = (n1 + n2 + 1)/2 - cut1;
Int left1, left2,right1,right2;
if(cut1 == 0) left1 = INT_MIN;
Else left1 = nums1[cut1-1];

if(cut2 == 0) left2 = INT_MIN;
Else left2 = nums2[cut2-1];

if(cut1 == n1) right1 = INT_MAX;
Else right1 = nums1[cut1];

if(cut2 == n2) right2 = INT_MAX;
Else right2 = nums2[cut2];

if(left1 <= right2 && left2 <= right1){
if( (n1 + n2) % 2 ==0) return (max(left1,left2) + min(right1, right2) /2.0);
Else return max(left1, left2);
}
Else if(left1 > right2){
High = cut1-1;
}
Else{
Low = cut1+1;
}
Return 0.0;
```

Time complexity is O(log (m + n)) where m and n is size of two arrays.

K-th element of two sorted Arrays(GFG)

Approach: there are number of ways to solve this question but we use here the optimised way which is binary search.
This problem is almost similar to our previous problem, just we have to change some condition of that cut1 and cut2 conditions.
Algo:

```
if(m > n) return kthElement(arr2, arr1, m, n, k);
Int low = max(0, k-m), high = min(k , n);
while(low <= high){
Int cut1 = (low + high) /2;
Int cut2 = (k - cut1);
Int left1, left2, right1, right2;
if(cut1 == 0) left1 = INT_MAX;
Else left1 = arr1[cut1-1];
if(cut2 == 0) left2 = INT_MAX;
Else left2 = arr2[cut-1];
if(cut2 == n) right1 = INT_MIN;
Else right1 = arr1[cut1];
if(cut2 == m) right2 = INT_MIN;
Else right2 = arr2[cut2];
if(left1 <= right2 && left2<= right1) return max(left1, left2);
Else if(left1 > right2) high = cut1-1;
Else low = cut1+1;
}
Return 1;
}
```

Time complexity is O (log min(m + n))

Allocate Books (InterviewBit)

This is quite tricky🥲 and Important question.

Approach:Binary search

Question is how we arrive at binary search approach instead of dynamic programming and recursion?

One observation is that our answer will lies between the min element from array and maximum(sum of all elements). So here the very first approach is arrive in our mind is binary search.

Algo:

Write one function which gives the number of students can formed after assigning n books.

```
Bool min_pages(vector<int>&A, int students, int val){
Int sum = 0, cnt=0;
for(int I=0; I<A.size(); I++){
if(sum + A[I] > val){
Cnt++;
Sum = A[I];
if(sum > val) return false;
}
else Sum += A[I];
```

```
}
Return cnt<students;
}

Int books(vector<int>&A, int B){
Int low = INT_MAX, high = 0;
for(int I=0;i<A.size();I++){
High += A[I];
Low = min(low, A[I];
}
while(low <= high){
Int mid = (low + high)/2;
if(min_pages(A, B, mid) high = mid-1;
Else low = mid+1;
}
Return low;
```

Time complexity is O (log n) *n;

Aggressive Cows(Coding Ninja)

Approach: After observing a question the very first approach arrives in our mind is recursion but it will take exponential time. After carefully observed a question I realise that all the distances between all the cows will lies between 1 and maximum - minimum element from array. So we conclude that our best and optimal approach is binary search.

Algo:

```
Bool place_cow(vector<int>& stalls, int cow, int mid){
Int cnt = 1;
Int place = stalls[0];
for(int I=1;i<stalls.size();I++){
if((stalls[I] - place) >= mid){
Cnt ++;
Place = stalls[I];
}
if(cnt == cow) return true;
}
Return false;
}
Int Aggressivecows(vector<int>&stalls, int k){
sort(stalls.begin(), stalls.end());
Int low = 1, high = stalls[stalls.size()-1] - stalls[0];
Int ans = 0;
while(low <= high){
```

```
Int mid = (low + high)/2;
if(place_cow(stalls, k, mid){
Ans = max(ans, mid);
Low = mid+1; }
Else
High = mid-1;
}
Return ans;
```

Time complexity is O (n * log n)

Min Heap (coding Ninja)

Approach: it is a basic heap question you have to some understanding of heap data structure.
Algo:
make separate functions for insert and remove elements in heap.
And finally return the resultant vector.
Link: https://www.codingninjas.com/codestudio/problems/min-heap_4691801?topList=striver-sde-sheet-problems&utm_source=striver&utm_medium=website&leftPanelTab=1

215. Kth largest element:

Approach: Minheap
this question is solved by very simple approach. In that you just sort the array and return nums[n-k] but it will take O(n log n) time. Another way is use heap data structure(queue). In that build a max heap in o(n) time and return the kth element. But,but,but the another best optimised solution is build a min heap until the size of queue is k and whoever queue size is > k pop the elements from queue and at the end return the top of queue.
Algo:

```
priority_queue<int, vector<int>, greater<int>> pq;
for(int I=0;i<nums.size();I++){
Pq.push(nums[I);
if(pq.size() > k) pq.pop();
}
Return pq.top();
```

Time complexity is O (n)

Maximum Sum Combinations (Interview Bit)

Approach: Maxheap (priority queue).
Here logic is, first sort both arrays in descending order so all the max elements are occurred first in the arrays. After create a priority queue with pair of pair in that outer pair will store the sum and another pair and inner pair

will store the indices of array A and B. Maintain another set containing the pair which check whether that indices are used or not prior? Now iterate from 0 to C and take all the elements of priority queue and insert it into ans vector parallely push the sums and indices into the priority queue.
Algo:

```
sort(A.rbegin(), A.rend());
sort(B.rbegin(), B.rend());
priority_queue<pair<int, pair<int,int>>>;
Set<pair<int,int>> st;
Vector<int> ans;
St.insert({0,0});
Pq.push({A[0] + B[0], {0,0}});
for(int I=0;i<C;i++){
Auto it = pq.top();
Ans.push_back(it.first);
Pq.pop();
Int Left = it.second.first;
Int Right = it.second.first;
if(left+1 < A.size() && !st.count({left+1, right}){
Pq.push({A[left+1] + B[right], {left+1,right}});
St.insert({left+1, right}); }

if(right+1 < B.size() && !st.count({left, right+1}){
Pq.push({A[left] + B[right+1], {left,right+1}});
St.insert({left, right+1}); }
}
Return ans;
```

Time complexity is O (C log C)

295. Find Median from Data stream

Approach: Min and Max heap
This is hard as well very tricky question but solution is also super simple. Lets example we have numbers like 4, 1, 3, 2, 6, 5 and all the data points will occurs in continues streaming so how to handle it. If we count the median at each time then obviously it will not a feasible solution. So. Here we maintain two heaps called min_heap and max_heap and store that continues stream of data points. And for median if our input data points are odd then we simple return the top element of heap which contain higher size then another otherwise we will take top of elements from both heaps and return the average.
Algo:
—> At the top create two heaps called:

priority_queue<int> max_heap;
priority_queue<int, vector<int>, greater<int>> min_heap;
—> now in the addendum function add the number into our heap using below conditions.
if(min_heap.empty() && max_heap.empty()) max_heap.push(num);
Else{
if(max_heap.top() < num) min_heap.push(num);
Else max_heap.push(num);
}
—> now writes the logic in find mean function
Double FindMean(){
if(min_heap.size() == max_heap.size())
return ((double)min_heap.top() + (double)max_heap.top()) / 2.0;
Else if(max_heap.size() == min_heap.size() +1)
return (double)max_heap.top();
Else if(max_heap.size()+1 == min_heap.size())
return (double)min_heap.top();
Else if(max_heap.size() > min_heap.size()+1){
Int ele = max_heap.top();
min_heap.push(ele);
max_heap.pop();
Return FindMedian();
}
Else if(min_heap.size() > max_heap.size()+1){
Int ele = min_heap.top();
max_heap.push(ele);
min_heap.pop();
Return FindMedian();
}
Return 0.0;
}
Time complexity will be O (log n) because we used a heap data structure
Space complexity is O (n) because we store n elements in two heaps.

Merge K Sorted Arrays (GFG)

Approach: 1)create one array take all the elements from 2d vector and sort it.
2) take one minheap(priority queue) put all the elements on it and convert it into vector.
3) make your own data structure and comparator to make efficient use of priority queue. In this we take all the elements of all arrays at same time from its position and then put it into queue. We also use our own comparator to store the result in final vector.

Algo:
//make your data structure
Class data{
Public:
Int val, apos, valpos;
Data(int v, int ap, int vp){
Val = v;
apos= ap;
Valpos = vp;
} };
// make own comparator
Struct my_comp{
Bool operator()(data &d1, data &d2){
Return d1.val > d2.val;
} };
//actual logic
Vector<int> ans;
priority_queue<data, vector<data>, my_comp>> pq;
for(int I=0;i<k;i++){
Data d(arr[i][0], i, 0);
Pq.push(d);
}
while(!pq.empty()){
Data curr = pq.top(); pq.pop();
Ans.push_back(curr.val);
if(vp+1 < arr[ap].size()){
Data d(arr[ap][vp+1] ,ap, vp+1);
Pq.push(d);
} }
Return ans;
Time complexity: approach 2: O (n log k) where n is all the elements across arr and k is number of sub vectors of arr.
Approach 2: for push all the elements O (n log n) and poping O (n log n) overall will also O(n log n).
In summary, if K is significantly smaller than N or the size of each individual array is large, Approach 2 (flattening all arrays into a single sorted sequence) may be more efficient. On the other hand, if K is close to N or both are similar in magnitude, or the size of each individual array is relatively small, Approach 3 (using a priority queue of custom data structure) may be more efficient. It's important to consider the specific characteristics of the problem and the input data to determine which approach is more suitable in a given scenario.

347. To K frequent Elements

Approach: 1) use the map
2) use max heap with pair data structure

Algo:

First store all the frequency of elements in map then insert all that pairs into max heap. While inserting elements remember that you can push second element of pair into first element of heap. And then return top k's second elements of priority queue.

```
Map<int,int> mp;
for(auto it: nums) mp[it]++;
Priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;
Vector<int> ans;
for(auto it: mp){
Pq.push({it.second,it.first});
if(pq.size()>k) pq.pop();
}
while(!pq.empty()){
ans.push_back(pq.top().second);
pq.pop();}
Return ans;
```

approach 1: Time. - O (n log n) Space - O(n)
Approach 2: Time O (n log k) Space O(k)

225. Implementation Stack Using Queues

Approach: 1) Using 2 Queues
2) Using Single Queue

Algo: 1)

```
Queue<int> q1, q2;
Void push(int x){
while(!q1.empty()){
q2.push(q1.front());
q1.pop();
}
Q1.push(x);
while(!q2.empty()){
q1.push(q2.front());
q2.pop(); }
Int pop(){
Int val = q1.front();
q1.pop();
Return val;
}
```

```
Int top(){ return q1,front());
Bool empty(){ return q1.empty());

2)
Void push(int x){
q1.push(x)); }
Void pop(){
Int size = q1.size();
while(size > 1){
Q1.push(q1.front());
Q1.pop();
Size- -;
}
Int val = q1.front();
q1.pop();
Return val;
}
int top() { return q1.back());  }
Void empty() { return q1,empty()); }
```

Time and Space Complexities:

1) Push -> O (n) and for empty, pop and top O (1)
2) Pop -> O (n) and for empty, push and top O (1)

Implement Queue Using Stack

Approach: Stacks Use 2

Here logic is when you push the elements simply push elements into stack 1. And while you make pop operation check if stack2 is not empty if it is empty then shift all the elements of stack 1 into stack2 and pop the top element of stack2 and if it is not empty then simply pop the top element of stack2. Same process will followed for peek operation and for check the empty simply check both the stacks will not empty.

Algo:

```
Stack<int> st1, st2;
Void push(int x){ st1.push(x); }
Int pop(){
If (st2.empty()){
St2.push(st1.top());
st1.pop();}
Int element = st2.top();
St2.pop();
Return element; }
```

```
Int pop(){
If (st2.empty()){
St2.push(st1.top());
st1.pop();}
Return st2.top(); }
Bool empty(){
Return (st1.empty() && st2.empty());
}
```

Time complexity for push is O (1) but time complexity for pop and peek will also not O(1) or O(n) always it will be a Amortised.

20. Valid Parenthesis

Approach: stack.

Algo:

```
stack<char> st;
        if(s.size() == 1) return false;
        for(int i=0;i<s.size();i++){
            if(s[i] == '(' || s[i] == '{' || s[i] == '['){
                st.push(s[i]);
            }
            else if(s[i] == ')' || s[i] == '}' || s[i] == ']'){
                if(st.empty()) return false;
                if(s[i] == ')' && st.top() == '(') st.pop();
                else if(s[i] == '}' && st.top() == '{') st.pop();
                else if(s[i] == ']' && st.top() == '[') st.pop();
                else return false;
            }
        }
        return st.empty();
```

Time complexity is O(n);

496. Next Greater Element I

Approach: Use stack and hash map to keep track of next greater element. Logic is start from right hand side and for last element we all know there is no next greater element because it is last index so. For that put -1 and for remaining elements maintain the stack which store the elements from right side and for fill the hash map for each number check if top of the stack is > then the number otherwise pop that element. And lastly return the ans vector which contain all the next greater element of nums1 from hash map.

Algo:

```
Int n = nums2.size();
unordered_map<int,int>mp; stack<int> st; vector<int> ans;
```

```
mp[nums2[n-1]] = -1;
St.push(nums2[n-1]);
for(int I = n-2;i>=0;i - -){
while(!st.empty() && nums[I] >= st.top()) st.pop();
if(!st.empty(){ mp[nums[I]] = st.top(); }
mp[nums2[I]] = -1;
}
For(int I=0;i<nums1.size();I++) ans.push_back(mp[nums1[I]];
Return ans;
```

Time complexity is : O(n) where n is total numbers in nums2 array.

503. Next Greater Element II

Approach: 1) bruit force iterate array circularly (Accepted)
2) Use Stack

Logic is from right side skip very first element and store another elements. Remember for store the elements if element is greater then top of stack then pop the stack until the top of stack is not >= to the input element.
Now iterate again from right hand side and remember at this time you have to iterate entire array so don't skip very first element from R.H.S now if input element is greater then top f stack then pop the elements of stack until input element >= top of stack. If stack was empty then ans[I] = -1, or if top of stack > nums[I] then an[I]= was nums[I].
Algo:

```
Int n= nums.size();
Vector<int>ans(n,-1); stack<int>st;
For(int I=n-2; I>=0; I- -){
while(!st.empty() && st.top() <= nums[I) st.pop();
St.push(nums[I]);
}
for(int I=n-1; I>=0; I - -){
while(!st.empty() && st.top() <= nums[I) st.pop();
if(st.empty()) ans[I] = -1;
Else ans[I] = nums[I];
St.push(nums[I]);
}
Return ans;
```

Time complexity is O (n) + O(n); and for bruit force it will O (n ^ 2)

Sort a Stack (Coding Ninja)

Approach : 1) Recursion and without extra space
2) Using the another extra stack

Algo:

```
Stack<int> temp;
while(!stack.empty()){
Int element = stack.top();
stack.pop();
while(!temp.empty() && temp.top() < element){
stack.push(temp.top());
Temp.pop();}
temp.push(element);
}
while(!temp.empty()){
stack.push(temp.top();
Temp.pop();
}
```

Time complexity is O(n^2) where n is total number of elements in input stack
In recursion the time complexity will also O (n^2).

Nearest Smaller Element (Interview Bit)

Approach: Use stack
Previously we solved a question called next greater element. In that we maintained a hash map for store the elements. But in this question we all know the minimum answer is -1. So we don't use map here. Because if there are no such nearest smaller element is present then we simply put -1 from stack.
Algo:

```
Int n = A.size();
Stack<int> st;
Vector<int> ans(n);
for(int I=0;i<n;i++){
while(!st.empty() && st.top() >= A[I]) st.pop();
ans[I] = st.top();
St.push(A[I]);
}
Return ans;
```

Time complexity is O (n).

146. LRU Cache(IMP)

Approach: capacity size doubly linked list and Hashmat.
Here logic is you can use the doubly linked list to implement LRU.
Firstly create a linked list which contains the head and tail node and remember that we created a head and tail because we don't want so many if else conditions for NULL checkup.

Overall, this LRUCache implementation ensures that the most recently used items remain at the front of the cache, while the least recently used items are gradually pushed towards the tail and eventually evicted when the cache reaches its capacity.
Sol link: https://leetcode.com/problems/lru-cache/submissions/968015124/
The get method is take O(1) time and put will also take O(1) time and map will take O (n) is thousand of case otherwise it always take O(1) time to search a element, and remember map will take O(n) time in very rare case.

460.LFU Cache(IMP)

Approach : Doubly linked list and 2 hash maps
The given approach will try to implements an LFU (Least Frequently Used) cache using a combination of unordered maps and lists. The LFUCache class maintains two unordered maps: keynode for storing key-node pairs and freq_list_map for storing frequency-node lists. Each node contains information about the key, value, frequency, and links to the next and previous nodes. The LFUCache constructor initializes the cache capacity and other variables. The update_freq function updates the frequency of a node, moving it to the corresponding frequency-node list. The get function retrieves the value associated with a key, updates the frequency, and returns the value or -1 if the key is not found. The put function adds a new key-value pair or updates an existing pair, considering the cache capacity. If the cache is full, it removes the least frequently used node based on the min_freq variable.
Overall, understanding the key components, such as node structure, frequency tracking, and insertion/removal logic, is crucial for effectively implementing and utilizing the LFU cache in an interview setting.
Sol Link: https://leetcode.com/problems/lfu-cache/submissions/968072018/
Time complexity will be O(1) for all the operations.

84. Largest Rectangle in Histogram

Approach: Stack.
If you remember our trapping rain water problem we find the maximum at each with iteration. Similarly in this case we create an two array called left_smaller and right_smaller to find the left and right element at each point in height array. And for find the maximum area we take the (right_smaller - left_smaller) +1 * height[I];
Algo:

```
Int n = heights.size();
Vector<int> left_smaller(n), right_smaller(n);
Stack<int> st;
for(int I=0;i<n;i++){
```

```
while(!st.empty() && heights[st.top()] >= heights[I]) st.pop();
if(st.empty) left_smaller[I] = 0;
Else left_smaller[I] = st.top();
St.push(I);
}
while(!st.empty()) st.pop();
for(int I=n-1;I>=0;i- -){
while(!st.empty() && heights[st.top()] >= heights[I]) st.pop();
if(st.empty) right_smaller[I] = n-1;
Else right_smaller[I] = st.top();
St.push(I);
}
Int ans = 0;
for(int I=0;i<n;i++){
Int val = ((right_smaller[I] - left_smaller[I]) +1) * heights[I] ;
Ans = max(ans,val);
```

Time complexity is O(n) + O(n) + O(n). n for two pass and n for find the final answer.

*Now one question is arise in mind , can we convert those 2 pass into a single?

So, the answer is yes. If we carefully observe our stack at each iteration then it has contain both the right_smaller and left_smaller elements.

How it is possible?

```
Algo:
Int n= heights.size();
Stack<int>st;
Int ans= 0;
for(int I=0;i<=n;i++){
while(!st.empty() && (I==n || heights[st.top()] >= heights[I]){
Int height = st.top();
St.pop();
Int width = I - st.top() - 1;
Ans = max(Ans, height*width);
}
st.push(i);
}
Return ans;
```

Time complexity is just O(n) and space will also O(n) for stack. Here we reduced two pass as well 2 extra array for store the right_smaller and left_smaller.

239. Sliding Window Maximum

Approach: Doubly Ended Queue

Here logic is you can maintain a k size queue for your answer. If the current front of your queue is from previous subarray then remove it. For front if the current element which is nums[I] is greater then the front then remove the front and push that nums[I] at the front of queue. Now for remaining case you have to just check from back side if queue back element is less then the nums[I] just remove it and push that nums[I].

Algo:

```
deque<int>dq;
Vector<int>ans;
// first iterate only for very first window
for(int I=0;i<k;i++) {
while(!dq.empty() && nums[I] > dq.back()) dq.pop_back();
dq.push_back(nums[I]);
}
Ans.push_back(dq.front());
//now iterate for remaining sliding windows
for(int I=k;i<nums.ize();i++){
if(dq.front() == nums[i-k]) dq.pop_front();
while(!dq.empty() && nums[I] > dq.back()) dq.pop_back();
dq.push_back(nums[I]);
Ans.push_back(dq.front());
}
Return ans;
```

Time complexity is O(n)

155. Min stack

Approach : Two stack approach

Logic is at the push operation first push the element into the original stack and check if our another stack is empty or second the element is <= second stack's top then push that element into the second stack and for pop if the original stack's top == second stack's top then pop from both stack other wise pop only from the original stack.

Algo:

```
Stack<int>st, min_ele;
Void push(int val){
St.push(val);
if(min_ele.empty() II val <= min_ele.top()){min_ele.push(val);
}
Int pop(){
if(st.top() == min_ele.top()) min_ele.pop();
```

St.pop();
}
Int top() { return st.top(); }
Int getmin() { return min_ele.top(); }
Time complexity is O (1) for all operations;

994. Rotting Oranges

Approach : BFS and Queue
1. Create a function named "is_valid" that takes the current indices (i, j), grid dimensions (n, m), and the grid itself as parameters.
    - Check if the current indices (i, j) are within the grid boundaries and the cell at (i, j) contains a ripe orange (value 1).
    - Return true if the conditions are met, otherwise false.
2. Create a function named "orangesRotting" that takes a 2D vector grid as a parameter.
    - Initialize variables n and m with the dimensions of the grid.
    - Create a queue named "q" to store the coordinates of rotten oranges.
    - Initialize a variable named "fresh" to keep track of the count of fresh oranges.
3. Iterate through each cell in the grid using nested loops.
    - If the current cell contains a rotten orange (value 2), add its indices to the queue.
    - If the current cell contains a fresh orange (value 1), increment the "fresh" count.
4. If there are no fresh oranges (fresh == 0), return 0 since there are no oranges to rot.
5. Initialize a variable named "time" to keep track of the time required for all oranges to rot.
6. Enter a loop that continues until the queue is empty.
    - Get the current size of the queue and store it in "q_size".
    - Initialize a variable named "temp" to keep track of the newly rotten oranges in each iteration.
7. Enter another loop that runs "q_size" times.
    - Dequeue a pair of indices (x1, y1) from the front of the queue.
    - Create two vectors, "a" and "b", to represent the possible movements in four directions (up, down, left, right).
8. Iterate through the four directions using a loop.
    - Calculate the new indices (x, y) based on the current indices (x1, y1) and the direction vectors (a[i], b[i]).
    - Check if the new indices are valid using the "is_valid" function.
        - If valid, update the cell at (x, y) to a rotten orange (value 2), enqueue the new indices to the queue, and increment "temp".

9. After the inner loop ends, check if "temp" is non-zero.
    - If true, increment "time" since at least one orange has been rotten during this iteration.
10. Iterate through the grid one more time to check if there are any remaining fresh oranges.
    - If a fresh orange is found, set "time" to -1 to indicate that not all oranges can be rotten.
11. Return the value of "time" as the result of the function.
Time complexity is O(n * m) + O(n * m) + O(n * m)

901. Online Stock Span

Approach :
1) Two array
Algo:

```
Vector<int> stock, span;
Int next(int price){
stock.push_back(price);
Int n = stock.size()-2;
Int cnt = 1;
while(n>=0 && price >= stock[n]){
Cnt += span[n];
N -= span[n];
}
Span.push_back(cnt);
Return cnt;
}
```

Time complexity is: O (n) space complexity is O(2 n)
2) Stack approach

```
Stack<pair<int,int>> st;
Int next (int price ){
Int span = 1;
while(!st.empty() && price>= st.top().first){
Span += st.top().second;
St.pop;
}
St.push({price, span});
Return span;
```

Time complexity is O (n) space complexity is O (n)

Maximum of minimum for every window size (coding Ninja)

Approach stack:

This problem can also will solved by a doubly ended queue like our previous problem **Sliding Window maximum.** It will take O (n^2) time to find all the maximum for every window size minimum elements. So if you remember then we solved a problem called **Largest Rectangle in Histogram.** In that we funded the left_smaller indices and right smaller indices to calculate the rectangle area. We use same approach to solve this question also. Thing is first we find out all the left_smaller and right_smaller indices for all the elements and then count the window size using formula right_smaller[I] - left_smaller[I] - 1. And update our ans vector. Then we observe that there is some windows are left so for that we iterate from the right side to left and update the ans as max(ans[I] , ans[I+1]).
Algo:

```
Int n = a.size();
Vector<int>left_smaller(n), right_smaller(n), ans(n, INT_MIN);
Stack<int> st;
for(int I=0;i<n;i++){
while(! St.rmpty() && a[st.top()] >= a[I]) st.pop();
if(st.empty()) left_smaller[I] = -1;
Else left_smaller[I] = st.top();
st.push(i);
}
//same for right_smaller
while(! St.rmpty() && a[st.top()] >= a[I]) st.pop();
if(st.empty()) right_smaller[I] = -1;
Else right_smaller[I] = st.top();
st.push(i);
}
//now updating the ans vector
for(int I=0;i<n;i++){
Int len = (right_smaller[I] - left_smaller[I])-1;
ans[len-1] = max(ans[len-1], a[i]);
}
//for remaining windows
for(int I=n-2; I>=0 ;I - -){
ans[I] = max(ans[I], ans[I+1]);
}
Return ans;
```

Time complexity is O (n) + O (n) + O (n) + O (n). Which is overall O(n).

The Celebrity Problem (coding Ninja & GFG)

Approach: simple for loop for coding ninja and stack for GFG

1) Solution For GFG

Here you can first put all the elements in stack from 0 to n. And then take top 2-2 elements from stack and if a will know to b then definitely a is not a celebrity. So pop that and push again b in stack other wise push a in stack. Now the top of stack is celebrity is not defined till now so we have to check the candidacy for that element. So we know that the celebrity is all 1 in its column and all 0 in its row except diagonal. So we check that condition also.
Algo:

```
Stack<int> st;
for(int I=0; I<n; I++) st.push(i);
while(st.size() > 1){
Int a = st.top(); st.pop();
Int b = st.top(); st.pop();
if(knows (a, b)) st.push(b);
Else st.push(a);
}
Int potential_candidate = st.top();
Bool row_check = true;
for(int I=0; I<n ;I++)
if(matrix[potential_candidate] [I]  == 1 && potential_candidate != I) row_check = false;
}
Bool col_check = true;
for(int I=0; I<n; I++){
if(matrix[I] [potential_candidate] == 0) col_check = false;
}
if(row_check == col_check) return potential_candidate;
Return -1;
```

Time complexity is O (n);

2) Solution for coding ninja

In GFG matrix is given but in this only n is given so to solve this problem first we assume 0 as celebrity ans iterate from 1 to n. If that assumed celebrity is know someone than we update the celebrity as I. And also remember that till now it is not defined ki that element is celebrity we have to check some condition for that candidency.
Algo:

```
Int celebrity = 0;
for(int I=0; I<n; I++){
if( knows(celebrity, I)) celebrity = I;
}
for(int I=0; I<n; I++) if( I != celebrity &&( knows(celebrity, I) II !knows(I, celebrity)) return -1;
```

Return celebrity.
Time complexity is O(n).

151. Reverse Words In a String

Approach: simple approach is start from end of the string and if you see the blank space then reverse that word. The crucial thing in this question is how you can handle the extra blank spaces.
Algo:

```
Int n = s.size() -1;
String ans;
while(n>=0){
while(n>=0 && s[n] == ‘ ‘) n - -;
Int end = n;
while(n>=0 && s[n] != ‘ ‘ ) n - -;
Ans += s.substr(n+1, end-n);
while(n>=0 && s[n] == ‘ ‘)n  - -;
if(n>=0) ans += ‘ ‘;
}
Return ans;
```

Time complexity is O (n).

5. Longest Palindromic Substring

Approach: Dynamic programming
if we write the palindrome function and check all the possible substrings of string s then it will take O(n^3) time. So we optimise that O(n^3) to O(n^2) using dynamic programming. We simply create a n*n dp table and fill all the upper diagonal cells. Because every Big palindrome string will produced using each small palindrome string.
Algo:

```
Int n = s.size(), max_len = 0;
String ans;
Vector<vector<int>>dp(n, vector<int>(n));
//we have to fill the upper diagonal so we take the difference between I and j
for(int diff=0; diff<n; diff++){
for(int I=0, j=I+diff; j<n; I++, j++){
//for diagonal elements because each single character will always palindrome
if(I == j) dp[I][j] = 1;
//for 2 characters
Else if(diff == 1){
if(s[I] == s[j]) dp[i][j] = 2;
}
//for more than 2 characters
```

```
Else if(diff > 1){
if(s[I] == s[j] && dp[i+1][j-1] != 0) dp[i][j] = dp[i+1][j-1] + 2;
}
//finding the max_len substring
if(dp[I][j] != 0){
If(j-i+1 > max_len) {
max_len = j-i+1;
Ans = s.substr(I, max_len);
} } };
Return ans;
```

Time complexity is O (n^2) space will also O(n*n)

13. Roman To Integer

Approach : simple linear iteration

Algo:

```
Int ans = 0;
for(int I=0;i<s.size();I++){
for(int i=0;i<s.size();i++){
        if(s[i] == 'I'){
            if(s[i+1] == 'V'){ans+= 4; i++;}
            else if(s[i+1] == 'X') {ans+=9; i++;}
            else ans+=1;
        }
        else if(s[i] == 'X'){
            if(s[i+1] == 'L') {ans+= 40; i++;}
            else if(s[i+1] == 'C') {ans+=90; i++;}
            else ans+=10;
        }
        else if(s[i] == 'C'){
            if(s[i+1] == 'D') {ans+= 400; i++;}
            else if(s[i+1] == 'M') {ans+=900;i++;}
            else ans+=100;
        }
        else if(s[i] == 'V') ans+=5;
        else if(s[i] == 'L') ans+=50;
        else if(s[i] == 'D') ans+=500;
        else if(s[i] == 'M') ans+=1000;
    }
    return ans;
```

Time complexity is O (n).

8. String to Integer (ATOI)

Approach: SImple calculation of digits using its ascii
Algo:

```
Double ans;
Int n = s.size();
Int len = 0;
While (len < n && s[len] == ‘ ‘) len++;
Int sign = 1;
if(s[len] == ‘-‘){
Sign = -1;
Len ++;
}
Else if(s[len] == ‘+’) len++;
while(len <n && s[len]-‘0’ >= 0 && s[len]-‘0’ <= 9){
Ans = (ans*10 + s[len]-‘0’);
if(ans < INT_MIN) ans = INT_MIN;
if(ans < INT_MAX) ans = INT_MAX;
Len ++;
}
Return (int)ans * sign;
```

Time complexity is O(n)

14. Longest Common Prefix

Approach 1). Sort the string array and then check the common characters in first and last index string.
2) take very first string and check that string in entire array.
Algo:

```
String solve(string s1, string s2){
if(s2.size() < s1.size()) solve(s2, s1);
Int good_string = “”;
for(int I=0;i<s1.size();I++){
if(s1[I] == s2[I]) good_string += s[I];
Else break;
}
Return good_string;

String ans = strs[0];
For (auto it: strs) ans = solve(Ans, it);
Return ans;
```

Time complexity is : O(n * min(s1.size, s2.size)).
For First Approach it will be : O(n log n * min(s1.size, s2.size)).

686. Repeated String Match

Approach: first until the a size is lesser then the b append the a with a and when the a size is greater or equal to the b then check if substring is present or not then at the last again append a with a and again check if substring present then return count otherwise -1.

Algo:

String temp = a;
Int count = 1;
while(a.size() < b.size()){
a += temp;
count++;
}
if(a.find(b) != string::npos) return count;
if(a.append(temp).find(b) != string::npos) return count+1;
Return -1;
}

Time complexity is: O (n * m) where n is length of string a and m is length of string b.

What is string::npos ?

string::npos is a static member constant of the **std::string** class in C++. It represents a special value that indicates the absence or failure of a specific string operation.

28. Find the Index of the First Occurrence in a String

Approach: 1) Use find method of c++;

Algo:

Return return haystack.find(needle);

Time complexity is : O(n) in worst case.

Approach 2: KMP algorithm (Important)

Use the KMP algorithm and build the LPS array to avoid the back track of original string.

Algo:

//Building a LPS array
Int n = haystack.size, m = needle.size();
Vector<int> LPS(m);
Int len = 0, i=1;
LPS[0] = 0;
wjile(I <m){
if(needle[I] == needle[len]){
Len ++;

```
LPS[I] = len;
I++; }
Else{
if(len == 0){
LPS[I] = 0;
I++; }
Else{
Len = LPS[len-1];
}
}
//Implementation of actual KMP algorithm
I=0; int j = 0;
while(I < n){
if(haystack[I] == needle[j]){
I++;
j++;
if(j == m) return j-i;
}
Else{
if(j != 0) j = LPS[j-1];
 Else I++;
}
Return -1;
```

Time complexity is O(n).

Minimum Characters required to make a String Palindromic (Interview Bit)

Approach: LPS array

Here thing is create a string which contain the original string and its reversal with separation of One special character. Now calculate the ups array and return the length of original string - last element of ops array.

Algo:

```
String temp = A;
reverse(A.begin(), A.end());
Temp += “$” + A;
Vector<int> LPS(temp.size());
Int len = 0, i=1;
LPS[0];
while(I<temp.size()){
if(temp[I] == temp[len]){
len++;
LPS[I] = len;
```

```
I++;
}
Else{
if(len == 0){
LPS[I] = 0;
I++;
}
Else Len = LPS[len-1];
}
Return A.size() - LPS.back();
```

Time complexity is O(n) which calculating for LPS array.
N will made using (A + A + 1) A - length of string, A - reverse of that string, 1 = special character

242. Valid Anagram

Approach 1) sort those 2 string and checkin equal
2) use hash map

2) Algo:

```
Map<char, int> mp;
for(auto it: s) mp[it]++;
for(auto it: t) mp[it]- -;
for(auto it: mp){
if(mp.second != 0) return false;
}
Return true;
```

Time complexity is : O(n + n + n) and for the first approach it will O(n Log n + log n).

38. Count ans Say

Approach : Use simple approach which can given in the question which is for 1 answer is “1”. You have to add the telemeter also for check the last character of a string.

Algo:

```
if(n == 1) return “1”;
if(n == 2) return “11”;
String s = “11”;
for(int I=3; I<=n; I++){
String temp = “”;
S += “$”;
Int c= 1;
for(int j=1;j<s.size();j++){
if(s[j] != s[j-1]){
```

```
T += to_string(c);
T += s[j-1];
c=1;
}
Else c++;
}
s = t;
}
Return s;
```

Time complexity is O(2 ^ n-2)

165. Compare Version Numbers

Approach: Simply iterate from left to right and take each elements before the . And compare it with version 2 number if n1 greater then n2 return 1. If n2 greater then n1 return -1 otherwise at the end return 0.

Algo:

```
Int I=0, j=0, n1=0, n2=0, n = version1.size(), m = version2.size();
while(I<n II j<m){
n1=n2=0;
while(I<n && version1[I] != '0'){
N1 = n1*10 + (version1[I] - '0');
I++;
}
while(j<m && version2[j] != '0'){
N2 = n2*10 + (version2[j] - '0');
j++;
}
if(n1 > n2) return 1;
Else if( n2 > n1) return -1;
I++;
j++;
}
Return 0;
```

Here if you observed we will make I++ and j++ after each outer while loop. Because the after number there is a '.' So for skip that dot we make I++, j++.

Time complexity is O(n + m).

94. Binary Tree Preorder Traversal

Approach : Root->left->right

Algo:

```
Void inorder_traversal(vector<int>& ans, TreeNode* root){
if(root == NULL) return;
```

```
Ans.push_back(root->val);
inorder_traversal(root->left);
inorder_traversal(root->right);
}
```

144. Binary Tree Inorder Traversal

Approach : left->root->right
Algo:

```
Void inorder_traversal(vector<int>& ans, TreeNode* root){
if(root == NULL) return;
inorder_traversal(root->left);
Ans.push_back(root->val);
inorder_traversal(root->right);
}
```

145. Binary Tree Postorder Traversal

Approach : left->right->root
Algo:

```
Void postorder_traversal(vector<int>& ans, TreeNode* root){
if(root == NULL) return;
inorder_traversal(root->left);
inorder_traversal(root->right);
Ans.push_back(root->val);
}
```

Time complexity is O(n) for all three Traversal

Morris Inorder Traversal (Threaded Binary Tree)

It will Allows to traverse a Binary Tree without any extra stock or Recursion.
Algo:

```
Vector<int> Morris_inorder_traversal(node* root){
Vector<int>inorder;
Node* curr = root;
while(curr != NULL){
if(curr->left == NULL){
inorder.push_back(curr->val);
Curr = curr->right;
}
Else{
node* prev = curr->left;
while(prev->right && prev->right != curr){
Prev = prev->next;
}
```

```
if(prev ->right == NULL){
Prev->right = curr;
Curr = curr->left;
}
Else{
Prev->right = NULL;
Inorder.push_back(curr->val);
Curr = curr->right;
}
}
}
```

Time complexity is Amortised (n) + O(n) which is almost O(n).

Morris Preorder Traversal (Threaded Binary Tree)

It is almost similar to inordere traversal but the thing is that, when you reach the root and you mark it as thread. At that moment please print the curr value.

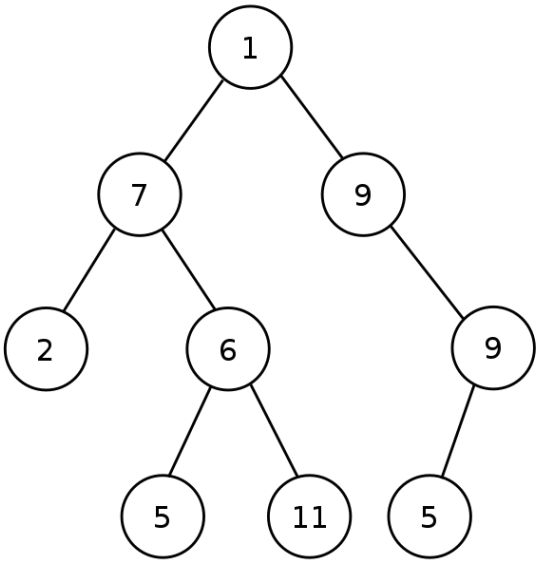
Algo:

```
Vector<int> Morris_inorder_traversal(node* root){
Vector<int>inorder;
Node* curr = root;
while(curr != NULL){
if(curr->left == NULL){
inorder.push_back(curr->val);
Curr = curr->right;
}
Else{
node* prev = curr->left;
while(prev->right && prev->right != curr){
Prev = prev->next;
}
if(prev ->right == NULL){
Prev->right = curr;
Inorder.push_back(curr->val);
Curr = curr->left;
}
Else{
Prev->right = NULL;
Curr = curr->right;
}
}
}
```

Time complexity is Amortised (n) + O(n) which is almost O(n).

Left View Of Binary Tree (GFG)

Approach:



In the given tree if we see from left side then we able to see the 1-7-2-5. At the first look it will a tricky question. But if we take this question as level order traversal then it will become very easy question. To solve the question we just traverse from root and maintain a ans vector. And at each level we increase the level if level is >= ans.size then we put that value in our ans.

Algo:

```
Void solve(TreeNode* root, int level, vector<int>& ans){
if(root == NULL) return;
if(level >= ans.size()) ans.push_back(root->data)
solve(root->left, level+1, ans);
solve(root->right, level+1, ans);
}
```

Time complexity is O(n) ans space will also a O(n) in worst case.

Right View Of Binary Tree (GFG)

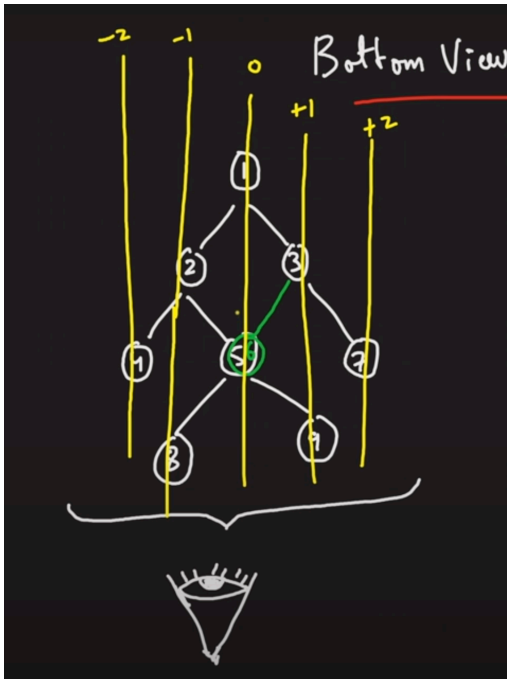
Right view of binary tree is also same as the above question only changes is you just first call the root->right and then root->left for recursion.

Algo:

```
Void solve(TreeNode* root, int level, vector<int>& ans){
if(root == NULL) return;
if(level >= ans.size()) ans.push_back(root->data)
solve(root->right, level+1, ans);
solve(root->left, level+1, ans);
}
```

Bottom View Of Binary Tree (GFG)

Approach:
Logic is that make the level traversal and distribute the tree vertically and the bottom element of your distribution put it into the ans vector. Use queue and map to keep track that. We use map because we want answer from left to right and map will give the answer in sorted order. We also makes the line like -2, -1, 0, 1, 2 etc which is shown in the figure.



Algo:

```
Vector<int> ans;
Map<int, int>mp;
queue<pair<Node*, int>>q;
q.push({root, 0});
while(!q.empty()){
Auto it = q.front();
Q.pop();
Node* node = it.first;
Int line = it.second;
mp[line] = node;
if(node->left != NULL) Q.push({node->left, line-1});
if(node->right != NULL) q.push({node->right, line+1});
}
for(auto it: mp) ans.push_back(it.second);
Return ans;
```

Time complexity is O(n). Space will also O(n).

Top View Of binary Tree(GFG)

Approach : in previous question for bottom view of binary tree you can override the elements in the line but according to this question logic is do not override that elements. Remaining logic is same.

Algo:

```
Vector<int> ans;
Map<int, int>mp;
queue<pair<Node*, int>>q;
q.push({root, 0});
while(!q.empty()){
Auto it = q.front();
Q.pop();
Node* node = it.first;
Int line = it.second;
if(!mp.count(line) mp[line] = node->data;
if(node->left != NULL) Q.push({node->left, line-1});
if(node->right != NULL) q.push({node->right, line+1});
}
for(auto it: mp) ans.push_back(it.second);
Return ans;
```

Time complexity is O(n). Space will also O(n).

Tree Traversals (Coding Ninja)

Approach:

In each recursion we had just pushed elements of inorder, preorder and post order. But to traverse all three types combined you have to use three arrays simeltenously at each recursion.

Algo:

```
void traversal(BinaryTreeNode<int> *root, vector<int>&inorder,
vector<int>&preorder, vector<int>&postorder){
    if(root == NULL){
        return;
    }
    preorder.push_back(root->data);
    traversal(root->left,inorder,preorder,postorder);
    inorder.push_back(root->data);
    traversal(root->right,inorder,preorder,postorder);
    postorder.push_back(root->data);
}
vector<vector<int>> ans;
    vector<int> inorder, preorder, postorder;
    traversal(root,inorder,preorder,postorder);
```

```
    ans.push_back(inorder);
    ans.push_back(preorder);
    ans.push_back(postorder);
    return ans;
```

Time complexity is O(n).

987. Vertical Order Traversal Of a Binary Tree

Approach: idea is same as previous question but here we do not override the elements in map or storing the previous element at same line or anything. But here the new task is you have to store all the elements at each level in sorted order so for that we use a map with multisite and for queue we use two elements called level and line.

Algo:

```
Vector<vector<int>> ans;
Map<int, map<int, multiset<int>>> mp;
Queue<pair<TreeNode*, pair<int,int>>>q;
q.push({root, {0,0}});
while(! Q.empty()){
Auto it = q.front();
q.pop();
TreeNode* node = it.first;
Int line = it.second.first;
Int col = it.second.second;
mp[line][col].insert(node->val);
if(node->left != NULL){
q.push({node->left, {line-1, col+1}});
}
if(node->right != NULL){
Q.push({node->right, {line+!, col+1}});
}
}
for(auto it: mp){
        vector<int> col;
        for(auto sec: it.second){
                for(auto third: sec.second){
                        col.push_back(third);
                }
        }
        ans.push_back(col);
}
Return ans;
```

Time complexity is O(n * logn). n for traversal and logn for sorted multiset.

Path in a Tree(Coding Ninja)

Approach: Use inorder, preorder, postorder any of traversal and for recursion please use the boolean function so if chid is return true or false according to that we push and pop the node values in our ans vector.
Algo:

```
Bool find_path(TreeNode* root, int x, vector<int>& ans){
if(root == NULL){
Return true;
}
ans.push_back(root->data);
if(root->data == x) return true;
if(find_path(root->left, x, ans) ll find_path(root->right, x, ans)) return true;
Ans.pop_back();
Return false;
}
```

Call this function and return the ans vector.
Time complexity is O(n) in worst case.

257. Binary Tree Paths

Approach: Recursion ans Backtrack

```
Void find_path(TreeNode* root, vector<string>&path, string s){
if(!root->left && !root->right){
path.push_back(s + to_string(root->val));
}
if(root->left != NULL){
find_path(root->left, path, s+ to_string(root->val) + “->”);
}
if(root->right != NULL){
find_path(root->right, path, s+ to_string(root->val) + “->”);
}
```

Time complexity O(n)

662. Maximum Width of binary tree

Approach: Level Order Traversal
This is a very tricky question if you don’t know the logic of segment tree. Here the logic is if your left most node is contains the value 2 and your right most node contains the value 5 then your answer will 5-2+1 as your width. So how to approach it if you give indexing to all the node from 0 to n level wise then it was possible. Formula to give the indexing a node at each level is:
2*I +1 for left child and 2*I+2 for right child. But,but,but.. remember one thing each and every time you made a multiplication operation to find a index for that particular node which might creates a overflow error at some point.

So, the question is how to handle such overflow?
Ans is give the 0 index for very first(root) node then for second level there are only two node so gives 1 and 2 index. Now for third level start from 0 again. And continuously do this process for every node. Which can solve the overflow problem.
Now another question is above logic is work only when the left most and right most child is present. What if we don't have an right most and left most child? So for that there is a formula for left child is curr_index*2 + 1 and for right side is curr_index*2+2. Now, let's relate above logic with our question. So, to solve our question we maintain a queue which contains the level and index at each node.
Algo:

```
Queue<pair<TreeNode*, long long>> q;
Int width = 0;
q.push({root, 0});
while(!q.empty()){
Int size = q.size();
Int min_level = q.front()second;
Int first, last;
for(int I=0;i<size;i++){
Long long curr_index = q.front().second - min_level;
TreeNode* node = q.front().second;
Q.pop();
if(I == 0) first = curr_index;
if(I == size-1) last = curr_index;
if(root->left) q.push({root->left, curr_index*2+1)
if(root->right) q.push({root->right, curr_index_2+2);
}
Width = max(width, last-first + 1);
}
```

Time complexity is O(n) space is also O(n)

102. Binary Tree Level Order Traversal

Approach: Since we solved few level order traversal question it is a easy question.

```
if(!root) return {};
Queue<TreeNode*>q;
Vector<vector<int>>ans;
q.push(root);
while(!q.empty()){
Vector<int> temp;
Int size = q.size();
```

```
for(int I=0;i<size;i++){
TreeNode* node=q.front();
q.pop();
Temp.push_back(node->val);
if(!node->left) q.push(node->left);
if(!node->right) q.push(node->right);
}
ans.push_back(temp);
```

Time complexity is O(n). Space is also O(n)

104. Maximum Depth of Binary tree

Approach: Level order traversal

Algo:

```
Queue<TreeNode*>q;
Int level=0;
q.push(root);
while(!q.empty()){
Int size = q.size();
for(int I=0;i<size;i++){
TreeNode* node = q.front();
q.pop();
if(node->left) q.push(node->left);
if(node->right) q.push(node->right);
}
level++;
}
Return level;
```

Time complexity O(n) space is also O(n)

543. Diameter of Binary tree

Approach if you find first left maximum and then find the right maximum then it will takes the O(n^2) Time. So now the question is how to convert it into the O(n). So for that we made a int function in which we first find the left and right height then we will find the diameter using maximum. And in function please return the 1 + max(left_height, right_height);

Algo:

```
Int solve(TreeNode* root, int diameter){
if(!root) return 0;
Int left_height = solve(root->left, diameter);
Int right+_height = solve(root->right, diameter);
Diameter = max(diameter, left_height+right_height);
Return 1 + max(left_height, right_height);
```

}
Time complexity is O(n);

110. Balanced Binary Tree

Approach: Again if you map this this question with our find height question then it is very easy. All you need to have just return -1 at each stage if the right and left height absolute difference is >1 or the left or right height does not return the value -1. Again this questions are is if and only if you are very good at recursion.
Algo:
Int solve(TreeNode* root){
if(!root) return 0;
Int left_height = solve(root->left);
Int right_height = solve(root->right);
if(‘left_height = -1 ll right_height == -1) return -1;
if(abs(left_height - right_height) == -1) return -1;
Return 1+ max(right_height, left_height);
}
In main method:
if(solve(root) == -1) return false;
Return true;
Time complexity is O(n)/

236. Lowest Common Ancestor of a Binary Tree

Approach: the very first approach occurs in our mind is naive approach, which is first find the path from root to p and root to q and store those two path into the two different vectors then iterate that two vectors parallelly. If path1[l] != path2[l] then break the loop and return the either path1[l] or either path2[l]. This is a good solution but it will take the total O(n + n + min(path1, path2)) time and it will also take a space of O(n + n) to store the two node paths.
So the question is how to improve it?
Answer is, iterate all the nodes recursively if the node is == p or q then return that node. If left is null then return right, if right is null then return left else both are not null then return that node itself.
Algo:
If (root == NULL ll root == p ll root == q) return root;

TreeNode* left = lowestCommonAncestor(root->left, p, q);
TreeNode* right = lowestCommonAncestor(root->right, p, q);
if(right == NULL){
Return left;
}

```
Else if(left == NULL){
Return right;
}
Else{
Return root;
}
```

Time complexity is O(n)

100. Same Tree

Approach: 2 -tree Preorder traversal at same time.

Algo:

```
bool identical(TreeNode* p, TreeNode* q){
if(!p && !q){
Return true;
}
if((!p && q) ll (p && !q)){
Return false;
}
if(!identical(p->left,q->left)){
Return false;
}
if(!identical(p->right, q->right)){
Return false;
}
Return true;
}
```

Time complexity is O(n) because we traverse two tree at same time.

103. Binary Tree Zigzag level order traversal

Approach: level order traversal with boolean flag

You have to just maintain a flag either you have to push the value at front to the vector or back to the vector otherwise entire logic of level order traversal will be same.

Algo:

```
if(!root) return {};
Queue<TreeNode*>q;
Vector<vector<int>>ans;
Bool zigzag = true;
Q.push(root);
while(!q.empty()){
Int size = q.size();
Vector<int> temp(Size);
```

```
for(int I=0;i<size;i++){
TreeNode* node = q.front();
Q.pop();
Int idx;
if(zigzag == true) idx = I;
Else idx = size -1 -I;
temp[idx] = node->val;
if(node->left) q.push(node->left);
if(node->right) q.push(node->right);
}
if(zigzag == true) zigzag = false;
Else zigzag = true;
Ans.push_back(temp)
}
Return ans;
```

Time complexity is O(n) space is also O(n).

Boundary Traversal of Binary Tree (Coding Ninja)

Approach: Devide entire problem into three part
There are three logics in entire problem 1) find all the left tree nodes excluding leaf node.
2) find all the leaf nodes from left to right. for that please use inorder traversal instead of level order traversal because there is a possibility in that in-between some leaf nodes are there.
3) find the right part nodes of a tree excluding leaf node in reverse order because in question it is clearly mention that we have to find answer in anti clock wise.
Algo:

```
Bool is_leaf(TreeNode* root){
Return (root->left = NULL && root->right == NULL);
}
Void find_rightpart(TreeNode* root, vector<int>& ans){
TreeNode* curr = root->left;
while(curr){
if(!is_leaf(curr)  Ans.push_back(curr->data);
if(curr->left) curr = curr->left;
Else curr = curr->right;
}
Void find_rightpart(TreeNode* root, vector<int>& ans){
Stack<int>st;
TreeNode* curr;
While(curr){
```

```
if(!is_leaf(curr) st.push(curr->data);
if(curr->right) curr = curr->right;
Else curr = curr->right;
}
while(!st.empty()){
Ans.push_back(st.top());
St.pop();
}
Void find_leaf(TreeNode* root, vector<int>&ans){
if(is_leaf(root)) ans.push_back(root->data);
if(root->left) find_leaf(root->left, ans);
if(root->right) find_leaf(root->right, ans);
}
```

In main method first check if root node is null then return empty vector. if root is not leaf node then push it into the ans vector. And then call those three function in order find_left -> find_leaf -> find->right.

Time complexity is O(height + height + n) space is O(n). 2 times height for find the left and right part and O(n) for find the leaf nodes using inorder traversal.

124. Binary tree Maximum Path Sum

Approach: same as the maximum length path and diameter of binary tree. If you remember this 2 question in which we used a backtracking. In the diameter question we returned a 1 + max(left height, right height). And in the maximum depth binary tree question we returned 1 + max(left height, right height). This question is exact same as that two questions. Only you have to add some 2 or three line logic. Here is that additions.

1) for left and right sum left/right sum = max(0, solve(root->left/right));

Here we take a maximum of 0 and that solve function because what is the node contains the negative value. So to handle that constraint we take the maximum of 0 and ans of that solve function

2) for max_sum: max_sum = max(max_sum, left_sum+right_sum_root->val)

3) for return a value max(left_sum, right_sum) + root->val;

Algo:

```
Int solve(TreeNode* root, int &max_sum){
if(!root) return 0;
Int left_sum = max(0, solve(root->left, max_sum));
Int right_sum = max(0, solve(root->right, max_sum));
max_sum = max(max_sum, left_sum + right_sum + root->val);
Return root->val + max(left_sum, right_sum);
}
```

Time complexity is O(n) space is also O(n).

105. Construct a binary tree from inorder and preorder traversal

Approach: recursion
We know that from the inorder and preorder unique binary tree is possible and it is also known that, inorder will follow: left -> root -> right and preorder will follow the root->left->right so we take each root from preorder and make the left and right parts from inorder because inorder will contains the left->root->right and call the recursive function for each part. We also maintain a Hashmap so we can easily get the index of root value from inorder array.
Algo:
TreeNode* createTree(vector<int>&inorder, vector<int>& preorder, int instart, int inend , int prestart, int preend, map<int, int>mp){
if(prestart > pretend || instart > inend) return NULL;
TreeNode* root = new TreeNode(preorder[prestart]);
Int right_part = mp[root->val];
Int left_part = right_part - instart;
Root->left = createTree(inorder, preorder, instart, right-1, prestart+1, prestart + left_part, mp);
Root->right = createTree(inorder, preorder, instart+left_part, inend, right_part+1, preend, mp);
Return root;
}
In main function;
Map<int, int>mp;
for(int I=0;i<inorder.size();I++) mp[inorder[i]] = i;
Return createTree(inorder, preorder, 0, inorder.size()-1, 0, preorder.size()-1, mp);
Time complexity Is O(n * logn) logn for search a element in Hashmap.

106. Construct Binary Tree from Inorder and Postorder Traversal

Approach: Recursion
It is a exact same logic as our previous question you have to just change that four pointers indexes. And remember one thing also is for very first node please enter the last element of postorder array. Because postorder traversal will follows the left->right->root.
Algo:
TreeNode* createTree(vector<int>&inorder, vector<int>& postorder, int instart, int inend , int poststart, int postend, map<int, int>mp){
if(poststart > postend || instart > inend) return NULL;
TreeNode* root = new TreeNode(postorder[postend]);
Int right_part = mp[root->val];

```
Int left_part = right_part - instart;
Root->left = createTree(inorder, preorder, instart, right_part-1, poststart,
prestart + left_part - 1, mp);
Root->right = createTree(inorder, preorder, right_part+1, inend, poststart +
left_part, postend-1, mp);
Return root;
}
In main function;
Map<int, int>mp;
for(int I=0;i<inorder.size();I++) mp[inorder[i]] = i;
Return createTree(inorder, preorder, 0, inorder.size()-1, 0, postorder.size()-1,
mp);
```

Time complexity is O(n * log n) logn for search a element in Hashmap

101. Symmetric Tree

Approach: 1) level order traversal
2) Recursion

In he first approach we can traverse the tree level by level and store those nodes into the array. And at each level we check whether that array is mirror itself or not using another function. But this will take lot of extra space. So we follow second approach.

In recursion we take to arguments which is root1 and root2 if both are null then we return true. Otherwise we check the specific conditions and then call the recursion two time with && operation to check the symmetry tree.

Algo:

```
Bool is_mirror(TreeNode* root1, TreeNode* root2){
if(root1 == NULL && root2 == NULL) return true;
if(root1 && root2 && root1->val == root2->val){
Return is_mirror(root1->left, root2->right) && is_mirror(root1->right, root2-
>left);
}
Return true;
}
In main method:
Return is_mirror(root, root);
```

Time complexity is O(n).

114. Flatten Binary Tree to Linked List

Approach: Morris Traversal concept

If you use the single stack or vector and make the preorder traversal then it is very easy question you can able to solve in O(n) time and O(n) space. But the question is how to solve in O(1) space?

Ans is if you remember our Morris traversal concept(threaded binary tree) in which we can connecting the left subtrees right most node with the root node. Here changes are instead of root node you connect the left subtree's rightmost node with the root's right node.
Algo:

```
TreeNode* curr = root;
while(curr != NULL){
if(curr->left != NULL){
TreeNode* prev = curr->left;
while(prev->right){
Prev = prev->right;
}
Prev->right = curr->right;
Curr->right = curr->left;
Curr->left = NULL;
}
Curr = curr->right;
}
```

Time complexity is O(n) space is O(1).

Mirror Tree (GFG)

Approach: recursion.
Simply just take a root node in each recursion call and swap the left and right node of tree.

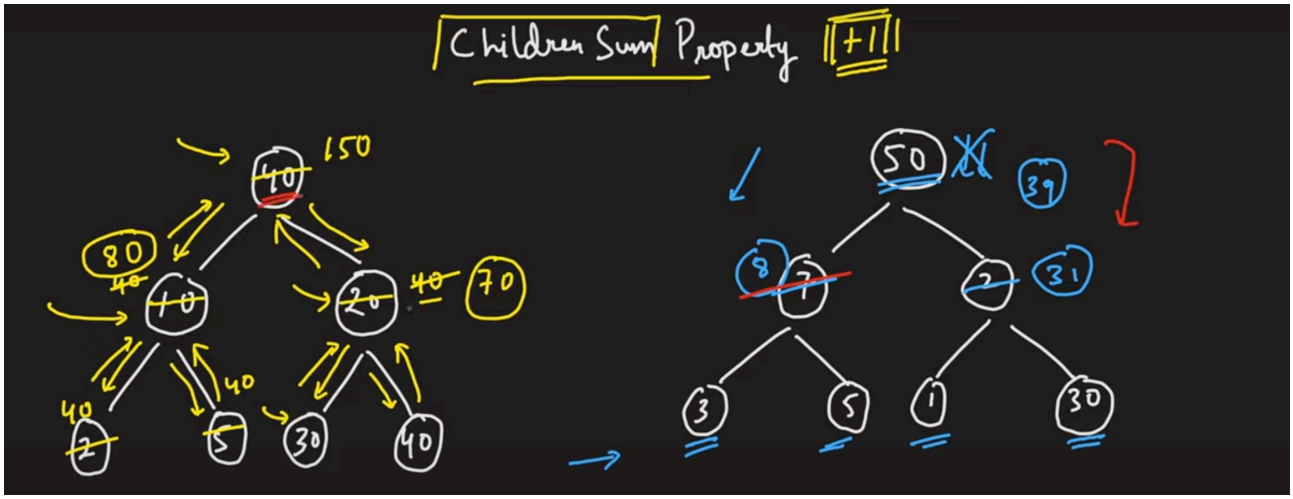
```
Void solve(Node* node){
if(!node) return;
solve(node->left);
solve(node->right);
Node* prev = node->right;
Node->right = node->left;
Node->left = prev;
}
```

Time complexity is O(n)

Children Sum Property (Coding Ninja)

Approach: Recursion and Backtracking
One interesting thing in this question is you can make the value of binary tree anything like you want but the simple constraint is it will holds the property of children sum property. At the first look of this question we say that it is a easy question but this question is not that much easy. If you follow the simple approach and you go down to the tree and check those conditions then it will create a shortage of sum for root node like in given image's right side tree. So

here we apply the approach is, when you go down if the child sum is not > then the root node then make that child value same as root node. And you again come back then make to root node value by taking the sum of it's child nodes. Remember it will create some large numbers but still it will solve the question.



Algo:
if(!root) return;
Int child_sum = 0;
if(root->left) child_sum += root->left->data;
if(root->right) child_sum += root->right->data;
if(child_sum >= root->data) root->data = child_sum;
Else{
if(root->left) root->left->data = root->data;
if(root->right) root->right->data = root->data;
}
solve(root->left);
solve(root->right);
Int total_sum = 0;
if(root->left) total_sum += root->left->data;
if(root->right) total_sum += root->right->data;
if(root->left || root->right) root->data = total_sum;
}
Time complexity is O(n)

116. Populating Next Right Pointers in Each Node

Approach: Level Order Traversal
Store all the nodes in ans vector and at each level assign all the ith node next pointer to I+1 node.
Algo:

```
if(!root) return NULL;
        queue<Node* >q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            vector<Node*> ans;
            for(int i=0;i<size;i++){
                Node* temp = q.front();
                q.pop();
                ans.push_back(temp);

                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
            }
            if(ans.size() == 2){
                Node* first = ans[0];
                Node* second = ans[1];
                first->next = second;
            }
            else if(ans.size()>2){
                for(int i=0;i<ans.size()-1;i++){
                    Node* first = ans[i];
                    Node* second = ans[i+1];
                    first->next = second;
                }
            }
        }
        return root;
```

Time and space complexity will O(n)

700. Search In Binary Search Tree

Approach : we know that in BST every child node has lesser then its parent and every right node will greater then it's paren so we use this property ton search element in O(h) time.

Algo:

```
if(!root) return NULL;
if(root->val == val) return root;
if(val < root->val) return searchBST(root->left, val);
Else return searchBST(root->right, val);
Return NULL;
```

Time complexity is O(h) where h is height of the tree

108. Convert Sorted Array to Binary Search Tree

Approach: treat this question as binary search problem
Algo:
TreeNode* insertBST(vector<int>& nums, int start, int end){
if(start > end) return NULL;
Int mid = (start + end)/2;
TreeNode* root = new TreeNode(nums[mid]);
Root->left = insertBST(nums, start, mid-1);
Root->right = insertBST(nums, mid+1, end);
Return root;
Time complexity is O(n)

1008. Construct a Binary search tree from given preorder traversal

Approach: 1) we know that the inorder traversal of BST is always in sorted order. What if I sort the preorder then it is a simple question to solve using preorder and inorder traversal. This approach is accepted but it will take O(n * logn) time as well it takes the O(n + n) extra space to store inorder and Hashmap that inorder array.
2). We all know that in BST left subtrees is always lesser then root and right subtree is always greater then the root. Why we not use this property to solve this question. We start from the very first index and goes until the element is greater then the current element. To solve this question we also take the one variable called upper bound. And when you are going to left then upper bound is your root value and when you are going to right your upper bound does not changed because by default your upper bound is parent node.
For example: if you have array like [8 5 1 7 10 12] when you reach 7 and if you go the left then your upper bound is 5 and go to the right then upper bound is 5 and like wise.
Remember initially you take the upper bound as INT_MAX.
Algo:
TreeNode* createBST(vector<int>& preorder, int I, int u_bound){
if(I == preorder.size() || preorder[I] > u_bound) return NULL;
TreeNode* root = new TreeNode(preorder[I]);
I++;
Root->left = createBST(preorder, I, root->val);
Root->right = createBST(preorder, I, u_bound);
Return root;}
//In main function
Int I = preorder.size, u_bound = INT_MAX;
Return createBST(preorder, I , u_bound);
Time complexity is O(n)

98. Validate Binary Search Tree

Approach: 1) make inorder traversal of tree if in order traversal of tree is sorted then return true otherwise false. In this the time complexity is O(n + n) and extra space is also O(n) to store the inorder traversal.
2) instead of checking entire tree what if I check each node and gives it a two range low and high if that node is between range then it will valid otherwise not.
Algo:

```
Bool BSTvalid(TreeNode* root, long long lower_range, long long
higher_range)
if(!root) return true;
if(root->val <= lower_range ll root->val >= higher_range) return false;
Return (BSTvalid(root->left, lower_range, root->val) && BSTvalid(root->right,
root->val, higher_range));
}
//In main function
Return BSTvalid(root, LONG_MIN, LONG_MAX);
```

Time complexity is O(n) and it will not take any extra space.

235. Lowest Common Ancestor of a Binary Search Tree

Approach: if you remember the LCA question of Binary tree, in that we can used the recursion and backtracking approach to find the intersect point of two node. But there is slite difference in this question because instead of BT it is a BST so it is a possible we find intersect point in O(h) instead of O(n) time.
Algo:

```
if(!root) return NULL;
TreeNode* curr = root;
while(curr){
Int data = curr->val;
if(data < p->val && data < q->val){
Curr = curr->right;
}
Else if(data > p->val && data > q->val){
Curr = curr->left;
}
Else return curr;
}
```

Time complexity is O(h) where h is height of tree and space is O(1).

Predecessor and Successor in BST (GFG)

Approach : 1) use simple recursion with O(n) time
2) use the BST property and iterative approach to solve in O(h) time.

```
Algo;
While(root){
if(root->key == key){
if(root->left){
Suc = root->left;
While(suc->right) suc = suc->right;
}
if(root->right){
Pred = root->right;
while(pred->left) red = pred->left;
}
Break;
}
Else if(root-key < key){
Pred = root;
Root = root->right;
}
Else{
Suc = root;
Root = root->left;
}
```

Time complexity is O(h)

Floor In BST (Coding Ninja)

Approach: if you able to map this question with binary search then it is a very easy question.

Algo:

```
TreeNode* floor;
while(root){
if(root->val == X) return root->val;
Else if(X < root->val) root = root->left;
Else{
Floor = root;
Root = root->right;
}
if(floor) return floor->val;
Return -1;
```

Time complexity is O(h) where h is the height of the tree.

Ceil from BST (Coding Ninja)

Approach: binary search with iterative approach.

```
 BinaryTreeNode<int>* ceil = NULL;
```

```
  while(node){
     if(node->data == x){
          return node->data;
     }
     else if(x < node->data){
          ceil = node;
          node = node->left;
     }
     else{
         node = node->right;
     }
  }
   if(ceil) return ceil->data;
   return -1;
```

Time complexity is O(h)

230. Kth Smallest Element in a BST

Approach: we know that in BST, inorder traversal will gives always sorted array. So why we not use that property. We write a recursion and maintain one element called k, if k == 0 then return that answer.

Algo:

```
Void solve(TreeNode*root , int &k){
if(!root) return;
solve(root->left, k);
k—;
if(k==0) ans = root->val;
solve(root->right, k);
}
//In main function
Int ans;
solve(Root, ans);
Return ans;
```

Time complexity is O(n) and space is O(1).

Kth largest element in BST (GFG)

Approach:

Very first approach comes in our mind is count the number of nodes and find the position which is no_of_nodes - K and return that element using inorder traversal. This will take O(n + n) time to execute. But can we reduce this time ? So, ans is yes. The reverse operation of inorder is Right -> root -> left if we travers the tree in revers inorder then we able to find kth largest element in O(n) time.

Algo:
Void solve(Node* root, int &ans, int &K){
if(!root) return;
solve(root->right, ans, K)
K - -;
if(K == 0){
Ans = root->data;
Return;
}
solve(root->left, ans, K);
}
//in main function
Int ans;
solve(root, ans, K);
Return ans;
Time complexity is O(n)

173. BST Iterator

Approach: use stack
In this approach you can start with root and push all the extreme left nodes in the stack. In the next function first take the top node of stack and if that node has right node and again call the push_all function to again push all the extreme left nodes in the stack. And for the hasNext function simply return if stack is empty or not.
Algo:
Stack<TreeNode*> st;
Void push_all(TreeNode* root){
while(root){
St.push(root);
Root = root->left;
} }
//int the BSTIterator constructor
BSTIterator(TreeNode* root) push_all(root);

Int next(){
TreeNode* temp = st.top();
St.pop();
if(temp->right) push_all(temp->right);
Return temp->val;
}
// in hasnext function
Return !st.empty();

Time complexity: on and average push_all function will take O(h) time and next function will takes the O(h) because we call the push_all in that function also and hasNext will take O(1). So the overall time complexity is O(h) ans space is also O(h).

653. Two Sum IV - Input is a BST

Approach: 1) make the inorder traversal and store all the elements in a ans vector. Now the given problem is same as our famous two sum problem. You can solve it using the two pointer approach.
2) 2nd approach is use a set to store all the unique elements in given BST. And also write a recursive function which can return true if the k-root->val is found in the set.
3) Use the BST iterative approach: in this approach we utilise the concept of Iterative BST.
Let's understand this concept step by step
First in Iterative BST we find the inorder traversal with left->root->right so in this we find that which gives sorted array.
Second thing is we find the reverse inorder traversal which gives the decreasing sorted array.
Now we have our Radimed search spaces with increasing and decreasing order. Now we treat this problem as our famous 2 sum problem using two pointer.
Algo:

```
Stack<TreeNode*> st;
Bool reverse = true; // to take our search space in O(n) only
Class BSTIterator{
BSTIterator (TreeNode* root, bool is_reverse){
Reverse = is_reverse;
push_all(root);
}
Int next(){
TreeNode* temp = st.top();
St.pop();
if(reverse == true) push_all(temp->left);
Else push_all(temp->right);
}
Void push_all(TreeNode* root){
while(root){
St.push(root);
if(reverse == true) root = root->right;
Else root = root->left;
} };
```

```
Class Solution{
Bool findTarget(TreeNode* root, int k){
if(!root) return false;
// let's create two objects called l and r using reverse true and false;
BSTIterator l(root, false);
BSTIterator r(root, true);
// let's treat this question as our famous two sum problem
Int l = l.next();
Int j = r.next();
While (l < j){
if(l+j == k) return true;
Else if(l+j < k) l = l.next();
Else j = r.next();
}
Return false;
```

Time complexity is O(n) bur very importantly space complexity is almost O(h).

1373. Maximum Sum BST in Binary Tree

Approach: use a data structure with 4 variables called: isBST, max_node, min_node, max_sum with the use of postorder traversal.
In this logic is at each node you have to check if the current nodes all the left and right child nodes will holds the property or not if it is then you can update your max_sum ans other wise not. And remember to find whether given subtree is BST or not we use a left subtree's maximum element and right subtrees min element if root is lies between this two then the given subtree is BST.
Algo:

```
Class NodeValue{
Int max_node;
Int min_node;
Int max_sum;
Boll isBST;
};
class solution{
NodeValue solve(TreeNode* root, int &ans){
if(!root) return {INT_MAX, INT_MIN, 0, true};
NodeValue left = solve(root->left, ans);
NodeValue right = solve(root->right, ans);
NodeValue curr;
Curr.max_node = max(root->val, right.max_node);
curr.min_node = min(curr->val, left.min_node);
```

```
if(left.isBST && right.isBST && (left.max_node < root->val) &&
(right.min_node > root->val)){
Curr.isBST = true;
}
Else {
Curr.isBST = false;
}
Curr.max_sum = left.max_sum + right.max_sum + root->val;
if(curr.isBST){
Ans = max(Ans, curr.max_sum);
}
Return curr;
}
Int maxSumBST(TreeNode* root){
Int ans = 0;
solve(root, ans);
Return ans;
```

Time complexity is O(n) ans space complexity will be a O(1).

297. Serialize and Deserialize Binary Tree

Approach: for converting a tree to string use the level order traversal and for converting the string to tree use the queue. For ex first push the level 1 element in queue then level 2 and like wise. and we made a level order traversal so it can makes our work easy to assign the left and right child to the nodes.

Algo:

```
 string serialize(TreeNode* root) {
        if(!root) return "";
       queue<TreeNode*> q;
       string s="";
       q.push(root);
       while(!q.empty()){
             TreeNode* temp = q.front();
             q.pop();
             if(temp == NULL) s += "$,";
             else s += to_string(temp->val) + ",";
             if(temp != NULL){
                 q.push(temp->left);
                 q.push(temp->right);
             }
       }
       return s;
```

```
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data == "") return NULL;
        stringstream s(data);
        string str;
        queue<TreeNode*> q;
        getline(s, str,',');
        TreeNode* root = new TreeNode(stoi(str));
        q.push(root);
        while(!q.empty()){
            TreeNode* temp = q.front();
            q.pop();

            //processing the left subtree
           getline(s, str, ',');
            if(str == "$") temp->left = NULL;
            else{
                TreeNode* temp1 = new TreeNode(stoi(str));
                temp->left = temp1;
                q.push(temp1);
            }

           //processing the right subtree
            getline(s, str, ',');
            if(str == "$") temp->right = NULL;
            else{
                TreeNode* temp2 = new TreeNode(stoi(str));
                temp->right = temp2;
                q.push(temp2);
            }
        }
        return root;
```

Here we used a stringstream it will a default iterator which takes the three arguments first is string, second is ans string, and third is separator. For example our string is 1,20,$,3,4, then it will gives first 1 20 $ and like wise. For that syntax is:
stringstream(original string);
getline(original string, resultant string, separator) which is ‘,’ in our case.
Time complexity is O(n) and space will also O(n) because we used a extra queue to store all the nodes.

Convert a Given Binary Tree to Doubly Linked List (Coding Ninja)

Approach: In-order traversal
Use the inorder traversal and take 2 dummy node called head and prev. At first time you goes to the recursion update the head as root. And each recursion make the pre node as root node. Now after first recursion make root->left = prev and prev->right = root.

```
Void solve(TreeNode* root, TreeNode* &head, TreeNode* &prev){
if(!root ) return;
solve(root->left, head, prev);
if(prev == NULL){
Head = root;
}
Else{
Root->left = prev;
Prev = root;
}
Prev = root;
solve(root->right, head, prev);
}
//int main method
TreeNode* head = NULL, prev = NULL;
solve(root, head, prev);
Return head;
```

Time complexity is O(n)

295. Find Median from Data stream

Approach: Min and Max heap
This is hard as well very tricky question but solution is also super simple. Lets example we have numbers like 4, 1, 3, 2, 6, 5 and all the data points will occurs in continues streaming so how to handle it. If we count the median at each time then obviously it will not a feasible solution. So. Here we maintain two heaps called min_heap and max_heap and store that continues stream of data points. And for median if our input data points are odd then we simple return the top element of heap which contain higher size then another otherwise we will take top of elements from both heaps and return the average.
Algo:
—> At the top create two heaps called:

```
priority_queue<int> max_heap;
priority_queue<int, vector<int>, greater<int>> min_heap;
```

—> now in the addendum function add the number into our heap using below conditions.

```
if(min_heap.empty() && max_heap.empty()) max_heap.push(num);
Else{
if(max_heap.top() < num) min_heap.push(num);
Else max_heap.push(num);
}
```

—> now writes the logic in find mean function

```
Double FindMean(){
if(min_heap.size() == max_heap.size())
return ((double)min_heap.top() + (double)max_heap.top()) / 2.0;
Else if(max_heap.size() == min_heap.size() +1)
return (double)max_heap.top();
Else if(max_heap.size()+1 == min_heap.size())
return (double)min_heap.top();
Else if( max_heap.size() > min_heap.size()+1){
Int ele = max_heap.top();
min_heap.push(ele);
max_heap.pop();
Return FindMedian();
}
Else if( min_heap.size() > max_heap.size()+1){
Int ele = min_heap.top();
max_heap.push(ele);
min_heap.pop();
Return FindMedian();
}
Return 0.0;
}
```

Time complexity will be O (log n) because we used a heap data structure
Space complexity is O (n) because we store n elements in two heaps.

703. Kth Largest Element in a Stream

Approach: Min heap(priority queue)

This is a continues stream of data so we have to find such a solution which can gives a answer in O(log n) at each time stamp. Whenever this type of answer is asked the very first solution is occurred in our mind is min heap. So for that we first push all the elements of given vector in the priority queue. Noe in add function first push that given element in the queue if the queue size is greater then the K then pop the elements from queue until it become <= K. Now return the top element of queue.

Algo:
First assign int K and min heap globally.

```
Int K;
priority_queue<int, vector<int>, greater<int>> min_heap;
// In the constructer push all the elements of vector in the queue. Also sign k to K.
KthLargest(int k, vector<int>& nums) {
        K = k;
        for(int i=0;i<nums.size();i++){
            min_heap.push(nums[i]);
        }
    }
// in the add function push the element in queue and then pop the element until queue size will not become <= K.
int add(int val) {
        min_heap.push(val);
        while(min_heap.size()>K){
            min_heap.pop();
        }
        return min_heap.top();
    }
```

Time complexity is : O(n log n) for constructor and O(log n) for add function.

Distinct Numbers in Window (Interview Bit).

Approach: Bruit force will able to solve this question in O(n ^ 2) time.
here the very first thing is use the Hashmat to find the distinct elements in each sliding window. And at each iteration add the frequency go next element and decrease the frequency of last element in the map if that element's frequency is 0 then remove it from map.
Algo:

```
unordered_map<int, int>mp;
Vector<int> ans;
for(int I=0; I<B;i++){
mp[A[I]]++;
}
Ans.push_back(mp.size());
for(int I=B;i<n;i++){
mp[A[i-B]] - - ;
if(mp[I-B]] == 0; mp.erase(mp[i-B]);
mp[A[I]]++;
Ans.push_back(mp.size());
}
```

Return ans;
Time complexity is O(n);

215. Kth largest element:

Approach: Minheap
this question is solved by very simple approach. In that you just sort the array and return nums[n-k] but it will take O(n log n) time. Another way is use heap data structure(queue). In that build a max heap in o(n) time and return the kth element. But,but,but the another best optimised solution is build a min heap until the size of queue is k and whoever queue size is > k pop the elements from queue and at the end return the top of queue.
Algo:

```
priority_queue<int, vector<int>, greater<int>> pq;
for(int I=0;i<nums.size();I++){
Pq.push(nums[I);
if(pq.size() > k) pq.pop();
}
Return pq.top();
```

Time complexity is O (n)

733. Flood Fill

Approach: Recursion
Start from source cell and move four direction recursively if cooler is = source cooler then change them.
Algo:

```
Void solve(vector<vector<int>>& image, in sr, int sc, int new_color, int n, int m, int source){
if(sr<0 II sr>=m II sc<0 II sc>=n){
Return;
}
Else if(image[sr][sc] != source){
Return;
}
image[sr][sc] = new_color;
solve(image, sr-1, sc, new_color, n, m, source);
solve(image, sr+1, sc, new_color, n, m, source);
solve(image, sr, sc-1, new_color, n, m, source);
solve(image, sr, sc+1, new_color, n, m, source);
}
//in main function
Int m = image.size(), n = image[0].size();
```

```
Int source = image[sr][sc];
if(source == color) return image;
solve(image, sr, sc, color, m, n, source);
Return image;
}
```

Time complexity is O(m * n);

133. Clone Graph

Approach : BFS and hash map

In entire YouTube, peoples are said that this is a toughest question ever in the graph. But instead of listening them let's made this question easy as possible. So even 2nd year student can understand easily. Please follow the steps given below:

Step1) make the new node with value as per the very first node.

Step2) make the map as a node, node pair and simple queue for BFS.

Step3) in map for very first. Use node as key which is given in the question and our copy node as value.

4) now until q become not empty follow the given steps:

```
while(!q.empty()){
Node* temp = q.front();
q.pop();
For(auto it: temp->neighbours){
if(! Mp.count(it)){
Node* clone = new Node(it->val);
mp[it] = clone;
Q.push(it);
}
Node* clone1  = mp[temp];
Node* clone2 = mp[it];
Clone1->neighbours.push_back(clone2);
}
}
Return copy;
```

Time complexity is O(E + V) where E = edges and V = Vertices.

BFS of Graph (GFG):

Approach: bfs

Algo:

```
Vector<int> visited(V, 0);
Vector<int> bfs;
Queue<int> q;
visited[0] = 1;
```

```
q.push(0);
while(!q.empty()){
Int node = q.front();
Q.pop();
bfs.push_back(node);
For (auto it: adj[node]){
if(!visited[it]){
visited[it] = 1;
Q.push(it);
} }
Return bfs;
```

Time complexity : we know that in undirected graph total degree of the graph is 2*edges and we iterate entire queue that's why it is a O(n + 2*E) And space complexity is O(n+n+n) because we used two vectors and one queue.

DFS of Graph (GFG):

Approach: DFS

Algo:

```
Void DFS(int node, vector<int> adj[], vector<int>&ans, vector<int>&visited){
visited[node] = 1;
ans.push_back(node);
for(auto it: adj[node]){
if(!visited[it]]{
visited[it] = 1;
DFS(it, adj, ans, visited);
}
}
// in the main function
Vector<int> visited(V, 0);
Vector<int> ans;
Int start = 0;
DFS(start, adj, ans, visited);
Return ans;
```

Time complexity is Same as the BFS which is O(n + 2*total degree) and space complexity is also O(n + n + n). N for recursive stack, another 2n for store the visited and ans vector.

Cycle Detection in the Undirected Graph(Coding Ninja):

Approach : Simple BFS Traversal

In the bfs you can start from the first node and travers entire graph. Now you can follow two path if available and you can reach some node by both the

path then definitely there is a cycle in the graph otherwise not. And remember one thing is, there is a possiblity in which there are some disconnected graph also in the question so you have check those all the graph for cycle. And to solve this question please maintain a queue which holds the where you go and from where the node is came.

Algo:

```
Bool BFS(vector<int>adj[], vector<int>&visited, int source){
Queue<pair<int, int>> q;
q.push({source, -1});
visited[source] = 1;
while(!q.empty()){
Int node = q.front().first;
Int parent = q.front().second;
Q.pop();
For(auto it: adj[node]){
if(!visited[it]){
visited[it] = 1;
Q.push({it, parent});
}
Else if(it != parent){
Return true;
} }
Return false;
// in the main method
Vector<int>adj[n+1];
Vector<int>visited(n,0);
for(int I=0;i<m;i++){
adj[edges[I][0]].push_back(edges[I][1]);
adj[edges[I][1]].push_back(edges[I][0]);
}
// for disconnected components
for(int I=1; I<=n; I++){
if(!visited[I] && BFS(adj, visited, I)) return “Yes”;
}
Return “No”;
```

Time complexity is : O(n + 2*E) where E is number of edges and it’s total is total degree of a given graph. Space is O(n + n)

Cycle Detection in the Undirected Graph(Coding Ninja):

Approach : DFS Traversal

In DFS start from first node also take one more parameter which is parent. Approach is that at each recursion first mark that node as visited and then

iterate its neighbours if the node is not visited and recursion will return true in back then also return true. And if it is visited and parent not equal to the node then also return true. All the thing is that you can goes deeper and deeper in nodes and at the back of recursion check if previous recursion is return true then this will also return true. Because the previous node said that I have a cycle then current node will also said that I have also cycle.
Algo:

```
Void DFS(vector<inr>adj[], vector<int>& visited, int source, int parent){
visited[source] = 1;
for(auto it: adj[source]){
if(!visited[it]){
if(DFS(adj, visited, it, source){
Return true;
}
Else if(visited[it] && parent != it){
Return true;
}
Return false;
}
//in main function vector<int> adj[n+1];
    for(int i=0;i<m;i++){
        adj[edges[i][0]].push_back(edges[i][1]);
        adj[edges[i][1]].push_back(edges[i][0]);
    }first make the adjanjancy matrix
// for disconnected components
for(int I=1; I<=n; I++){
if(!visited[I] && BFS(adj, visited, I)) return “Yes”;
}
Return “No”;
```

Time complexity is : O(n + 2*E) where E is number of edges and it’s total is total degree of a given graph. Space is O(n + n)

Detect Cycle In A Directed Graph (Coding Ninja)

Approach: DFS
If you are think this question like undirected graph then you are wrong. Because in directed graph there is a clearly defined edges. So the question is how to treat this question. Ans is take one more dfs_visited array along with the visited array. Idea is when you mark any node in visited array you can also mark it in the bfs_visited array. Only the difference is when you backtrack from recursion then unmark that node from only dfs_visited array. And any point the visited[node] == 1 and dfs_visited[node] == 1. Means there is a cycle in the graph other wise not.

Algo:
Bool DFS(unordered_map<int, vector<int>> &adj, vector<int>& visited, vector<int>& dfs_visited, int source){
visited[source] = 1;
dfs_visited[source] = 1;
for(auto it: adj[source]){
if(!visited[it]){
if(DFS(adj, visited, dfs_visited, it) return true;
}
Else if(dfs_visited[it]) return true;
}
dfs_visited[source] = 0;
Return false;
}
//in main function
unordered_map<int, vector<int>> adj;
for(auto it: edges){
adj[it.first].push_back(it.second);
}
Vector<int>visited(n+1, 0);
Vector<int>dfs_visited(n+1, 0);
for(int I=1; I<=n; I++){
If!visited[I] && DFS(adj, visited, dfs_visited, I)) return true;
}
Return false;
Time complexity: O(E + V), space is O(n + n) because we use two extra array

Detect Cycle In A Directed Graph (Coding Ninja)

Approach: BFS & Toposort
Approach is simple if we able to find toposort of size n then it has not contains the cycle other wise it has the cycle. Here we not use the vector to store the toposort because we don't need the actual order of toposort we just need a count of toposort order so we simply take one counter.
Algo:
unordered_map<int, vector<int>>adj;
Vector<int> indegree(n+1,0);
for(auto it: edges){
adj[it.first].push_back(it.second);
indegree[it.second]++;
}
Queue<int>q;
for(int I=1;i<=n;i++){

```
if(indegree[I] == 0) q.push(I);
}
Int count=0;
while(!q.empty()){
Int node = q.front();
count++;
for(auto it: adj[node]){
indegree[it] - -;
if(indegree[it] == 0) q.push(it);
} }
if(count == n) return 0;
Return 1;
```

Time complexityO(E+V) space is: O(n+n) for extra queue and indegree vector

Topological sort DFS (GFG):

Approach : DFS

What is Topological Sort?

Topological sort is a method of arranging the vertices of a directed graph in a linear order. It ensures that for every directed edge (u, v), vertex u comes before vertex v in the ordering. This sorting is only applicable to directed acyclic graphs (DAGs). The goal is to find a valid order by systematically visiting the vertices. The resulting order is not unique, but it provides insights into dependencies and is useful for tasks such as scheduling, dependency resolution, and compilation order determination.

Que-> How to solve this question using DFS?

To solve this question we take one stack and start from the any node in the graph. It is a graph question so obviously you have to maintain visited array also. Now go in depth of all the nodes recursively and when you make the back track ans reach last means dead node then push it into stack. Then take all the elements of stack it will your final topological order.

Algo:

```
Void DFS(vector<int>& adj[], stack<int>& st, vector<int>& visited, int source){
Visited[source] = 1;
for(auto it: adj[source){
if(!visited[it]){
DFS(adj, st, visited, it);
} }
St.push(source);
}
// in main function
Vector<int> visited(V,0);
```

```
Stack<int> st;
for(int I=0;i<V;i++){
if(!visited[I]){
DFS(adj, st, visited, I); }
}
Vector<int>& ans;
while(!st.empty()){
Ans.push_back(st.top());
St.pop();
}
Return ans;
```

Time complexity: O(E + V) space is O(n+n) for one stack and visited array.

Topological sort BFS (GFG) (kahn's algorithm):

Approach: BFS

If we try to solve this question similar to the DFS then again you are wrong. Because here the famous kahn's algorithm is come into the picture.

What is kahn's algorithm?

Kahn's algorithm, also known as Kahn's topological sorting algorithm, is used to find a topological ordering of vertices in a directed acyclic graph (DAG). It works by iteratively selecting vertices with no incoming edges, removing them from the graph, and adding them to the topological ordering. The algorithm uses a queue to keep track of vertices with no incoming edges and removes edges as it progresses. If the algorithm successfully completes, the resulting list will contain a valid topological ordering. However, if the graph contains a cycle, no valid topological ordering is possible.

In this we maintain one queue and one vector array called in degree. And first find all the in-degrees for all the nodes. Now que is how to find in degree of the node? Then ans is, we have given a adj list right? We iterate that list and if it is on right side then obviously it will have the incoming edge.

Algo:

```
Vector<int> indegree(V,0);
for(int I=0;i<V;i++){
for(aut it: adj[i]){
indegree[it]++; } }
//if the graph is acyclic then definitely it has the nodes which have 0 in degree
queue<int>q;
for(int I=0;i<V;i++){
if(inorder[I] == 0) q.push(i);
}
vector<int>topo_ordere;
while(!q.empty()){
```

```
Int node = q.front();
topo_ordere.push_back(node);
Q.pop();
for(auto it: adj[node]){
indegree[it] - -;
if(indegree[it] == 0) q.push(it);
} }
```

Time complexityO(E+V) space is: O(n+n) for extra queue and indegree vector

207. Course Schedule (kahn's algorithm)

Approach: Topological sort & BFS

If you see carefully then this question then it is a question related to find a cycle in directed graph. And we also know that one the advantage of topological sort is course scheduling.

Algo:

```
unordered_map<int, vector<int>> adj;
        vector<int>indegree(numCourses, 0);
        for(auto it: prerequisites){
            adj[it[0]].push_back(it[1]);
            indegree[it[1]]++;
        }
        queue<int>q;
        for(int i=0;i<numCourses;i++){
            if(indegree[i] == 0)q.push(i);
        }
        int count = 0;
        while(!q.empty()){
            int node = q.front();
            q.pop();
            count++;
            for(auto it: adj[node]){
                indegree[it]--;
                if(indegree[it]==0)q.push(it);
            }
        }
        if(count == numCourses) return true;
        return false;
```

Time complexityO(E+V) space is: O(n+n) for extra queue and indegree vector

200. Number of Islands

Approach : DFS in 4 direction with recursion

If we carefully observe this question then the main task is we have to find the number of connected components in the graph or grid. So at each recursion we goes 4 direction and make that 1 to 0. And importantly note that this question is very important. according to my suggestion, “it is a solution of another different 200 questions.” If you understand this recursion and DFS.

Algo:

```
void DFS(vector<vector<char>>& grid, int m, int n, int i, int j){
    if(i<0 || i>=m || j<0 || j>=n || grid[i][j] == '0'){
        return;
    }
        grid[i][j] = '0';
    DFS(grid, m, n, i-1, j);
    DFS(grid, m, n, i+1, j);
    DFS(grid, m, n, i, j-1);
    DFS(grid, m, n, i, j+1);
}
//in main function
 int m = grid.size();
        int n = grid[0].size();
        int componenets = 0;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(grid[i][j] == '1'){
                    componenets++;
                    DFS(grid, m, n, i, j);
                }
            }
        }
        return componenets;
```

Time complexity is: O(m * n) without use of any extra space

Above question can also solved by BFS traversal. But it will takes the O(m*n) extra space and O(n) for storing a queue.

Algo:

```
void BFS(vector<vector<char>>& grid, vector<vector<int>>& visited, int i, int j){
    int m = grid.size();
    int n = grid[0].size();
    visited[i][j] = 1;
    queue<pair<int,int>>q;
    q.push({i,j});
    while(!q.empty()){
```

```
        int row = q.front().first;
        int col = q.front().second;
        q.pop();
        vector<int> a = {1,-1,0,0};
        vector<int> b = {0,0,1,-1};
        for(int i=0; i<4;i++){
            int x = row + a[i];
            int y = col + b[i];
            if(x>=0 && x<m && y>=0 && y<n &&
               grid[x][y] == '1' && !visited[x][y]){
                   visited[x][y] = 1;
                   q.push({x,y});
               }
        }
    }
}
public:
    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<int>> visited(m, vector<int>(n,0));
        int componenets = 0;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(grid[i][j] == '1' && !visited[i][j]){
                    componenets++;
                    BFS(grid, visited, i, j);
                }
            }
        }
        return componenets;
```

Time complexity is O(n^2) ruffly, and space is O(m*n) to store visited matrix and O(n) extra to store the Queue in BFS.

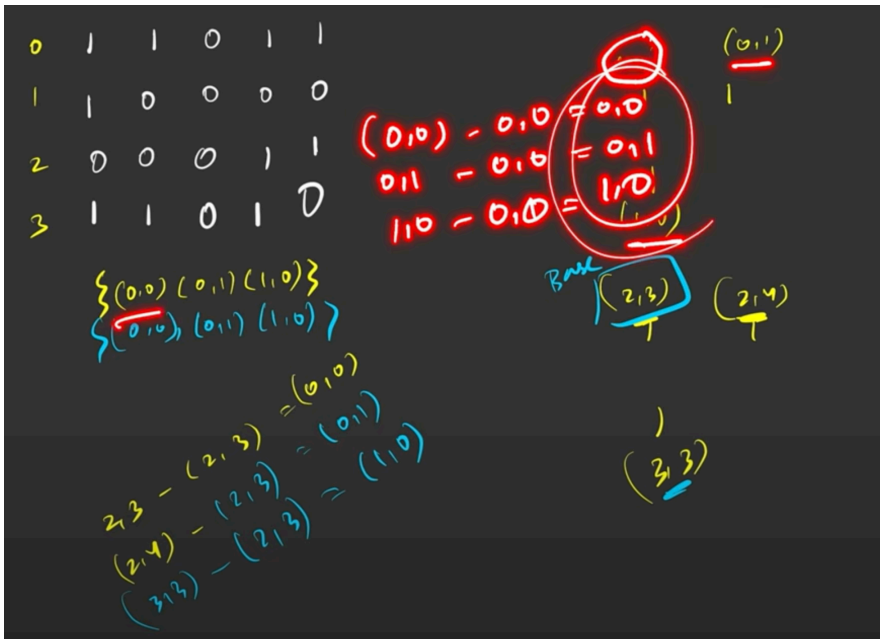
There is a another question which is higher version of this question **Find the number of islands(GFG)** which has the constraints of 8 direction instead of 4. Which can also solve by above 2 similar concepts.

Number of Distinct Islands (GFG)

Approach : DFS/BFS anything

It is a variation of Number of island of leetcode problems. But the tricky part is we don't count the mirror or same island. So for that one possible approach is

store that pattern of island into the set and at the last return the size of set.
Now the important question is how to store those island in the set?
For that answer is: take very first 1's coordinates as base and subtract from all the coordinates of that island so you can got the same island as shown in the given figure:



Algo:

```
    void DFS(vector<vector<int>>& grid, int i, int j, int n, int m,
    vector<pair<int,int>>& list,int &relative_i, int &relative_j) {
        if(i<0 || i>=n || j<0 || j>=m || grid[i][j] != 1){
            return;
        }
        grid[i][j] = 0;
        list.push_back({i-relative_i, j-relative_j});

        vector<int> a= {1,-1,0,0};
        vector<int> b = {0, 0, 1, -1};
        for(int temp = 0;temp<4;temp++){
            int x = i + a[temp];
            int y = j + b[temp];
            DFS(grid, x, y, n, m, list, relative_i, relative_j);
        }
        }

public:
    int countDistinctIslands(vector<vector<int>>& grid) {
        int n = grid.size();
```

```
int m = grid[0].size();
vector<vector<int>> visited(n, vector<int>(m, 0));
set<vector<pair<int,int>>> st;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (grid[i][j] == 1) {
            vector<pair<int,int>> list;
            DFS(grid, i, j, n, m, list, i, j);
            st.insert(list);
        }
    }
}
return st.size();
```

Time complexity is: O(m * n) without use of any extra space

785. Is Graph Bipartite?

Approach: BFS

The definition of bipartite graph is said that you can start from first node and if you can able to paint all the nodes with alternative colours then the given graph Is bipartite other wise not. And another approach Is that if your graph contains the cycle and the cycle length is even then definitely your graph is bipartite and if cycle length is odd then the graph is not bipartite.

For that instead of visited array we take the colour array with all the values of -1. And in the queue current node will tells to its neighbours if neighbour will not collard yet, then color it with opposite colour of current node and if it was collard and the colour is == to current node return false.

Algo:

```
Bool BFS(unordered_map<int, vector<int>>& adj, vector<int>& color, int
source){
color[source] = 0;
Queue<int>q;
q.push(source);
while(!q,empty()){
Int node = q.front();
q.pop();
for(auto it: adj[node]){
if(color[it] == -1){
color[it] = !color[node];
Q.push(it); }
Else if(color[it] == color[node]){
Return false; }
```

```
Return true;
}
Int n = graph.size();
unordered_map<int, vector<int>>adj;
for(int I=0;i<n;i++){
for(int j: graph[I]){
adj(i).push_back(j);
adj(j).push_back(I);
} }
Vector<int>color(n,-1);
for(int I=0;i<n;i++){
if(color[I] != -1 && (!BFS(Adj, colour, I)) return false;
}
Return true;
```

Time complexity is O(E + V). Space is O(n + n) for colour array and queue.
Using DFS: https://leetcode.com/problems/is-graph-bipartite/submissions/979216877/

Strongly Connected components Kosaraju's algorithm (GFG)

Approach: Kosaraju's algorithm with DFS
Kosaraju's algorithm will said that in a graph if there is a subgraph is present in which one node can reach all the nodes in that subgraph then that subgraph is called Strongly Connect component of a graph(SCC). The entire algorithm is broken into three parts:
1- sort the graph according to finish time and store it into stack.
2- Reverse all the edges of the graph
3- Again traverse the graph using DFS and store all the strong components in your ans vector.
Algo:
Link(GFG) Find the number of SCC: Link

Link (Coding Ninja) print all the SCC: Link

Time complexity is : O(V+E) + O(V+E) + O(V+E) ~ O(V+E) , where V = no. of vertices, E = no. of edges. The first step is a simple DFS, so the first term is O(V+E). The second step of reversing the graph and the third step, containing DFS again, will take O(V+E) each.
Space complexity: O(V)+O(V)+O(V+E), where V = no. of vertices, E = no. of edges. Two O(V) for the visited array and the stack we have used. O(V+E) space for the reversed adjacent list.

1192. Critical Connections in a Network (Tarjan's algorithm)

Approach: Trajan's algorithm

Critical components in a graph means if you can remove a edge from the graph and the graph will broken into 2 or more part then the given edge is a critical component of a graph.

So, how to approach this question?

Very first approach to solve this question is bruit force. You can start from first node and after removing each edge check whether the graph is remains connected or not. This will take lot of time because we remove and check the each edge.

What is the optimised approach? Ans id Tarjan's algorithm.

Logic is maintain 3 arrays called discovery time (in which time stamp you can reach the particular node), low time (either another path to reach that node or note?) and finally visited array. Start a DFS traversal and when you come after completing the recursion update the low (whether another path is present or not to reach that node?) And check if it is a connected component or not after removing that edge.

Algo:

```
Private:
Timer = 1;
Void DFS(int source, int parent, unordered_map<int, vector<int>>& adj, vector<int>& visited, vector<int>& time, vector<int>& lowest, vector<vector<int>>& bridges){
visited[source] = 1;
time[source] = timer;
lowest[source] = timer;
timer++;
for(auto it: adj[source]){
if(!visited[it]){
DFS(it, source, adj, visited, time, lowest, bridges);
lowest[source] = min(lowest[source], lowest[it]);
if(lowest[it] > time[source]){
Bridges.push_back({source, it});
} }
Else
Lowest[source] = min(lowest[source], lowest[it]);
} }
//in main function first create the adjanjancy list
unordered_map<int, vector<int>> adj;
for(auto it: connections){
adj[it[0]].push_back(it[1]);
adj[it[1]].push_back(it[0]); }
```

```
Vector<int> visited(n, 0), time(n,-1), lowest(n, -1);
Vector<vector<int>> bridges;
DFS(0, -1, adj, visited, time, lowest, bridges);
Return bridges;
}
```

Time complexity is O(E + V) space is O(3n) + O(N) to store the adj list.

Single source Shortest path Dijkstra algorithm (GFG):

Approach: Dijkstra algorithm using Priority Queue:
Initialise the Dist vector with size of number of nodes and values as a INT_MAX. Then traverse a graph using bfs traversal and fill all the values in Dist vector.
Algo:

```
Void BFS(int source, vector<vector<int>> adj[], vector<int>& dist){
dist[source] = 0;
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;
pq.push({0 , source});
While(!pq.empty()){
Int node = pq.front().second;
Int dis = pq.front().first;
for(auto it: adj[node]){
Int adjnode = it[0];
Int edgeweight = it[1];
if(dis + edgeweight < dist[adjnode]){
dist[adjnode] = dis + edgeweight;
Pq.push({dist[adjnode], adjnode});
} } }
// in main method
Vector<int>dist(V, INT_MAX);
BFS(S, adj, dist);
Return Dist;
```

Time complexity is O(E log V) and O(n) extra space for queue.

Single source Shortest path Dijkstra algorithm (GFG) (Set):

Approach: Dijkstra algorithm using Set Data Structure:
Initialise the Dist vector with size of number of nodes and values as a INT_MAX. Then traverse a graph using bfs traversal and fill all the values in Dist vector.
Algo:

```
void BFS(int source, vector<vector<int>> adj[], vector<int>& dist){
        dist[source] =0;
        set<pair<int,int>>st;
```

```
st.insert({0, source});
while(!st.empty()){
    auto container = *(st.begin()); //pointer og first element in the set
    int node = container.second;
    int dis = container.first;
    st.erase(container);
    for(auto it: adj[node]){
        int edgeweight = it[1];
        int adjnode = it[0];
        if(dis + edgeweight < dist[adjnode]){
            if(dist[adjnode] != INT_MAX){
            st.erase({dist[adjnode], adjnode});
            }
                dist[adjnode] = dis + edgeweight;
                st.insert({dist[adjnode], adjnode});
     } } } }
```

Time complexity is O(E log V) and O(n) extra space for set.

Single Source Shortest Path Bellman Ford Algorithm (GFG):

Approach: it is a solution for the drawbacks of Dijkstra algorithm in which it can able handle the negative weights as well negative cycle as well in the graph. But the constraint is you have to a directed graph if it is not a directed graph then please first covert it into directed graph. And we make a relaxation process for n-1 time.

Algo:

There is a adjanjancy list which is given like : {source, destination, weight}

```
Vector<int> dist(V, 1e8);
dist[S] = 0;
for(int I=0;i<V;i++){
for(auto it: edges){
Int source = it[0];
Int dest = it[1];
Int weight = it[2];
if(dist[source] != 1e8 && dist[source] + weight < dist[dest]){
dist[dest] = dist[source] + weight;
} }
//again make one relaxation to check the negative cycle
for(auto it: edges){
Int source = it[0];
Int dest = it[1];
Int weight = it[2];
if(dist[source] != 1e8 && dist[source] + weight < dist[dest]) {  return {-1}; } }
```

Time complexity is O(E * V)

Multisource Shortest Path Floyd warshall algorithm (GFG):

Approach: this question is different then all other shortest path algorithms because in those algorithms we will finds the shortest paths from any particular source node but in this graph we can find the matrix which holds the shortest distance from any node to any other nodes in the given graph.
Intuition is simple. If you want to find the distance from 1 to 3 via 2 then take the distance as: matrix[1] [3] = min(matrix [1][3], matrix[1][2] + matrix[2][3]);
Algo:
In the given question of GFG for not reachable path -1 will given. So we first convert it into the infinity and at the end of the day we can make that infinity as -1.

```
Int n = matrix.size();
for(int via=0; via<n; via++){
for(int I=0;i<n;i++){
for(int j=0;j<n;j++){
matrix[I][j] = min(matrix[I][j], matrix[I][via] + matrix[via][j]);
} } }
```

Time complexity is O(n^3) where n is number of nodes. It is a kind of exponential because It try all possible shortest path as brute force.
*note: if interviewer will said that find if negative cycle in the graph then try following:

```
for(int I=0;i<n;i++){
if(matrix[I] [I] < 0) “negative cycle is present” }
```

Now the question is can Dijkstra algorithm will able to find multi-source shortest path?
Then ans is yes. If graph does not contains the cycle. Solution is find all the shortest path from each nodes using time: V * (E log V).
According to striver bhaiya “do not tells the directly floyd warshall algo to the interviewer. Please tells all those things of also which is a plus point”.

Find a MST Prim’s Algorithm (GFG) :

Approach: take a visited array priority queue and a total_cost variable and start a BFS traversal from node 0. And remember that, till now we can mark a visited array in the for loop inside while loop of a priority queue but here the only changes is we mark the visited array in while loop of a priority queue and also adding the cost in that place only.
Algo:

```
Vector<int> visited(V, 0);
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;
```

```
Pq.push({0,0});
while(!pq.empty()){
Int node = pq.top().second;
Int weight = pq.top().second;
if(visited[node] == 1) continue;
visited[node] = 1;
total_cost += weight;
for(auto it: adj[node]){
Int adj_node = it[0];
Int adj_weight = it[1];
if(visited[adj_node] != 1){
pq.push({adj_weight, adj_node});
} }}
Return total_cost;
```

Time complexity is: O(E log E + E log E) and space is O(E) for priority queue.

Find a MST Krushkal Algorithm (GFG) :

Approach: Disjoint data structure:

If you know the disjoint data structure and it's implementation then the krushkal algorithm is much much easiest for you. Because there are only two steps in this: 1) sort the edges according to it's weights. 2) give all the nodes and it's adjacent nodes to the disjoint data structure if it is connected component then just ignore that weight and if not a connected component then add those nodes into our disjoint data structure and weight in our final cost for MST.

Algorithm for Disjoint Data Structure.

```
class disjointset{
    vector<int> rank, parent,size;
    public:
    disjointset(int n){
        rank.resize(n+1, 0);
        size.resize(n+1);
        parent.resize(n+1);
        for(int i=0;i<n;i++){
            size[i] = 1;
            parent[i] = i;
        }
    }
    int find_ultimate_parent(int node){
        if(node == parent[node]){
            return node;
        }
```

```
        return parent[node] = find_ultimate_parent(parent[node]);
    }
    void union_by_rank(int u, int v){
        int ultimate_parent_u = find_ultimate_parent(u);
        int ultimate_parent_v = find_ultimate_parent(v);
        if(ultimate_parent_u == ultimate_parent_v) return;
        if(rank[ultimate_parent_u] < rank[ultimate_parent_v]){
            parent[ultimate_parent_u] = ultimate_parent_v;
        }
        else if(rank[ultimate_parent_v] < rank[ultimate_parent_u]){
            parent[ultimate_parent_v] = ultimate_parent_u;
        }
        else{
            //if equal then you can connect either u to v or v to u anything
            parent[ultimate_parent_v] = ultimate_parent_u;
            rank[ultimate_parent_u]++;
        }

    }
    void union_by_size(int u, int v){
        int ultimate_parent_u = find_ultimate_parent(u);
        int ultimate_parent_v = find_ultimate_parent(v);
        if(ultimate_parent_u == ultimate_parent_v) return;
        if(size[ultimate_parent_u] < size[ultimate_parent_v]){
            parent[ultimate_parent_u] = ultimate_parent_v;
            size[ultimate_parent_v] += size[ultimate_parent_u];
        }
        else{
            //if equal then you can connect either u to v or v to u anything
             parent[ultimate_parent_u] = parent[ultimate_parent_v];
             size[ultimate_parent_u] += size[ultimate_parent_v];
        }
    }
};
```

Krushkal Algo:

```
int spanningTree(int V, vector<vector<int>> adj[])
    {
       vector<pair<int, vector<int>>> edges;
       for(int i=0;i<V;i++){
           for(auto it: adj[i ]){
               int node = i;
               int weight = it[1];
```

```
            int adjnode = it[0];
            edges.push_back({weight, {node, adjnode}});
        }
    }
    //creating the object of our desjointset datastructure with size V
    disjointset ds(V);
    sort(edges.begin(),edges.end());
    int total_cost = 0;
    for(auto it: edges){
        int wt = it.first;
        int u = it.second[0];
        int v = it.second[1];
        if(ds.find_ultimate_parent(u) != ds.find_ultimate_parent(v)){
            total_cost += wt;
            ds.union_by_rank(u,v);
        }
    }
    return total_cost;
```

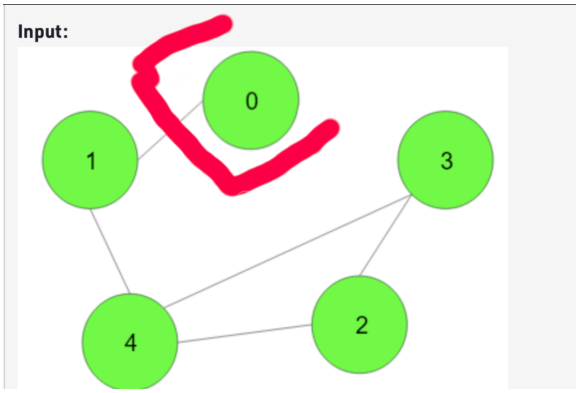
Time complexity is O(E log E) to sort the edges. O(V) for assigning verses to our disjoint data structure. O(4 alpha) which is almost constant for find_parent and Union operation.

Articulation Point - I(GFG)

Approach: Similar kind of Tarjan's Algorithm

Int the Number of critical component question of Leetcode we can check after removing edge, if graph is broken not 2 or more part or not? And in this question slight difference is we have to remove the nodes instead of edge. So the question is how to approach this question?

According to me my first observation is if the parent node or very first node of the graph contain only 1 edge and if we remove that node then the graph is not broken into two part it is still only single part as given below:



In the given image if we remove the 0 then still there is only one component in the graph. But in case if very first node is 2 or more then edges then definitely the very first node is a articulation point. So this is a critical point because if we are trying to add the node as articulation point then we have to check it is not a parent or first node of the graph. Now the only changes in the solution is when the node is not visited and we update the low[source].

Algo:

```
void DFS(int source, int parent, vector<int>adj[], vector<int>& visited,
  vector<int>& time, vector<int>& low, vector<int>& articulation_point){
      visited[source] = 1;
      time[source] = timer;
      low[source] = timer;
      timer++;
      int child = 0;
      for(auto it: adj[source]){
          if(it == parent) continue;
          if(!visited[it]){
              child ++;
              DFS(it, source, adj, visited, time, low, articulation_point);
              low[source] = min(low[source], low[it]);
              if(low[it] >= time[source] && parent != -1){
                  articulation_point[source] = 1;
              } }
// different thing then finding a bridge
else{
              low[source] = min(low[source], time[it]);
} }
      if(child > 1 && parent == -1){
         articulation_point[source] = 1;
      }
// in the main function:
vector<int> articulationPoints(int V, vector<int>adj[]) {
        vector<int> visited(V, 0);
        vector<int> time(V);
        vector<int> low(V);
        vector<int> articulation_point(V,0);
        for(int i=0;i<V;i++){
            if(!visited[i]){
                DFS(i, -1, adj, visited, time, low, articulation_point);
            }
        }
        vector<int> ans;
```

```
        for(int i=0;i<V;i++){
            if(articulation_point[i] == 1){
                ans.push_back(i);
            }
        }
        if(ans.size() == 0) return {-1};
        return ans;
    }
```

Time complexity is O(E + V) for DFS and space complexity is O(4n) for storing 4 different vectors.

Articulation Point - II(GFG)

Approach: Same as the previous.

This question is similar to previous question but only difference is there are more then 1 components are there in the given graph so we have to put only one for loop in previous approach to handle the multiple components in the graph.

Alien Dictionary (GFG)

Approach: topological sort with DFS

Here the problem is not how to you can implement and solve this question but the problem is how to identify this question as topological sort. So, remember one thing whenever you find the problems like ordering and finding “something before something” then topological sort will always helps you. To solve this question we first iterate all the strings and finding the not equal characters and according to that made a adjacency list in the form of directed graph. Because we know that topological sort will works for only directed graph.

Algo:

```
Void TOPOSORT(int source, vector<int> adj[], vector<int>& visited, stack<int>& st){
visited[source] = 1;
for(auto it: adj[source]){
if(!visited[it]){
TOPOSORT(it, adj, visited, st);
}
St.push(source);
}
//in main function
Vector<int> adj[K];
for(int I=0; I<N-1;I++){
string s1 = dict[I];
```

String s2 = dict[I+1];
Int len = min(s1.size(), s2.size());
for(int j=0;j<len;j++){
if(s1[j] != s2[j]){
adj[s1[j] - ‘a’].push_back(s2[j] - ‘a’);
} }
Vector<int> visited(K,0);
Stack<int> st;
for(int I=0;i<K;i++){
if(!visited[I]){
TOPOSORT(i, adj, visited, st);
} }
String ans = “”;
while(!st.empty()){
Char ch = (st.top() + ‘a’);
Ans += ch;
St.pop();
}
Return ans;
Time complexity is O(E + V) and space is O(K) for stack.
*note if interviewer will said that what about the dict like: [“abc”, “bad”, “ade”]. Here there is a cycle and we know that topological sort will not possible for cyclic graph and if we can say that topological sort is not possible for given dicts then obviously ordering of alien language is not possible.and another case is if larger string given before the smaller one then it is a not valid dictionary like : [“abcd”, “abc”].

542. 0/1 Matrix

Approach: BFS
If we seen this problem very first time then obviously anyone can say that this problem is solved by DFS. But but but, you are wrong. DFS will only works when you have an 0 with your surrounded cells otherwise it fails. That’s why we use the BFS over here. Only BFS will able to goes step wise in each cell.
Algo:
Write a is safe function to check row and columns occurs within that grid.
Step 1 - iterate over matrix and if the cell is already 0 then mark at visited and push in queue with steps = 0.
For taking a queue: queue<pair<pair<int,int>,int>> q;
Step 2- while(! Q.empty()){
Take the roe, col and steps from queue and make pop operation.
Put the steps into ans vector -> ans[row][col] = steps;
Now iterate that cell over it’s four direction using smart technique like:

Vector<int> a = {-1, 0 ,1, 0}, b= {0, 1, 0, -1};
Ans iterate from 0 to 4. Now new nrow = row + a[I] and ncol = col + b[I];
If new row and col is not visited then mark it visited then push into queue with steps+=1.
}
Return ans;
Time complexity is O(n * m) + O(n * m * 4) which is almost O(n * m) space is 2(n *m) to store visited matrix and final ans matrix.

130. Surrounded Regions

Approach: DFS/ BFS anything you want
The very first and important observation over here is the cell which contains 'O' at boundary points and connected with other cell in the grid which contains 'O' will never become a 'X'. So what is next? As simple as that remaining 'O' in the grid will converted into X.
Algo:

```
void DFS(int i, int j, int n, int m, vector<vector<char>> board,
            vector<vector<int>>& visited){
   if(i<0 || i>=n || j<0 || j>=m || board[i][j] == 'X' || visited[i][j] == 1) {return;}
   visited[i][j] = 1;
   DFS(i-1, j, n, m, board, visited);
   DFS(i+1, j, n, m, board, visited);
   DFS(i, j-1, n, m, board, visited);
   DFS(i, j+1, n, m, board, visited);
   }
//in the main function
```

First check all the boundry points and if there is a 'O' then call the DFS for that cell.
And finally check for the remaining 'O' in the grid using:

```
for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
           if(!visited[i][j] && board[i][j] == 'O'){
                 board[i][j] = 'X';
           } } }
```

Time complexity is O(m * n) + O(m * n) for DFS in worst case in all the cell of Grid will contains the 'O'. Space is O(m *n) for visited array.

1020. Number of Enclaves

Approach : DFS/ BFS anything you want
There is a simple observation in the question. Find the 1 with can touched on boundary points 1. First make them 0 and then count the number of remaining 1's in the grid.

```
void DFS(int i, int j, int n, int m, vector<vector<int>>& grid){
    if(i<0 || i>=n || j<0 || j>=m || grid[i][j] == 0){
        return;
    }
    grid[i][j] = 0;
    DFS(i-1,j,n,m,grid);
    DFS(i+1,j,n,m,grid);
    DFS(i,j-1,n,m,grid);
    DFS(i,j+1,n,m,grid);
}
```

//in the main function

First check all the boundry points and if there is a 1 then call the DFS for that cell.

Now at the end count the remains g 1's in the grid and return the count.

```
int count = 0;
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            if(grid[i][j] == 1) count++;
        }
    }
    return count;
```

Time complexity is O(n * m) with no extra space.

802. Find Eventual Safe States

Approach : DFS

The simple observation is take each node and It will not goes to cycle or make the cycle then simply it is a safe node. So how to do it? It is a as simple as that. Take another safe_node array along with visited and dfs_visited for detect a cycle using dfs and if node is not a part of cycle then simple push it into the safe_node array and finally sort it and return it.

Algo:

```
Bool DFS(int source, vector<vector<int>>& graph, vector<int>& visited, vector<int>& dfs_visited, vector<int>& safe_node){
visited[source] = 1;
dfs_visited[source] = 1;
for(auto it: graph[source]){
if(!visited[source]){
if(DFS(it, graph, visited, dfs_visited, safe_node)) return true;
}
Else if(dfs_visited[it] == 1) return true;
}
dfs_visited[source] = 0;
```

```
safe_node.push_back(source);
Return false;
}
vector<int>visited(n,0);
        vector<int>dfs_visited(n,0);
        vector<int> safe_node;
        for(int i=0;i<graph.size();i++){
            if(!visited[i]){
                DFS(i, graph, visited, dfs_visited, safe_node);
 } }
        sort(safe_node.begin(),safe_node.end());
        return safe_node;
```

Time complexity is O(E + V) and space is O(2n) for two visited array and 1 extra (n) to store the safe nodes in worst case where all the nodes are safe.

How to solve this Question using Topological sort?

We know that topological sort will works for only directed acyclic graph. When the cycle is coming the topological sort will stops it's execution. So one important thing over here is, if we reverse our graph then apply a topological sort then definitely it will gives all the safe nodes.

Time complexity using topological sort is same as our DFS solution but it will reduce some space complexity because don't used dfs_visited array over here.

Shortest path in Directed Acyclic Graph (GFG):

Approach: Topological Sort.

There is a question, if we find the shortest paths using standard algorithms like Dijkstra then why we use topological sort?. So, ans is topological sort will keep going in sequential manner like first A -> B -> C in topological order. That's why it some how reduce the steps For finding the shortest path.

Algo:

First find the topological order of the graph.

Second is go through each node from topological order and perform the relaxation process for that node in your adjacency list.

Relaxation process for topological ordered nodes:

```
while(!st.empty()){
            int node = st.top(); st.pop();
            for(auto it: adj[node]){
                int dest = it.first;
                int weight = it.second;
                dist[dest] = min(dist[dest], dist[node] + weight); }}
```

Time and space complexity is our standard complexities for DFS and loop.

Shortest path in Undirected Graph having unit distance (GFG):

Approach: this question can be solved by our simple plain BFS traversal. How? Let's see in algorithm.

-> First create a adjacency list for undirected graph.

-> take one queue and push source node on it. Also take 1 Dist array with size N and value as infinity. And in dist[src] make 0.

-> while(!q.empty()){

```
int source = q.front();
  q.pop();
  for(auto it: adj[source]){
     if(dist[source] + 1 < dist[it]){
        dist[it] = dist[source] + 1;
        q.push(it); } } }
```

-> in final dist array if there is a value infinity then make it -1 and return Dist.

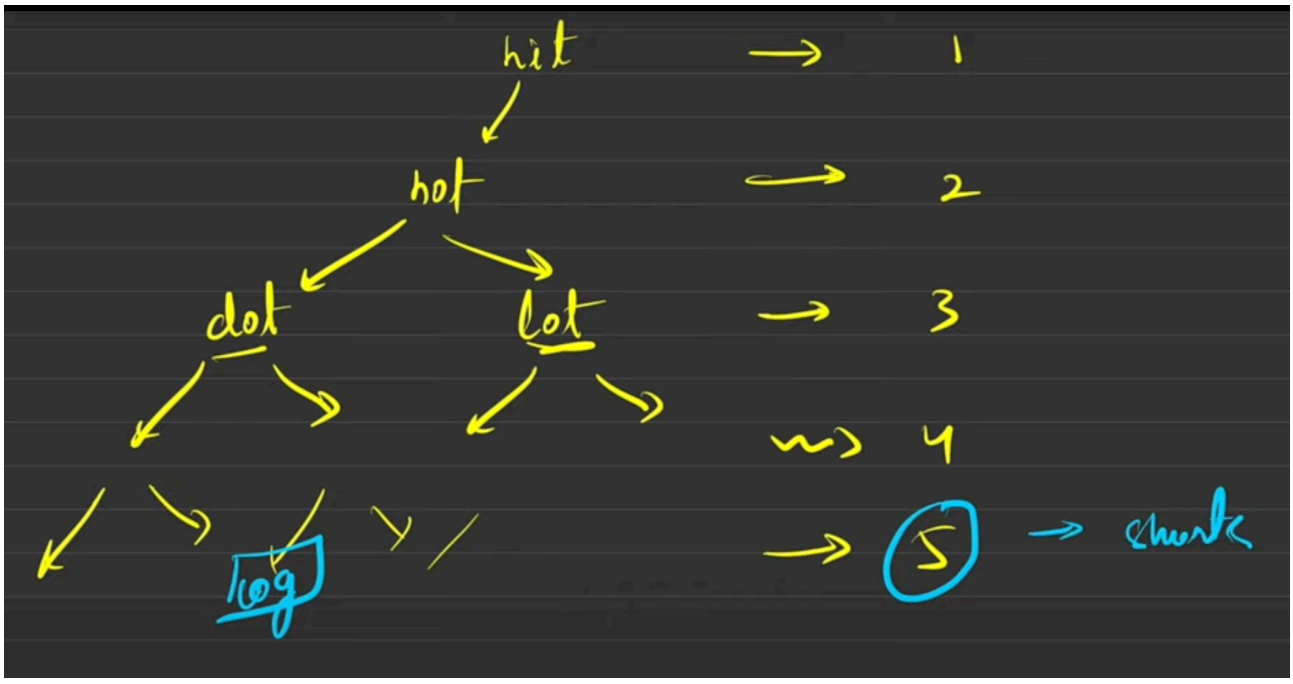
Time complexity is O(E + V) which is for standard BFS traversal. And space is O(n) for queue.

127. Word Ladder I

Approach: BFS

Here there is nothing like logic. Only thing is observation and using bruit force technique definitely you can able to solve this question.

Let's go over the example : if the given word is "hit" then you can try out all the character replacement from a to z like: "ait", "bit" … "zit" and similar to all the index I. And if you found that element in your dictionary then add it into our queue. And your traversal will looks like:



Since it is like a level wise so that's why we use a levelorder(BFS) traversal. And if you found the target word then return that length otherwise return 0.
Algo:
-> first create a set using your dictionary because it will find a word in O(1).

```
Queue<pair<string,int>>q;
q.push({startword,1});
St.erase(startword);
while(!q.empty()){
String word = q.front().first;
Int len = q.front().second;
q.pop();
for(int I=0;i<word.size();I++){
for(char c = 'a'; c<='z';c++){
String s1 = word;
s1[I] = c;
if(st.count(s1)){
st.erase(s1);
q.push({s1,len+1});
if(s1 == endowed) return len+1;
} } }
Return 0;
```

Time complexity is: O(word.length * 26 * log N) because for all the words length is same and 26 to iterate from a to z and log N which can take your set to find and remove the word.

Word Ladder II (GFG):

Approach : BFS.
In this question we also follows the same approach as per the previous approach. But if you carefully observe then there is a slight difference, which you do not erase element after updating our word. Why? Because it might happen that It will a sequence of another ans for ex-> for "hit" there are two ans is possible like: ["hit", "hot", "lot"], ["hit", "hot", "dot"] so after adding "hot" do not remove "hot" because it will forms a another ans. So for that we maintain another vector of string to remove the words and we also maintain queue which contains the vector of string.
Algo:

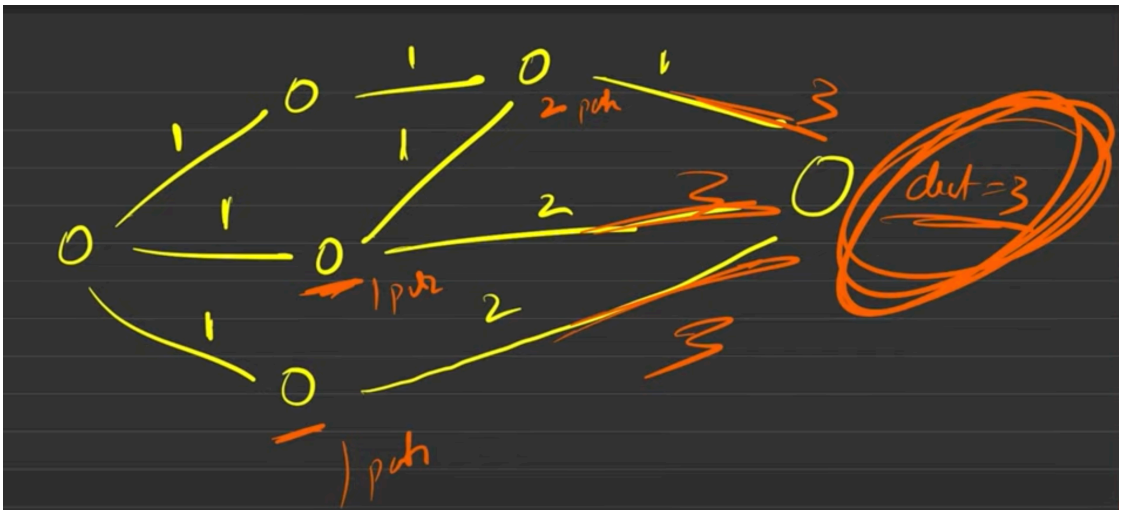
```
Set<string>st(wordlist.begin(), wordlist.end());
Vector<vector<string>>ans;
Vector<string>wordonlevel;
Queue<vector<string>>q;
Q.push({startword});
Wordonlevel.push_back(start word);
```

```
Int level = 0;
while(!q.empty()){
Auto vec = q.front();
if(vec.size() > level){
for(auto it: wordonlevel) st.erase(it);
}
String s1 = vec[vec.size()-1];
if(s1 == endowed){
Ans.push_back(vec);
}
for(int I=0;i<s1.size();I++){
Char original = s1[I];
for(char c = ‘a’;c<=‘z’;c++){
s1[I] = c;
if(st.count(s1)){
Vec.push_back(s1);
q.push(vec);
//mark as a visited word
wordonlevel.push_back(s1);
Vec.pop_back();
} }
s1[I] = original;
}}
Return ans;
```

There is not a well defined time complexity for this algorithm. Because it might be dependant on the size of the words and size of a wordlist size. This approach will not accepted at leetcode platform. That s we have to optimise it for leetcode. But this solution is more then enough for interview.

1976. Number of Ways to arrive at Destination

Approach : Dijkstra algorithm



very big mistake is we count only short distance of all arrival for that particular node. But this is completely wrong because there is number of paths are possible from previous node like this image.
In these image for reaching the last node min distance is 1 but for reach the previous node which have min distance there are two paths are possible. So instead of count min distance path we count the ways to reach the min distance path.
Algo:

```
while(!pq.empty()){
        ll node = pq.top().second;
        ll wt = pq.top().first;
        pq.pop();
        for(auto it: adj[node]){
            ll dest = it.first;
            ll weight = it.second;
            if(wt + weight < dist[dest]){
                dist[dest] = wt + weight;
                pq.push({dist[dest], dest});
                no_ways[dest] = no_ways[node]%mod;
            }
            else if(wt + weight == dist[dest]){
                no_ways[dest] = (no_ways[dest] + no_ways[node]) % mod;
          } } }
    return (int)no_ways[n-1];
```

Time complexity is O(E log V) which is standard for Dijkstra.

787. Cheapest Flights Within K Stops

Approach: Dijkstra algorithm:
This is another interesting question of Dijkstra algorithm. The only difference is you can take stops count in your queue and along with minimum distance also check can it will satisfy the condition of minimum stops or not?

Frog Jump (Coding Ninja):

Approach: recursion, dynamic programming, two pointer
The important question is why greedy method doesn't work here? Ans is, in greedy at each step you have to option either take that step into our ans or do not take. But important catches over here is you have to find out the minimum energy used and we all know that whenever such question occurs like find the minimum Distance or something then greedy approach will fails. So, why we not use recursion here?
Algo:

```
Int solve(int idx, vector<int>& heights){
if(idx == 0) return 0;
Int firststep = solve(idx-1, heights) + abs(heights[idx] - heights[idx-1]);
Int second step = INT_MAX;
```

if(idx > 1) secondstep = solve(idx-2, heights) + abs(heights[Idx] - heights[idx-2]);
Return min(firststep, secondstep);
}
This recursion will take exponential time.

Now the question is how to optimise it? Obviously my dear friend if you can see the recursion and overlapping sub-problems then one important thing called dynamic programming (memoization) Is come into the picture.
Algo:
Int solve(int idx, vector<int>&heights, vector<int>&dp){
if(idx == 0) return 0;
if(dp[idx] b != -1) return dp[idx];
Int firststep = solve(idx-1, heights, dp) + abs (heights[Idx] - heights[idx-1]);
Int secondstep = INT_MAX;
if(I > 1) secondstep = solve(idx-2, heights, dp) + abs(heights[Idx] - heights[idx-2]);
Return dp[idx] = min(first step, second step);
}
This function will runs in O(n) time and O(n) extra space.

*Note, remember one important thing is whenever you see the things like idx-1 and idx-2 the space optimisation will always possible.
Space optimised version;
Int prev = 0;
Int prev2 = 0;
for(int I=1;i<n;i++){
Int firststep = prev +abs(heights[I] - heights[i-1]);
Int secondstep = INT_MAX;
if(I>1) secondstep = prev2 + abs(heights[I] - heights[i-2]);
Int curr = min(firststep, secondstep);
Prev2 = prev;
Prev = curr; }
Return prev;
Time complexity is O(n) with no any extra space.

198. House Robber I (Maximum adjacent sum Coding ninja):

Approach: Recursion, memoization, Tabulation, Two pointer space optimised
This two questions are same. So first we tried the recursion with the logic of pick and not pick, then tried with dp array to store the overlapping subproblems and then instead of recursion's additional stack space we tried simple tabulation. And finally space optimised two pointer approach will be cunstructed because we know for calculating max sum at each stage we only need dp[i-1] and dp[i-2].
Algo:

```
Int prev = nums[0];
Int prev2 = 0;
for(int I=0;i<n;i++){
Int pick = nums[I];
if(I>1) pick+=prev2;
Int nitpick = prev;
Int curri = max(pick, nitpick);
Prev2 = prev;
Prev = curri;  }
Return prev;
```

Time complexity is O(n) with no any extra space.

213. House Robber II

The only change in this question is you can not add both first and last element because they are adjacent. So the simple solution of this question is tak 1 to n-1 elements and find maximum sum and again take 2 to n elements and find maximum ans so it is a solution of your House robber question. And finally return maximum among them.

Time complexity is O(n + n) because we iterate array two times.

Ninja's Training (Coding Ninja)

Approach: Recursion, memoization, space optimised version with 4 size array.

You can do a greedy approach but it will not work here. Because let's you have example like: [10 60 50] and [30 100 20] in this example if you can start with greedy approach then it will gives you the wrong answer which is 60+30 = 90 but if we carefully observe then the correct answer is 150. So that's why do not take greedy approach to solve such a questions.

There is a thumb rule 👍 that when greedy was fails and all the permutations are required, do not think complex just apply recursion.

According to suggestion please start from bottom up approach in recursion because it will gives you the base case easily. So in very less time coding round you not have to think more for base case of recursion.

Algo:

```
Int solve(int day, int last, vector<vector<int>>&points){
if(day == 0){
Int maxi = 0;
for(int tast=0;task<3;task++){
if(task != last ) Maxi = max(maxi, points[0][I]); }
Return maxi; }
Int maxi = o;
for(int task = 0;task<3;task++){
if(task != last){
Int point = points[day][task] + solve(day-1, task, points);
```

```
Maxi = max(maxi, point);
}
Return maxi; }
```

This code will perfectly fine. But, if you observe one thing it will take exponential time and it has some overlapping sub problems. So let's memoize it.

Algo:

```
Int solve(int day, int last, vector<vector<int>>&points){
if(day == 0){
Int maxi = 0;
for(int tast=0;task<3;task++){
if(task != last ) Maxi = max(maxi, points[0][l]); }
Return maxi; }
if(dp[day][last] != -1_ return dp[day][last];
Int maxi = 0;
for(int task = 0;task<3;task++){
if(task != last){
Int point = points[day][task] + solve(day-1, task, points);
Maxi = max(maxi, point);
}
Return dp[day][last] = maxi;
```

Now, this code will accepted. But what is the next step after memoization? Obviously tabulation. Let's apply tabulation.

Algo:

```
Vector<vector<int>>dp(n, vector<int>(4,-1));
//according to base case if the recursion we fill the first vector of dp table.
dp[0][0] = max(points[0][1], points[0][2]);
dp[0][1] = max(points[0][0], points[0][2]);
dp[0][2] = max(points[0][0], points[0][1]);
dp[0][3] = max(points[0][1], points[0][2]);
// second step is iterate the points table and fill the values in dp table.
for(int day= 1;day<n;day++){
for(int last=0;last<4;last++){
Int maxi = 0;
for(int task=0;task<3;task++){
if(task != last) maxi = max(maxi, points[day][task] + dp[day-1][task];
}
dp[day][last] = maxi;
```

And finally return the dp[n-1][3].

This is a perfect solution which can run in O(n * 4 * 3). Time and O(n *4) Space. But can space optimisation will allow here?

Previously we observed that there is another thumb 👍 rule in the Dp which is whenever you see like idx-1 or idx-2 then 100% space optimisation will also possible. And in our case which is dp[day-1] [task].

Instead of n * 4 array what if we take only 4 size 1D array because we need only previous day's data. So that is a best solution for this problem which runs in O(n * 4 *3) time and O(4) means almost constant space.

Algo:

```
vector<int>prev(4,0);
    prev[0] = max(points[0][1], points[0][2]);
    prev[1] = max(points[0][0], points[0][2]);
    prev[2] = max(points[0][0], points[0][1]);
    prev[3] = max(points[0][1], points[0][2]);
for(int day=1;day<n;day++){
    vector<int> dummy(4,0);
    for(int last=0;last<4;last++){
        for(int task = 0;task<3;task++){
            if(task != last){
                int point = points[day][task] + prev[task];
                 dummy[last] = max(dummy[last], point);
     } } }
    prev = dummy; }
return prev[3];
```

62. Unique Paths

Approach: Recursion, memoization, tabulation

Whenever you see such question like finding number of paths then you have to find the ways in given directions using recursion. Only recursion can give us the TLE so that's why we will used memoization and tabulation method.

Memoization with bottom up approach:

```
int solve(int m, int n, vector<vector<int>>&dp){
    if(m<0 || n<0){
    return 0;
}
if(m == 0 && n == 0){
    return 1;
}
if(dp[m][n] != -1) return dp[m][n];
int left = solve(m, n-1, dp);
int up = solve(m-1, n, dp);
return dp[m][n] = left + up;
}
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>>dp(m,vector<int>(n,-1));
        return solve(m-1,n-1,dp); } };
```

Time complexity is O(m*n) space is O 2(m*n) because we use recursive stack space also.

Remember one thing whenever you write recursive solution then try to build a bottom up recursion. Because it can helps to generate the tabulation approach.

Tabulation approach:

```
int uniquePaths(int m, int n) {
    vector<int> prev(n,0);
    for (int i = 0; i < m; i++) {
        vector<int>row(n,0);
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0) row[j]=1;
            else{
                int right = 0, down = 0;
                if (i > 0) right = prev[j];
                if (j >= 1) down = row[j-1];
                row[j] = right+down; } }
        prev = row; }
    return prev[n-1]; } }
```

Time complexity is O(m * n) space is also O(m * n) to store the Dp array.

63. Unique Paths II

This question is same as previous the only difference is in recursion if you can find the obstaclegrid[I][j] == 1 then simply return 0. Remaining logic will same.

Changes:

```
if(m<0 || n<0 || grid[m][n]==1){
        return 0;
    }
```

And in tabulation method after filling the dp cell if obstacle grid[I][j] == 1 then simply make that dp cell as 0.

```
 if(obstacleGrid[i][j] == 1) dp[i][j] = 0;
```

164. Minimum Path Sum

Approach: Dynamic Programming

The first approach come in mind after seen this question is greedy. But the important thing is in greedy today's decision might be affect a future why let's see in given figure:

In given example if you follow the greedy then greedy approach will say that please move down first. But we clearly says that if we move down then answer is definitely wrong.
That's why my dear friend what are you waiting go at leetcode and try recursion. In previous questions when we go beyond the gris we return 0 because in that's question we have to count the number of paths but here we have to find out minimum path that's why if we go beyond the grid then we return some big number which can never include in our answer. We all know that only recursion will never work if we have overlapping sub problems that's why we have to memoize it.
Memoization Algo:

```
int findmin(int m, int n, vector<vector<int>>& grid, vector<vector<int>>& dp){
    if(m<0 || n<0){
        return 1e5;
    }
    if(m==0 && n==0){
        return grid[m][n];
    }
    if(dp[m][n] != -1) return dp[m][n];
    int up = grid[m][n] + findmin(m-1, n, grid, dp);
    int left = grid[m][n] + findmin(m, n-1, grid, dp);
    return dp[m][n] = min(up, left); }
```

Time complexity is O(m * n) ans space is O 2(M*n) twice because we also use the recursion stack space.
And it is a thumb rule that if we able to memoize some than definitely we have also able to reduce the recursion stack space also.

```
dp[0][0] = grid[0][0];
        for(int i=0; i<m;i++){
            for(int j=0;j<n;j++){
                if(i==0 && j==0) continue;
                int left=1e5,up =1e5;
                if(i>0) up = grid[i][j] + dp[i-1][j];
                if(j>0) left = grid[i][j] + dp[i][j-1];
                dp[i][j] = min(up, left);
            }
        }
        return dp[m-1][n-1];
```

Time and space couple city will O(m * n)

120. Triangle

This is quite similar problem then the previous because here our recursion will have ends at any col of last row that's why we do not allow to use bottom up recursion here so we have to build a top down recursion which can start from very first cell and ends at any column in last row.
Recursion Algo:

```
int minpath(int i, int j, int m, vector<vector<int>>& triangle,
vector<vector<int>>& dp){
    if(i == m-1){
        return triangle[i][j];
    }
    if(dp[i][j] != -1) return dp[i][j];
    int down = triangle[i][j] + minpath(i+1, j, m, triangle, dp);
    int right_dia = triangle[i][j] + minpath(i+1, j+1, m, triangle, dp);
    return dp[i][j] = min(down, right_dia); }
```

It will gives TLE that's why we use memoization and it run on O(n*m) time and O(m*n) dp space and O(n) recursive stack space.

Important as we know here I+1, j+1 are there that's why space optimisation will also possible.

Space optimised Algo:

```
vector<int>prev(n,0);
        //first fill the last row of dp table
        for(int i=0;i<n;i++){
            prev[i] = triangle[m-1][i];
        }
        for(int i=n-2; i>=0; i--){
            vector<int>temp(n,0);
            for(int j=i; j>=0; j--){
                int upper = triangle[i][j] + prev[j];
                int diagonal = triangle[i][j] + prev[j+1];
                temp[j] = min(upper, diagonal);
            }
            prev = temp;
        }
        return prev[0];
```

Time complexity is O(n * m) space complexity is O(n).

Maximum Path Sum in the matrix (Coding Ninja)

Approach: Dynamic programming

Till now we solved examples like: recursion start with (0,0) cell and ends at (n-1, m-1), then we solved start from (0,0) and ends at any column in last row. But now we solve in which recursion can start from any col of first roe and ends at any col in last row.

Memoization Algo:

```
int f(int i, int j,vector<vector<int>> &matrix, vector<vector<int>> &dp){
    if(j<0 || j>= matrix[0].size()){
        return -1e8;
    }
    if(i == 0){
        return matrix[i][j];
    if(dp[i][j] != -1) return dp[i][j];
```

```
    int upper = matrix[i][j] + f(i-1, j, matrix, dp);
    int leftdia = matrix[i][j] + f(i-1, j-1, matrix, dp);
    int rightdia = matrix[i][j] + f(i-1, j+1, matrix, dp);
   return dp[i][j] = max(upper, max(leftdia, rightdia)); }
```

Time complexity is: O(n * m) and space is O(n *m) for store the memoization + O(n) for recursion stack space.

Tabulation algo:

```
int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>>dp(n, vector<int>(m,0));
    for(int j=0; j<m;j++){
        dp[0][j] = matrix[0][j]; }
    for(int i=1;i<n;i++){
        for(int j=0;j<m;j++){
            int upper = matrix[i][j] + dp[i-1][j];
            int leftdia = -1e8, rightdia = -1e8;
            if(j-1 >= 0) leftdia = matrix[i][j] + dp[i-1][j-1];
            if(j+1 < m) rightdia = matrix[i][j] + dp[i-1][j+1];
            dp[i][j] = max(upper, max(leftdia, rightdia));
        }}
    int ans = -1e8;
    for(int i=0;i<m;i++){
        ans = max(ans, dp[n-1][i]); }
    return ans;
```

Time complexity is same but here we do not use the recursion so we can say that we optimised a O(n) recursion stack space.

Further space optimisation will also possible using two dummy array.

Because at each point we only need previous row of the dp table.

Chocolate Pickup (Coding Ninja)

Approach: 3D DP

It Is quite Tough problem because it has the logic of two recursion in itself. Alice and bob are located at top two corners and they have to find maximum chocolate in the path. Here if you can find out the exploration points then bob and Alice will goes row wise simlteniously but in column wise if bob goes to one side then Alice have three option to go and if Alice go to one step then bob have three option to go that's why we use two for loop to call the recursion 9 times at each stage. One another important aspects is when bob and Alice will collide then do not return that grid cell 2 times just return 1 times. It will have some overlapping subproblems that's why we have to memoize it. The biggest question is how to memoize it because it has the three parameters which is I, j1, j2 so whenever you have to store three parameters use the 3dimension array which is called 3D Dp.

Recursive memoization Algo:

```
int solve(int i, int j1, int j2, int r, int c, vector<vector<int>> &grid,
vector<vector<vector<int>>>& dp) {
   if (j1 < 0 || j1 >= c || j2 < 0 || j2 >= c) {
       return -1e8; }
    if (i == r - 1) {
       if (j1 == j2) return grid[i][j1];
       else return grid[i][j1] + grid[I][j2]; }
    if(dp[i][j1][j2] != -1) return dp[i][j1][j2];
    int maxi = -1e8;
    for (int dj1 = -1; dj1 <= 1; dj1++) {
       for (int dj2 = -1; dj2 <= 1; dj2++) {
           int val = 0;
           if(j1 == j2) val = grid[i][j1];
           else val = grid[i][j1] + grid[i][j2];
           val += solve(i+1, j1+dj1, j2+dj2, r, c, grid, dp);
           maxi = max(maxi, val); } }
    return dp[i][j1][j2] = maxi;
```

Time complexity is O(N * M * 9) and space complexity is O(n * m * m) + O(n) recursive stack space.

To convert memoization to tabulation entire process will be same only difference id first fill the last row of dp table using last row of our grid.

```
 for(int j1 = 0; j1<c; j1++){
       for(int j2=0; j2<c; j2++){
           if(j1 == j2){
               dp[r-1][j1][j2] = grid[r-1][j1];
           }
           else{
               dp[r-1][j1][j2] = grid[r-1][j1] + grid[r-1][j2]; } } }
   for(int i=r-2;i>=0;i--){
       for(int j1 = 0;j1<c;j1++){
           for(int j2=0; j2<c; j2++){
               int maxi = -1e8;
               for (int dj1 = -1; dj1 <= 1; dj1++) {
                   for (int dj2 = -1; dj2 <= 1; dj2++) {
                       int val = 0;
                       if(j1 == j2) val = grid[i][j1];
                       else val = grid[i][j1] + grid[i][j2];
                       if(j1+dj1 >=0 && j1+dj1<c && j2+dj2>=0 && j2+dj2<c)
                               val += dp[i+1][j1+dj1][j2+dj2];
                       else val += -1e8;
                       maxi = max(maxi, val);
                       }
                   }
               dp[i][j1][j2] = maxi; } } }
```

And finally our answer stored at dp[0][0][c-1] so return it.

Here time ans space complexity is same but we reduce the extra recursive stack space.

Subset Sum Equal To K (Coding Ninja)

Approach : recursion and DP
The logic of pick and nitpick can be applied here because we have to find a subset sum. And here according to me please please please use the bottom approach because it will helps you to connect any recursion into DP if it has the overlapping subproblems.
Memoization Algo:

```
bool solve(int idx, vector<int> &arr, int k, vector<vector<int>>&dp) {
    if (k == 0) return true;
    if (idx == 0){
        if(arr[0] == k) return true;
        return false;
    }
  if(dp[idx][k] != -1) dp[idx][k];
    bool nottake =  solve(idx-1, arr, k, dp);
    bool take = false;
    if(arr[idx] <= k) take = solve(idx-1, arr, k-arr[idx], dp);
    return dp[idx][k] = (take || nottake); }
```

Time complexity is O(n * k) and space is O(n *k) for dp array and O(n) for recursive stack.
Tabulation Dp:

```
vector<vector<bool>>dp(n, vector<bool>(k+1,false));
    for(int i=0;i<n;i++){
        dp[i][0] = true;
    }
    dp[0][arr[0]] = true;
    for(int i=1;i<n;i++){
        for(int target = 1; target<=k; target++){
            bool nottake =  dp[i-1][target];
              bool take = false;
              if(arr[i] <= target) take = dp[i-1][target-arr[i]];
              dp[i][target] = (take || nottake); } }
return dp[n-1][k];
```

Here we don't able to optimise the Time complexity but we able too optimise the recursive stack space.
*And again to convert memoization to Tabulation don't make difficult your task just use the recurrence relationship which you can used in your memoization approach.

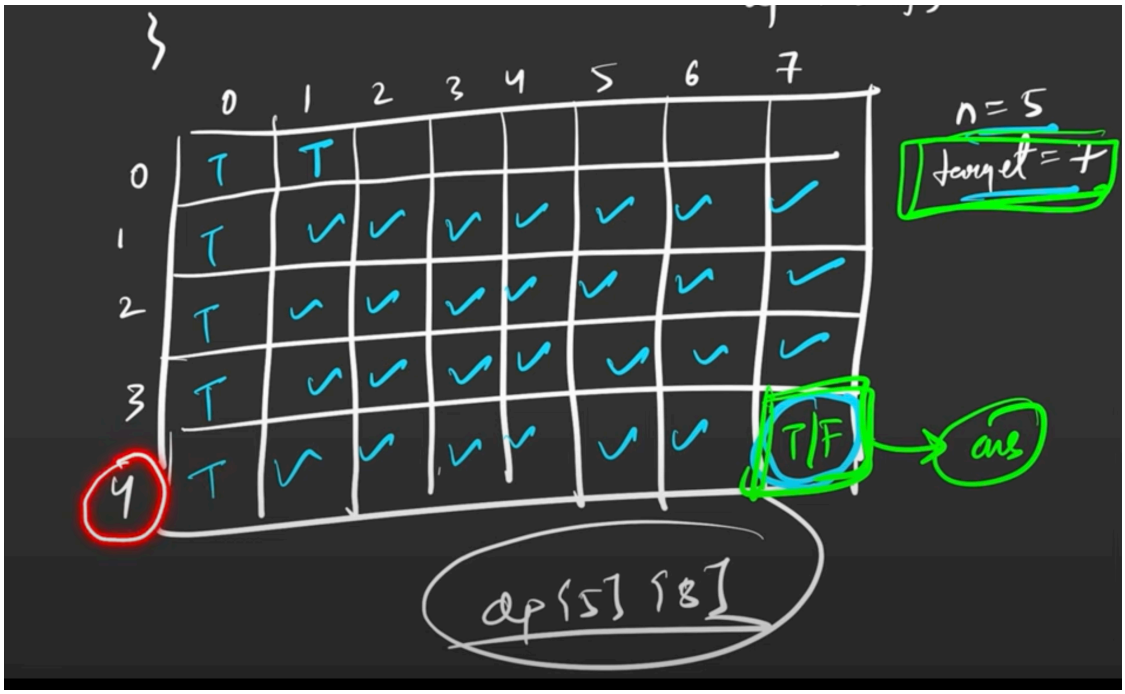
416. Partition Equal Subset Sum

Approach: memoization and tabulation
This problem is exactly similar to previous question. Because if the

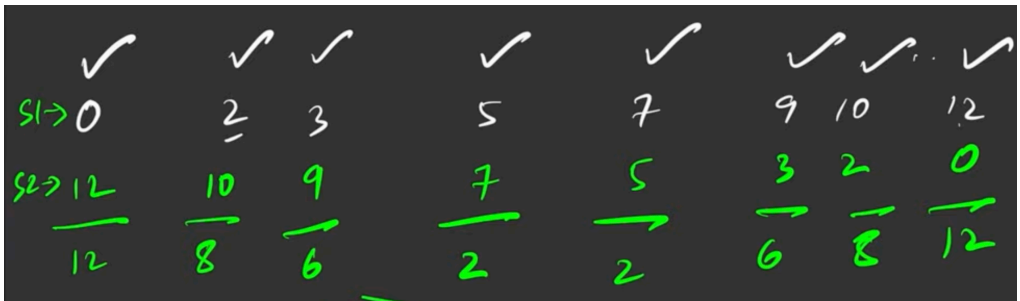
Sum of array is odd then definitely 2 partition is not possible but if the sum is even then devised it by 2 and you get your target. Now apply the similar logic to previous question and your question is become Subset sum equal to K.
Time complexity is O(n) + O(n * k) here n extra for first finding the sum of entire array. And O(n * k) to store the tabulation.

Partition a set into two subsets such that the difference of subset sums is minimum. (Coding Ninja)

Approach: Dynamic Programming Tabulation.
In the subset sum equal to k question we filled a dp table of size(n * k+1) if you remember then in that question we got a answer at dp[n-1][k]. How that question was made the answer is we filled a dp table from top to bottom in that if the particular sum from 0 to K is possible or not using the array. Which is looks like:



Now the logic is if we have the total sum and this last row of dp table then definitely we able to partition the array into two part for example. In the given image total sum is 7 of array and we are at index [4] [4] and it's value is true in dp table then the valid partition of that array is 4 and (7-4) = 3. Using this we definitely able to find the minimum difference between two partition of the array.
Now according to hint can we need to iterate entire last row?
Ans is no. why?



According to above example till 5 we has the answers like 12, 8, 6, 2 but after that answer was repeated because if you see the first and last index then absolute difference of (0-12) and (12-0), difference between (2-10) and (10-2) is same that's why this repeatation will occurs. So for find the min answer we have to iterate only k/2. So we can able to reduce the time complexity.
Total time complexity is : O(n *k) + O(n) + O(k/2) and space is O(n *k) to store dp table. Distribution of time complexity is: O(n*k) to fill the dp table. O(n) to find the total sum of array and O(k/2) to find the min difference of all the partition of array.

Count Subsets with Sum K
Approach: Recursion and memoization, dynamic programming
Logic of pick and notpick and adding them is a logic to solve this question.
With int return type of recursion.
Algo of recursion and memoization.

```
int solve(int idx, vector<int>&nums, int target, vector<vector<int>> &dp){
    if(target == 0){
        return 1;
    }
    if(idx == 0){
        if(nums[0] == target){
            return 1;
        }
        return 0;
    }
         if(dp[idx][target] != -1) return dp[idx][target];
     int pick = 0;
    int notpick = solve(idx-1, nums, target, dp);
    if(nums[idx] <= target) pick = solve(idx-1, nums, target-nums[idx], dp);
         return dp[idx][target] =  (pick+notpick) % mod; }
```

Time complexity is O(n * target) space is also same with extra n recursive stack space.
Tabulation:

```
int n = nums.size();
        vector<vector<int>>dp(n, vector<int>(target+1, 0));
        // return solve(n-1, nums, target, dp);
        for(int i=0;i<n;i++){
                dp[i][0] = 1;
        }
        if(nums[0] <= target) dp[0][nums[0]]=1;
        for(int i=1;i<n;i++){
                for(int j=1;j<=target;j++){
                        int pick = 0;
                        int notpick = dp[i-1][j];
```

```
                    if(nums[i]<=j) pick = dp[i-1][j-nums[i]];
                    dp[i][j] = (pick+notpick) % mod; }}
        return dp[n-1][target];
```

Remaining time and space is same but we reduce the recursive stack space.

Partitions With Given Difference

Here we have to partition entire array into two subset s1 and s2 where s1 is greater then the s2 and difference between s1-s2 will equal to d. If we apply simple mathematics over here the:
s1 + s2 == totals of array
And if totals -d /2 then partition is not possible so return 0.
Another edge case is if the totalsum - d is negative then also return the 0.
Now, s1 = totalsum - s2 - d
And s2 = totalsum - d /2;
And the question is simple s2 is our target and we have to just find the number of subsets which is equal to s2. And question is simple as our previous question.
Here the important edge case is number 0's are also there in input. so, we have to handle it. Our base case of recursion is :

```
if(idx == 0){
        if(tar == 0 || arr[0]==0) return 2;
        if(tar == 0 || arr[0] == tar) return 1;
        return 0;
    }
```

and
Entire logic is same as previous.

0/1 Knapsack (GFG):

Approach : recursion memoization, tabulation, space optimised in single array.
It is standard question of Subject DAA. We know here the greedy approach doesn't work that's why we have to follow a recursion to try every possible combinations. And to reduce the overlapping subproblems we also use the memoization and then we convert it into the tabulation then we can optimised the space into just single W size array using striver's best method.
Recursion and memoization Algo:

```
int solve(int idx, int W, int wt[], int val[], vector<vector<int>>&dp){
    if(idx == 0){
        if(wt[0] <= W){
            return val[0];
        }
        return 0;
    }
    if(dp[idx][W] != -1) return dp[idx][W];
```

```
    int pick = -1e8;
    int notpick = solve(idx-1, W, wt, val, dp);
    if(W >= wt[idx]) pick = solve(idx-1, W-wt[idx], wt,val, dp) + val[idx];
    return dp[idx][W] = max(pick, notpick);
}
```

Time complexity is O(n * W) space is also O(n * W) and with extra O(n) recursive stack space.

Tabulation Algo:

```
 vector<vector<int>>dp(n, vector<int>(W+1, 0));
    for(int i=wt[0]; i<= W; i++){
        dp[0][i] = val[0];
    }
    for(int i=1;i<n;i++){
        for(int j = 1;j<=W;j++){
            int pick = -1e8;
            int notpick = dp[i-1][j];
            if(j >= wt[i]) pick = dp[i-1] [j-wt[i]] + val[i];
            dp[i][j] = max(pick, notpick); } }
    return dp[n-1][W];
```

Here we optimised extra O(n) stack space.

We can fill the dp singleD array from left to right and also takes another temp array to store the current row for space optimisation. But there is no rule always you can start from left, start from right it will also work and it will allows to optimise space in only Single 1d array of size Weight.

```
vector<int>dp(W+1,0);
    for(int i=wt[0]; i<= W; i++){
        dp[i] = val[0];
    }
    for(int i=1;i<n;i++){
        for(int j = W;j>=0;j--){
            int pick = -1e8;
            int notpick = dp[j];
            if(j >= wt[i]) pick = dp[j-wt[i]] + val[i];
            dp[j] = max(pick, notpick); } }
    return dp[W];
```

322. Coin Change

Approach: Dynamic programming

The important thing here is, you can use each and every coin infinite time that's why in pick recursion call do not make idx-1. In the notpick recursive call by default it will made -1 if change is not possible.

*Important : whenever you see the statements like infinite supply or Multiple use do not make idx-1 in pick recursive call.

Recursive Memoization algo:

```
int solve(int idx, vector<int>& coins, int target, vector<vector<int>>&dp){
```

```
    if(idx == 0){
        if(target%coins[0] == 0){
            return target/coins[0];
        }
        return 1e9;
    }
    if(dp[idx][target] != -1) return dp[idx][target];
    int pick = 1e9;
    int notpick = solve(idx-1, coins, target, dp);
    if(target>=coins[idx]) pick = solve(idx, coins, target-coins[idx], dp) +1;
    return dp[idx][target] = min(pick, notpick); }
```

Conversion of memoization to tabulation is very easy.
Time complexity is Not fixed here. Because at each index you can use that elements infinite time until amount become 0. And that's why here we do not able to give theoretical time. It is just Exponential. Space is O(n * amount).

494. Target Sum

Approach : Memoization, Tabulation
If you remember, coding ninja's question Partitions With Given Difference Then it is a exact same question because it has also requirement to divide a array into two subsets such that s1 - s2 == target. We have to just change the language otherwise entire question will same. Because question itself is said that one subset has all the negative elements and another subset will have all the positive elements on it.

518. Coin Change II

Approach: Dynamic Programming
In coin change I, we have to find the number of minimum coins that's why we return number of min coins but, here we have to find number of ways to gain particular money. So instead of coin here we write a base case in such a way so it will return the total ways. And remember here also is a infinite supply of the Coins.
Memoization Algo:

```
int solve(int idx, vector<int>& coins, int target,vector<vector<int>>&dp){
    if(idx == 0){
        if(target%coins[0] == 0){
            return 1; }
        return 0; }
    if(dp[idx][target] != -1) return dp[idx][target];
    int pick = 0;
    int notpick = solve(idx-1, coins, target, dp);
    if(coins[idx] <= target) pick = solve(idx, coins, target - coins[idx], dp);

    return dp[idx][target] = pick+notpick; }
```

Time complexity is O(n * amount) space is also same with extra recursive stack space.

Unbounded Knapsack (Coding Ninja)

Approach: Dynamic programming
In the 0/1 knapsack the only changes is you do not make idx-1 in pick recursive call because in question it is given that the weights of array is a infinite supply. And we know one Thumb rule 👍 which is whenever you see such statements like infinite supply, multiple use then please do not make idx-1 in pick recursive call.
Now the important thing is how to write a base cases in such kind of question. So do not think different think simple.
For example if you are at index 0 and weight is 3, and your remaining bag capacity is 8 so how to tackle it?
You know that there is a infinite supply of the weights so how many times you can able to take that weight in your bag? Ans is 8 / 3 which is 2. So return 2 * profit of that weight. as simple as that.
Time and space complexities are same as our standard dp problems.

Rod Cutting (Coding Ninja)

Approach: Dynamic programming.
We have to maximise the cost for selling the cutted stick that's why we have to try all the possible combination. If you take n as a target then again it is our target sum problem. But here the base case is only different. because when we are at index 0 so the total cost we gain is number of pics of rod * cost of index 0. For example if we are at idx 0 and cost is 4 and n is 5 then definitely we can able to cut that rod in 5 different parts so cost is n * cost[0] in that case.
Memoization Algo:

```
int solve(int idx, vector<int>&price, int target, vector<vector<int>>& dp){
        if(idx == 0){
                        return  target * price[idx];
        }
        if(dp[idx][target] != -1) return dp[idx][target];
        int pick = 0;
        int notpick = solve(idx-1, price, target, dp);
        if(target >= idx+1)pick = solve(idx, price, target-(idx+1), dp) + price[idx];
        return dp[idx][target] = max(pick, notpick); }
```

Making a tabulation from memoization is easy.
Time complexity is O(n * n) space is also (n * n). But id=f we use the tabulation then we able to reduce the recursive stack space.

1143. Longest Common Subsequence

Approach : Dynamic Programming.

Till now we will solve the questions of target sum or array subsequence. But now we will going to solve the Dp on string problems. LCS is very popular and easy question. The only logic is how to write base case and how to express our recurrence in terms of index.
For writing the base case if idx1 and idx will lesser then 0 then return 0. It will works perfectly fine but when we convert memoization to tabulation then it will creates the problem for us. So we write a base case using shifting operation. So we write a base case in terms of I==0 || j==0. And for comparing the string we use s1[I-1] && s2[j-1].
To express our recurrence in terms of index we take two indexes I and j for two strings. If s1[I-1] == s2[j-1] then we call recursive function for I-1 and j-1. And if not match then we call function 2 times 1 for I-1, j and 1 for I, j-1 and whichever will gives max answer we take that in our not match variable.
Memoization algo:

```
int solve(int i, int j, string &text1, string &text2, vector<vector<int>>&dp){
    if(i == 0 || j == 0){
        return 0;
    }
    if(dp[i][j] != -1) return dp[i][j];
    int match = 0, notmatch=0;
    if(text1[i-1] == text2[j-1]){
        match = 1 + solve(i-1, j-1, text1, text2, dp);
    }
    else{
        notmatch = max(solve(i-1, j, text1, text2, dp),
                       solve(i, j-1, text1, text2, dp));
    }
    return dp[i][j] = max(match, notmatch); }
```

Time complexity is O(n1 * n2) space is also O(n1 * n2) with O(n1 + n2) recursive stack space. Why (n1 + n2) because we have an two strings for comparison.
Space optimised two array tabulation:

```
int n1 = text1.size(), n2 = text2.size();
        vector<int>prev(n2+1,0), curr(n2+1, 0);
        for(int i=1;i<=n1;i++){
            for(int j=1;j<=n2;j++){
                int match=0, notmatch=0;
                if(text1[i-1] == text2[j-1]){
                    match = 1 + prev[j-1];
                 }
                else{
                    notmatch = max(curr[j], curr[j-1]);
                 }
                curr[j] = max(match, notmatch);
            prev = curr;
```

```
        return prev[n2];
```

Time complexity still remains same but we optimised lot of space.

Longest Common Substring (GFG):

Approach: dynamic programming

This problem is similar to like the LCS but the important thing over here is we have to find longest common continues substring that's why we make slightly modifications in LCS approach.

The tabulation of LCS will said that if curr character of two strings are same then I will add + 1in diagonal and take it as dp[I][j] and. If it is not matched then it will take maximum from upper cell and left cell. Which is not work at here. So according to our approach in this question each character of string will says that if it matches with another string then I will add 1 with my upper left diagonal other wise make my cell as 0.

And finally return the maximum cell among entire table.

```
int n = S1.size(), m = S2.size();
        vector<vector<int>>dp(n+1, vector<int>(m+1,0));
        int ans = 0;
       for(int i=1;i<=n;i++){
          for(int j=1;j<=m;j++){
             if(S1[i-1] == S2[j-1]){
                dp[i][j] = dp[i-1][j-1] + 1;
                ans = max(ans, dp[i][j]);
             }
Else{ dp[i][j] = 0;} }}
       return ans;
```

Time complexity is O(n * m) space is also same.

And again if you want to optimise the space then it also possible using two array.

516. Longest Palindromic Subsequence

Approach: LCS

Very first approach is comes in our mind is solve this question using recursion and each time check for it's reverse is present in out string or not. But but but, we all know that if the subsequence is present in our string then for ans that reverse order will also have to present in our string that's why we use the LCS logic here. We can take the original string and it's reversal and then we can find the LCS for both the string which is our valid answer.

1312. Minimum Insertion Steps to Make a String Palindrome

Approach: LCS

According to leetcode hint if we can reverse a particular string and find the LCS of that string. And if we subtract that LCS from the length of a string then it is a answer of our question which is number of insertion operation required to to make string palindrome. N - LCS of that string.

Minimum Insertions/Deletions to Convert String A to String B (Coding Ninja)

Approach: LCS

Again it is a interesting problem ever to solve using LCS. In worst case if all the characters of string 1 will different from the string 2 then number of operation which requires to make string1 equal to string 2 is, delete all the elements from string1 and add all the elements of string2 to string1. So according to my observation, find the LCS of two string. Now subtract LCS from n and m. And add them which is your answer. Where n and m is length of string1 and string2.

1092. Shortest Common Supersequence

Approach: LCS

Again you might thinking that this is a hardest problem of leetcode. But trust me if you know the LCS in depth then it will easiest problem ever for you. Suppose if we have the task to find the length of the shortest common subsequence. Then if you find out the LCS then the length of that Supersequence is (n+m) - lcs that's for sure. Because you can remove the lcs from the union of the string then definitely the superset of that strings is (n+m) - lcs which are minimum because we subtracted the lcs from it. Not this is all about the length of super sequence. But, what if we want to print that supersequence.

If you remember for printing the lcs we can follow the backtracking of dp table. And we know if str1[I-1] == str2[j-1] then we added +1 in diagonally so we add that character in string and if string is not matched then we go in maximum of upper cell and left cell if upper cell is greater then left cell then we move upper and add that str1's character in answer. Otherwise we move in left and add the str2's character in our answer.

We made a loop which is while(I>0 && j>0) that's for sure. And we know there is still some characters are left if either I or j will not meets the 0 so we again check that indexes to add remaining characters. And finally we reverse the ans and return because we iterate from lower to upper in table.

Algo:

After finding LCS

```
int len = dp[n][m];
        int i = n, j= m;
        while(i>0 && j>0){
            if(str1[i-1] == str2[j-1]){
                ans += str1[i-1];
                i--;
                j--;
            }
            else if(dp[i-1][j] > dp[i][j-1]){
                ans+=str1[i-1];
                i--;
```

```
            }
            else{
                ans+=str2[j-1];
                j--;
            }
        }
        while(i>0){
            ans+=str1[i-1];
            i--;
        }
        while(j>0){
            ans+=str2[j-1];
            j—; }
cout<<(n+m) - len;
reverse(ans.begin(), ans.end());
return ans;
```

115. Distinct Subsequences

Approach: Dynamic programming

If we remember the logic of pick and not pick the element from the string 1. For example: s = "babgbag", t = “bag”. From t you can take ‘g’ from and take 2 times ‘g’ from s because there are two occurrences of g in s. So how to do this? Simple yarr use the approach of take and not take. Which is make the work easy.

Here only changes is you can return take + not take in take recursive call because we have to count the strin g matches.

Memoization Algo:

```
int solve(int i, int j, string s, string t, vector<vector<int>>&dp) {
        if(j == 0){
            return 1;
        }
        if(i == 0){
            return 0;
        }
        if(dp[i][j] != -1) return dp[i][j];
        if (s[i-1] == t[j-1]) {
            return dp[i][j] = solve(i - 1, j - 1, s, t, dp) + solve(i - 1, j, s, t, dp);
        }
        return dp[i][j] = solve(i - 1, j, s, t, dp); }
```

Time complexity is O(n *m) with dp space of O(n * m) and O(n + m) recursive stack space.

For converting this memoization to tabulation we know that for each j ==0 dp value is 1. And here we optimised the space also. So in each for loop make prev[0] = 1;

```
int n = s.size();
        int m = t.size();
        vector<double>prev(m+1, 0),curr(m+1, 0);
        for(int i=1;i<=n;i++){
            prev[0] = 1;
            for(int j=1;j<=m;j++){
                if (s[i-1] == t[j-1]) {
                curr[j] = prev[j - 1] + prev[j];
                }
                else{
                curr[j] = prev[j]; } }
            prev = curr; }
        return (int)prev[m];
```

Here we reduce the recursive stack space as well n * m space into O(2m) arrays.

72. Edit Distance

Approach: Dynamic Programming

This is another interesting question of string matching. If we find the exploration ways in this question then it is 1) insert 2)delete 3) replace. This operations are performed when the character of string is not matching. And if characters are matching then we have to just make I-1 and j-1 in recursion. That is fine but the hardest thing in such a questions is writing the base cases. How to think base cases for this question. So, let's understand with example: if we has the string1 as "horse" and string2 is "ros" now lets we are at idx0 in string 2 so how many remaining operation is required? For that answer is number of remaining characters in string1. Same as for string 1 if we are at index 0 of string 1 and string 2 is still some characters on it. so the remaining characters of string2 is our ans. So the base case is now become simple for I==0 return j and for j == 0 return I.

*remember one thing, here we can not alter the given strings we hypothetically just playing with indexes.

Memoization algo:

```
int solve(int i, int j, string word1, string word2, vector<vector<int>> &dp){
    if(i == 0){
        return j;
    }
    if(j==0){
        return i;
    }
    if(dp[i][j] != -1) return dp[i][j];
    if(word1[i-1] == word2[j-1]){
        return dp[i][j] = solve(i-1, j-1, word1, word2, dp); }
    else{
    int insert = solve(i, j-1, word1, word2, dp) + 1;
```

```
    int remove = solve(i-1, j, word1, word2, dp) + 1;
    int replace = solve(i-1, j-1, word1, word2, dp) + 1;
    return dp[i][j] = min(insert, min(remove, replace)); } }
```

Time and space complexity is similar to our standard dp problems.

Tabulation algo:

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

            for(int i=1;i<=n;i++){
                for(int j=1;j<=m;j++){
                    if(word1[i-1] == word2[j-1]){
                        dp[i][j] = dp[i-1][j-1];
                    }
                    else{
                    int insert = dp[i][j-1] + 1;
                    int remove = dp[i-1][j] + 1;
                    int replace = dp[i-1][j-1] + 1;
                    dp[i][j] = min(insert, min(remove, replace)); } }
        return dp[n][m];
```

*To convert that base cases to tabulation do not think different follow the same process. Which is for I=0 j will be anything and for j-0 I will be anything. And your tabulation is ready

Further space optimisation is also possible.

44. Wildcard Matching

Approach: Dynamic Programming

We know that '?' Will able to replace only one character. So that's why if two characters of a string is matches or the character of pattern is '?' Then we reduce both indexes I and j. If the characters are not matched and the current character of a pattern is '*' then we know it will able to replace all the characters of a target string. So here we confuse because we actually don't know whether we take or not take the '*' for replacing the character. And here the actual logic is come into the picture. This is a subproblem of our dp which is take or not take and we had the bool function so return or operation of take and not take.

If first condition is not true and second is also false then return false.

How to write base case?

Again do not think different. Just think what happen when we arrive at index 0 of two string.

1) when I==0 and j is also 0 then we say that string is matching so simply return true.

2) When j==0 and I is not 0 means there are still some characters are there in the string 1 then return false.
3) If I==0 && j is not 0 then go through remaining characters of pattern string and if all the characters are ‘*’ then return true otherwise false.

Memoization Algo:

```
bool solve(int i, int j, string s, string p, vector<vector<int>>&dp){
    if(i == 0 && j == 0){
        return true;
    }
    if(j == 0 && i > 0){
        return false;
    }
    if(i ==0 && j > 0){
        for(int k=0; k<j; k++){
            if(p[k] != '*') return false;
        }
        return true;
    }

    if(dp[i][j] != -1) return dp[i][j];

    if((s[i-1] == p[j-1]) || (p[j-1] == '?')){
        return dp[i][j] = solve(i-1, j-1, s, p, dp);
    }
    else if(p[j-1] == '*'){
       bool pick  = solve(i-1, j, s, p, dp);
       bool notpick = solve(i, j-1, s, p, dp);
       return dp[i][j] = pick || notpick;
    }
    else{
        return dp[i][j] = false;
    }
```

Time complexity is O(n * m) space is O(n * m) to store dp and O(m + n) for recursive stack space.

Now convert this memoization into tabulation.

```
vector<vector<bool>> dp(n + 1, vector<bool>(m + 1, false));
    dp[0][0] = true;
    for (int j = 1; j <= m; j++) {
        dp[0][j] = (p[j - 1] == '*') && dp[0][j - 1];
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s[i - 1] == p[j - 1] || p[j - 1] == '?') {
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] == '*') {
```

```
            bool pick = dp[i - 1][j];
            bool notpick = dp[i][j - 1];
            dp[i][j] = pick || notpick;
        } else {
            dp[i][j] = false; } } }
    return dp[n][m];
```

Space optimisation will also possible but it will little bit tricky in this case.

121. Best Time to Buy and Sell Stock

take mini as prices[0] and profit as 0. Now iterate the prices array and if prices [I] > the mini then make profit = max(profit, prices[I] - mini) and at each step update the mini as mini = min(mini, prices[I]).
Time complexity is O(n).

122. Best Time to Buy and Sell Stock II

Approach: dynamic programming
Simple approach does not work here because we are not allow to buy and sell only one time in all the days. Here you can multiple time buy and sold the stocks. And it has the multiple ways to buy and sell the stocks that's why for trying to multiple ways we can use the recursion. Here there are two changing variable which is index and variable buy which can tell you whether you allow to buy or not? And for base case it is a straight forward. When you come at index n means out of the bound simply return 0. And for buying and selling a stocks we have two option either buy or not buy and sell or not sell and whichever will gives the max return that.
Memoization Algo:

```
Int solve(int idx, int buy, vector<int>&prices, vector<vector<int>>&dp){
if(idx == n) return 0;
if(dp[idx][buy] != -1) return dp[idx][buy];
if(buy == 1){
Int take = -price[idx] + solve(idx+1, 0, prices, dp);
Int not take = solve(idx+1, 1, prices, dp);
Return max(take, not take);
}
Else{
Int sell = prices[idx] + solve(idx+1, 1, prices, dp);
Int notsell = solve(idx+1, 0, prices, dp);
dp[Idx][buy] = max(sell, not sell);
} }
```

Time complexity is O(n * 2) space is also O(n * 2) with O(n) extra recursive stack space.
Tabulation algo:

```
Vector<vector<int>>dp(n+1, vector<int>(2,0));
```

Here the base case is tells us when idx = n dp[I][0] = dp[I][1] = 0. But we already assigned all the elements of dp table as 0 so there is no need.

```
for(int I=n-1, I>=0;i—){
for(int j=1;j>=0;j—){
if(j == 1){
            int take = -prices[i] + dp[i+1][0];
             int nottake = dp[i+1][1];
          dp[i][j] = max(take, nottake);
         }
         else{
             int sell = prices[i] + dp[i+1][1];
             int notsell = dp[i+1][0];
             dp[i][j] = max(sell, notsell); } } }
    return dp[0][1];
```

Here we able to reduce the recursive stack space.

*remember guys till now the space optimisation is optional. But, for upcoming few problems the space optimisation is required.

```
vector<int>prev(2, 0), curr(2,0);
    for(int i=n-1;i>=0;i--){
        for(int j=1;j>=0;j--){
            if(j == 1){
            int take = -prices[i] + prev[0];
            int nottake = prev[1];
          curr[j] = max(take, nottake);
         }
         else{
            int sell = prices[i] + prev[1];
            int notsell = prev[0];
            curr[j] = max(sell, notsell); } }
        prev = curr;
    }
    return prev[1];
```

For space optimisation will take 2 1d array with size 2. You can also allow to take 4 variables also which is also gives the same answer.

123. Best Time to Buy and Sell Stock III

This is a slight different problem then the previous. Because in stock 1 problem we are allow to make only 1 transaction, in stock 2 problem we are allow to take infinite transactions and in this problem we are allow to make only 2 transactions. If you remember our knapsack problem then it will allows up certain amount of limited knapsack to fill the weights. Then it is a exact same problem. Because we have a 2 transactions limit. So we take another variable called limit along with the idx and buy.

Now the question is when the transaction is complete?

You can buy a stock and sell it then the one transaction is complete and you can allow this process two times.

Mmeoization algo:

```
if(idx == prices.size() || limit == 0){
        return 0;
    }
    if(dp[idx][buy][limit] != -1) return dp[idx][buy][limit];
    if(buy == 1){
        int take = -prices[idx] + solve(idx+1, 0, limit, prices, dp);
        int nottake = solve(idx+1, 1, limit, prices, dp);
        return dp[idx][buy][limit] = max(take, nottake);
    }
    else{
        int sell = prices[idx] + solve(idx+1, 1, limit-1, prices, dp);
        int notsell = solve(idx+1, 0, limit, prices, dp);
        return dp[idx][buy][limit] = max(sell, notsell);
    }
```

Time complexity is O(n * 2 * 3) and space is also O(n * 2 * 3) with the extra O(n) auxiliary stack space. And remember one thing here there are three changing variables so we take 3D dp.

Now let's convert this memoization into tabulation:

```
vector<vector<vector<int>>>dp(n+1, vector<vector<int>>(2,
vector<int>(2+1,0)));
        for(int ind = n-1; ind>=0; ind--){
        for(int buy = 1; buy>=0; buy--){
            for(int limit=2; limit>=1; limit--){

                if(buy==0){// We can buy the stock
                    int take =  -prices[ind] + dp[ind+1][1][limit];
                    int nottake = dp[ind+1][0][limit];
                    dp[ind][buy][limit] = max(take, nottake);
                 }

                else{// We can sell the stock
                    int sell =  prices[ind] + dp[ind+1][0][limit-1];
                    int notsell = dp[ind+1][1][limit];
                    dp[ind][buy][limit] = max(sell, notsell); }. } } }
        return dp[0][0][2];
```

Now can we able to Reduce further time complexity ? Yes

```
        for(int ind = n-1; ind>=0; ind--){
        for(int trans = 0; trans<=3; trans++){
                if(trans % 2 == 0){// We can buy the stock
                    int take =  -prices[ind+1] + dp[ind][trans];
                    int nottake = dp[ind+1][trans];
                    dp[ind][trans] = max(take, nottake);
                 }
                else{// We can sell the stock
                    int sell =  prices[ind] + dp[ind+1][trans];
```

```
                int notsell = dp[ind+1][trans];
                dp[ind][trans] = max(sell, notsell); } } }
        return dp[0][3];
```

Solution 2:

This is a approach where we can use the 3d concept in that we take three changing variable. But there is a solution in which we can use only two changing variable.

So the idea is take one variable called transaction id with max value as 4 means at index 1 and 3 you can buy the stock and index 2 and 4 sell the stock.

Let's see the memoization code of that approach:

```
int solve(int idx, int trans, vector<int>& prices, vector<vector<int>>&dp){
    if(idx == prices.size() || trans == 4){
        return 0;
    }
    if(dp[idx][trans] != -1) return dp[idx][trans];
    if(trans % 2 == 0){
        int take = -prices[idx] + solve(idx+1, trans+1, prices, dp);
        int nottake = solve(idx+1, trans, prices, dp);
        return dp[idx][trans] = max(take, nottake); }
    else{
        int sell = prices[idx] + solve(idx+1, trans+1, prices, dp);
        int notsell = solve(idx+1, trans, prices, dp);
        return dp[idx][trans] = max(sell, notsell); }
```

This will take only O(n * 4) time and space complexity with auxiliary (n) space. Now you can also convert this memoization to tabulation which again not takes the auxiliary space. And you also able to reduce further space in only 2 O(4) size 1d ARRAY.

188. Best Time to Buy and Sell Stock IV

Approach: Dynamic programming

To solve this question we use our previous approach which is the transaction numbers where on even index you are allow to buy stock and at odd index you are allow to sell a stock. And in that we don't use the 3D DP we just use the (n * k*2) 2D array to memoize.

Memoization Algo:

```
int solve(int idx, int trans, vector<int>& prices, vector<vector<int>> &dp){
    if(idx == prices.size() || trans == 0){
        return 0;
    }
    if(dp[idx][trans] != -1) return dp[idx][trans];
    if(trans %2 == 0){
        int buy = -prices[idx] + solve(idx+1, trans-1, prices, dp);
        int notbuy = solve(idx+1, trans, prices, dp);
        return  dp[idx][trans] = max(buy, notbuy);
```

```
    }
    else{
        int sell = prices[idx] + solve(idx+1, trans-1, prices, dp);
        int notsell = solve(idx+1, trans, prices, dp);
        return  dp[idx][trans] = max(sell, notsell); }}
```

Time complexity is O(n * 2*k) space is also O(n * 2*k) + auxiliary stack space.
Space optimised Tabulation Algo:

```
vector<int> prev (transactions+1, 0), curr(transactions+1, 0);
        for(int idx=n-1; idx>=0;idx--){
            for(int trans=transactions; trans>=1; trans--){
                if(trans %2 == 0){
                int buy = -prices[idx] + prev[trans-1];
                int notbuy = prev[trans];
                curr[trans] = max(buy, notbuy);
            }
            else{
                int sell = prices[idx] + prev[trans-1];
                int notsell = prev[trans];
                curr[trans] = max(sell, notsell); } }
            prev = curr;
        }
        return prev[transactions];
```

309. Best Time to Buy and Sell Stock with Cooldown

Approach: Dynamic Programming
The approach of this problem is exactly similar to Stock - 2 problem but the slight difference over here is you do not allow to buy a stock just after selling the stock which is a cooldown day. So you can easily make idx+2 after selling the stock and you are able to solve this particular question.
Time complexity is O(n) because in tabulation you run one for loop and another loop from 0 to 1 which is almost constant.

Now, 😇 the important question is Can we able to optimise space?

Ans is No. Why because we know that whenever we see the idx+1, idx-1 then we able to optimise the space otherwise no. and here there is a condition of idx+2.
And if you want to optimise the space then you can take 3 1d array instead of 2. Because you have to remember the previous to previous day transaction also.

714. Best Time to Buy and Sell Stock with Transaction Fee

Approach: Dynamic Programming
When we complete one transaction? If you know this ans then this question is done. When we purchase one stock and sold then we complete 1 transaction and when you complete 1 transaction please subtract the given

fee from it. And you are done. Use the stock 2 approach. Because here infinite transactions are allowed.

300. Longest Increasing Subsequence

This problem can be solved without recursion but we still try recursion because it can make more understanding about this problem. We can maintain two changing variable in recursion one can hold the curr element and one can holds the prev smaller element and all will done.
We also take the prev as -1 because for index 0 there is no prev guy. When you don't pick the element then your prev guy does not change but when you pick the element for your subsequence then your curr index become your prev guy.
Memoization algo:

```
Int solve(int idx, int prev, vector<int>&nums, vector<vector<int>>&dp){
if(idx == nums.size()){
Return 0;
}
if(dp[idx][prev+1] != -1) return dp[idx][prev+1] ;
Int notpick = solve(idx+1, prev, nums, dp);
Int pick = 0;
if(prev == -1 || nums[idx] > nums[prev]){
Pick = solve(idx+1, idx, nums, dp);
}
Return dp[idx][prev+1] = max(pick, notpick); }
```

Time complexity is O(n * n) means N^2 and space is also O(n * n) with auxiliary space O(n).
Tabulation Algo:
When we convert this memoization to tabulation please remember that prev does not start with n-1 prev will always start from index-1 because name itself said that prev means prev guys of current index.

```
for(int idx = n-1; idx>=0; idx--){
        for(int prev = idx-1; prev>=-1; prev--){
            int notpick = prev1[prev+1];

            int pick = 0;
            if (prev == -1 || nums[idx] > nums[prev]){
            pick = prev1[idx+1]+1;
            }
            curr[prev+1] = max(pick, notpick);
        }
        prev1 = curr;
    }
    return prev1[0];
```

We still used a O(n + n) space can we able optimise further? then ans is yes.

When you are at index I then check all its previous guys can they are holds the property of increasing subsequence.
Take one N size Dp array with all the values as 1 because each and every element will generate at least 1 size subsequence. Now iterate the array from 0 to n and for prep 0 to I and check the conditions and whichever the largest element in app array is my answer.

```
Int ans = 1;
for(int I=0 ;I<n;i++){
for(int prev = 0; prev<I;prev++){
if(nums[prev] < nums[I]){
dp[i] = max(dp[i], dp[prev] +1);
Ans = max(ans, dp[i];s
} } }
Return ans;
```

This approach will still use the O(N^2) time and O(n) space.
Now we optimise this O(n^2) into n using the term called Lower Bound.
The intuition to solve this question is for ex: [10,9,2,5,3,7,101,18] so for this array you can partition it in increasing order for example partition is:
[10] , [9], [2, 5], [2, 3, 7, 101] , [18] and larger one is our answer. So this is gives you a correct answer but it is not feasible solution because it will use the lots of space. So for that we use only single array and whenever new elements is occurred and it is not greater then the last element of our array then you can overwrite that element in it's lower bound. This will not gives you the exact sequence of LIS but it gives the exact answer of sequence length. So we use that.
Algo:

```
vector<int>ans;
        ans.push_back(nums[0]);
        for(int i=1;i<n;i++){
          if(nums[i] > ans.back()) ans.push_back(nums[i]);
          else{
             int idx = lower_bound(ans.begin(), ans.end(), nums[i]) - ans.begin();
             ans[idx] = nums[I]; } }
        return ans.size();
```

Time complexity is O(n * logn) because lower bound is used a binary search and O(n) space complexity.

Maximum sum increasing subsequence (GFG):

Approach: Dynamic programming
If you know the logic od largest increasing subsequence then this problem is variation of that. Because here you do not have to find largest increasing subsequence but you have to find the largest sum increasing subsequence. But still logic remains same because in that problem when you find some larger element from previous then you are returning the pick recursive call with 1 and here instead of 1 you can return the value of array of that index.

Mmeoization algo:

```
int solve(int idx, int prev, int arr[], int n, vector<vector<int>>&dp){
    if(idx == n){
        return 0;
    }
    if(dp[idx][prev+1] != -1) return dp[idx][prev+1];
    int notpick = solve(idx+1, prev, arr, n, dp);
    int pick = 0;
    if(prev == -1 || arr[idx] > arr[prev]){
        pick = solve(idx+1, idx, arr, n, dp) + arr[idx];
    }
    return dp[idx] [prev+1] = max(pick, notpick);
}
```

Time complexity is O(n * n) and space is also (n * n) with O(n) auxiliary stack space.

Now you can try a further process like converting a tabulation and then further space optimisation.

368. Largest Divisible Subset

Approach: LIS

If you can observe carefully then there are some hints are there in the question itself. In the question you can get that you are allow to print answer in any order and second is all the elements are distinct. So now we will able to connect this question with LIS. If we sort this array and try to apply algorithm of LIS then it will works we have to just print LIS vector. The only shuttle change is in that you can check the increasing order but here you have to check the divisibility. And why we said this will work confidently because let's say we have the sorted array like 1,2,4,8 here if 8 is divisible 4 then obviously 4 is also divisible by 8 and all the remaining elements also because when we added 4? When the 4 id divisible by 2 and when we added 2? When 2 is divisible by 1. So we are able to apply LIS logic Over here.

Algo:

```
int n = nums.size();
        sort(nums.begin(), nums.end());
        vector<int>dp(n,1);
        vector<int>hash(n);
        int lastidx = 0, maxi = 0;
        for(int idx=0;idx<n;idx++){
            hash[idx] = idx;
            for(int prev=0;prev<idx;prev++){
                if(nums[idx] % nums[prev] == 0 && dp[prev]+1 > dp[idx]){
                    dp[idx] =  dp[prev]+1;
                    hash[idx] = prev;
                }
```

```
            }
            if(dp[idx] > maxi){
                maxi = dp[idx];
                lastidx = idx;
            }
        }
        vector<int>LIS;
        LIS.push_back(nums[lastidx]);
        while(hash[lastidx] != lastidx){
            lastidx = hash[lastidx];
            LIS.push_back(nums[lastidx]);
        }
        return LIS;
```

Here answer will accepted in any order so you do not have to reverse the LIS array other wise you can.
Time complexity is O(n *n) + O(n) and space is O(3n).

1048. Longest String Chain

Approach: LIS
It is a exact similar problem to our LIS problem. Because in that we can find the LIS of integer numbers and here we have to find the increasing equal string with the character difference of 1. So for that logic is quite simple write a boolean function which can gives either string will matches the given conditions or not and apply the same DP logic Of LIS. Now this will woe=rks fine for some strings but it will not work at all. Why?
For example: [“xbc","pcxbcf","xb","cxbc","pcxbc"] in this example our also will start from index1 which is gives the answer 3 but it’s correct answer is 5. That’s why we first sort the string according to it’s length. And then apply the LOIS logic.
Algo:

```
bool str_compare(string s1, string s2){
    if(s1.size() != s2.size()+1) return false;
    int first = 0;
    int second = 0;
    while(first < s1.size()){
        if(s1[first] == s2[second]){
            first++;
            second++;
        }
        else{
            first++; } }
    return (first == s1.size() && second == s2.size());
}
static bool comp(string& str1, string& str2) {
    return str1.size() < str2.size(); }
```

```
int longestStrChain(vector<string>& words) {
        int n = words.size();
        sort(words.begin(), words.end(), comp);
        vector<int>dp(n, 1);
        int maxi = 1;
        for(int idx=0;idx<n; idx++){
            for(int prev=0;prev<idx;prev++){
              if(str_compare(words[idx], words[prev]) && dp[idx] < dp[prev]+1){
                  dp[idx] = dp[prev] + 1;
                  maxi = max(maxi, dp[idx]); } }
        return maxi;
```

Time complexity is: O(n log n) for sorting the string vector, O(n ^2 * length of strings) and space is O(n) to store the DP array.

Longest Bitonic Sequence (Coding Ninja):

Approach: LIS

We have funded the LIS from left to right which can gives the LIS. But what is we find the LIS from right to left then it will gives the longest Decreasing subsequence. And we know at each point there is a one common point so we make -1 at each point and which ever is maximum is our ans.

For example we have the array like : [1 2 1 3 4 3 2 1] in that LIS from the left is 4 which is at index 5 and LIS from the back side is 4 at index 4 from back side so we make 4+4-1 so ans is 7. And we know that 4 is common in both side so we make -1.

Algo:

```
//LIS from front
vector<int>dp(n, 1), revdp(n, 1);
          for(int i=0;i<n;i++){
                  for(int j=0;j<i;j++){
                          if(arr[i] > arr[j] && dp[i] < dp[j]+1){
                                  dp[i] = dp[j]+1; } } }
//LIS from back
for(int i=n-1;i>=0;i--){
                  for(int j=n-1;j>=i;j--){
                          if(arr[i] > arr[j] && revdp[i] < revdp[j]+1){
                                  revdp[i] = revdp[j]+1; } }
                  maxi = max(maxi, (dp[i] + revdp[i]) - 1);
                       }
```

Finally return maxi.

Time complexity is O(n^2 + n^2) space is O(2n).

673. Number of Longest Increasing Subsequence

Approach: LIS

The approach is take another array called count which can holds the number of count of maximum increasing numbers at each index. It will be 1 until the

dp[I] < 1+dp[j] but whenever the dp[I] == 1+dp[j] means the new guys are occurs which can said, I am also form a LIS so for that we have to make count[I] += count[j] and finally add all the count which can holds the LIS.
Algo:
Vetor<int> dp(n, 1), count(n,1);
for(int Idx=0; Idx<n;idx++){
for(int prev=0; prev<idx;prev++){
if(nums[prev] < nums[idx] && dp[idx] < dp[prev] +1){
dp[idx] = dp[prev] +1;
Maxi = max(maxi, dp[idx]);
count[I] = count[j] ;
}
Else if(nums[prev] < nums[idx] && dp[idx] == dp[prev] +1){ // shuttle change
count[I] += count[j];
} } }
Int ans = 0;
for(int I=0; I<n;i++){
if(count[I] == maxi) ans+= count[I];
}
Return ans;
Time complexity is O(n ^ 2)

Matrix Chain Multiplication (GFG):

Approach: Partitioning DP.
This is a best question ever to understands the Partitioning Dp in depth. We have given an array which holds the Dimensions of matrix if there are N elements are there in the array then there are N-1 matrix are possible that's we know. Now how to approach this question?
Step1-> start with index 1 and N-1 (in partitioning Dp express in terms of index I and j because we have to partition the entire array).
Step2-> now in recursion iterate entire array from I to j-1. And then multiply the current matrix dimensions and also add the two recursive call from I to k and k+1 to j so you can able to solve all the sub problems.
=> Recurrence of our question.
f(I, j){
if(I == j) return 0;
Int mini = 1e9;
for(int k=I;k<j;k++){
Int steps = arr[I-1] * arr[k] * arr[j] + f(i, k) + f(k+1, j);
Mini = min(mini, steps); }
Return mini.
This will takes the exponential time complexity. Can we Able to reduce it? Yes we can. Using the memoization.
Memoization algo:
int solve(int i, int j, int arr[], vector<vector<int>>&dp){

```
    if(i == j){
        return 0;
    }
    if(dp[i][j] != -1) return dp[i][j];
    int mini = 1e9;
    for(int k=i; k<j; k++){
        int steps = arr[i-1] * arr[k] * arr[j] + solve(i, k, arr, dp) +
                            solve(k+1, j, arr, dp);
        mini = min(mini, steps);
    }
    return dp[i][j] = mini; }
```

Time complexity is O(n * n * n) space is O(n * n) with extra auxiliary N stack space. Why n*n*n because we also run the extra for loop in the recursion.

Tabulation algo:

```
Vector<vector<int>>dp(n, vector<int>(n,0));
for(int I=n-1; I>=1; I- -){
for(int j=I+1; j<n;j++){
int mini = 1e9;
            for(int k=i; k<j; k++){
                int steps = arr[i-1] * arr[k] * arr[j] + dp[i][k] + dp[k+1][j];
                mini = min(mini, steps);
            }
            dp[i][j] = mini;
} }
Return dp[1][N-1];
```

Here we reduced the auxiliary stack space.

11547. Minimum Cost to Cut a Stick

Approach: Partitioning Dp

This is a one of the toughest question of partitioning dp because there is a hard logic on it. First important point is you can push 0 on front of the given vector and n on back side of the array. Now sort it. Now our answer is cuts[j+1] - cuts[i-1] plus the remaining partition of the array. Why cuts[j+1] - cuts[i-1] because we pushed the 0 at front and n at back so whenever you try this it will gives the current size of the partition array. For ex at first attempt the size of the stick is n-0 and like wise for all the cuts. And add the remaining partition of the array from I to k-1 and k+1 to j.

Memoization algo:

```
int solve(int i, int j, vector<int>& cuts,  vector<vector<int>> &dp){
    if(i > j){
        return 0;
    }
    if(dp[i][j] != -1) return dp[i][j];
    int mini = 1e9;
```

```
    for(int k=i; k<=j; k++){
        int cost = cuts[j+1] - cuts[i-1] + solve(i, k-1, cuts, dp) + solve(k+1, j, cuts,
dp);
        mini = min(mini, cost); }
    return dp[i][j] = mini;
}
```

Time complexity is O(n * n * n) and space is (n*n) to store the dp with extra auxiliary space.

Tabulation algo:

If you have the complete understanding of given recurrence then trust me guys tabulation is very easy for you.

```
cuts.insert(cuts.begin(), 0);
        cuts.push_back(n);
        sort(cuts.begin(), cuts.end());
        int sz = cuts.size();
        vector<vector<int>>dp(sz+1, vector<int>(sz+1,0));
        for(int i=sz; i>=1; i--){
            for(int j=1; j<=sz-2; j++){
            if(i>j) continue;
            int mini = 1e9;
            for(int k=i; k<=j; k++){
                int cost = cuts[j+1] - cuts[i-1] + dp[i][k-1] + dp[k+1][j];
                mini = min(mini, cost);
            }
            dp[i][j] = mini;
            }
        }
        return dp[1][sz-2];
```

312. Burst Balloons

Approach: Partitioning Dp

After seeing this question everyone can think like given in example means they can start to solve this question from top to bottom which is given in example 1: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

Everyone can start a question to solve from [3 5 8] to [3 8] and [8]. But my question is how to give the equation given which is for ith index coin is i-1 * I * I+1. if you apply the partitioning Dp then it will gives the wrong answers. Because partition dp will works on independent sub arrays and here we has the dependency. So how to approach this question? Ans is every where straight thinking doesn't work some time we have to think reverse. In above example we start from [3 5 8] to [3 8] and [8]. But what is we solve like [8] [3 8] and [3 5 8] so there is no dependency and we can easily able to solve using the Partition Dp. Noe first insert 1 on both side of the array so it will not

creates the problem in future and try to relate this problem with our Matrix chain multiplication (MCM) problem.
Memoization algo:

```
int solve(int i, int j, vector<int>& nums, vector<vector<int>>&dp){
    if(i > j){
        return 0;
    }
    if(dp[i][j] != -1) return dp[i][j];
    int maxi = -1e9;
    for(int k=i;k<=j;k++){
        int ballons = nums[i-1]*nums[k]*nums[j+1] + solve(i,k-1,nums, dp) +
solve(k+1, j, nums, dp);
        maxi = max(maxi, ballons);
    }
    return dp[i][j] = maxi;
}
 nums.insert(nums.begin(), 1);
        nums.push_back(1);
        int n = nums.size();
        vector<vector<int>>dp(n,vector<int>(n,-1));
         return solve(1, n-2, nums, dp);
```

* do not forgot about writing the base case if(I > j) continue; while converting this memoization to tabulation.
Time complexity is O(n * n * n) space is O(n *n) with auxiliary O(n) space.
Now you can easily able to convert this into tabulation

Boolean Parenthesization (GFG):

Approach: MCM(Partitioning Dp)
To solve this question we have to solve the left subpart and right subpart at each index and we have to return the logical operation. Remember that we follow the three logical operation so we have to write the all the three logical operation's truth table. And at each point we have to count the left true and right true , left false and right false and then apply the truth tables of three logical operations. Here there are three changing variables in our recursion. And we start from first as true condition. We know that first 'T' or 'F' character then operator and then again character are given in the string so we can move 2 steps in each point in the inner loop of the recursion.
Memoization algo:

```
int f(int i, int j, int isTrue, string &exp, vector<vector<vector<ll>>> &dp) {
    //Base case 1:
    if (i > j) return 0;
    //Base case 2:
    if (i == j) {
        if (isTrue == 1) return exp[i] == 'T';
        else return exp[i] == 'F';
```

```
    }

    if (dp[i][j][isTrue] != -1) return dp[i][j][isTrue];
    ll ways = 0;
    for (int ind = i + 1; ind <= j - 1; ind += 2) {
        ll lT = f(i, ind - 1, 1, exp, dp);
        ll lF = f(i, ind - 1, 0, exp, dp);
        ll rT = f(ind + 1, j, 1, exp, dp);
        ll rF = f(ind + 1, j, 0, exp, dp);

        if (exp[ind] == '&') {
            if (isTrue) ways = (ways + (lT * rT) % mod) % mod;
            else ways = (ways + (lF * rT) % mod + (lT * rF) % mod + (lF * rF) %
mod) % mod;
        }
        else if (exp[ind] == '|') {
            if (isTrue) ways = (ways + (lF * rT) % mod + (lT * rF) % mod + (lT * rT)
% mod) % mod;
            else ways = (ways + (lF * rF) % mod) % mod;
        }
        else {
            if (isTrue) ways = (ways + (lF * rT) % mod + (lT * rF) % mod) % mod;
            else ways = (ways + (lF * rF) % mod + (lT * rT) % mod) % mod;
        }
    }
    return dp[i][j][isTrue] = ways;
}
```

Time complexity is O(n * n * logn) long because in third inner loop we can goes 2 steps at each point. and space complexity is O(n * n * 2).
If we follow the same base case and 3D DP then we easily able to convert this memoization code into the Tabulation.

132. Palindrome Partitioning II

Approach: 1D DP and front partitioning
This is a similar problem to our 1D DP the only difference is we have to make the front partitioning to string. Front partitioning means start from left side and if the given substring is palindrome then try to make partition at that point and then call the recursion for remaining substring. And at the end return the minimum partitions.
Memoization algo:

```
int solve(int idx, string s, vector<int>& dp){
    if(idx >= s.size()){
        return 0;
    }
    if(dp[idx] != -1) return dp[idx];
```

```
    int mini = 1e8;
    for(int i=idx; i<s.size();i++){
        if(ispalindrome(s, idx, i)){
            int min_step = 1 + solve(i+1, s, dp);
            mini = min(mini, min_step); } }
    return dp[idx] = mini; }
```

Time complexity is : O(n ^ 2) and space complexity is O(n) + auxiliary stack space.

Tabulation algo:

```
vector<int>dp(s.size()+1, 0);
            for(int idx = s.size()-1; idx>=0; idx--){
            int mini = 1e8;
            for(int i=idx; i<s.size();i++){
                if(ispalindrome(s, idx, i)){
                    int min_step = 1 + dp[i+1];
                    mini = min(mini, min_step);
                }
            }
            dp[idx] = mini;
        }
        return dp[0]-1; }
```

If you notice we will returned dp[0]-1 over here. Because lets say we have the letters like abide when recursion call is come at e then our recursion will also try to partition ‘e’ that’s why extra 1 will added in our ans and for that we subtract the 1. This will works perfectly fine in the coding ninja. But this solution will give the Memory LIMIT Exceed in Leetcode because we call the us palindrome function at each point so we have to memoize the Ispalindrome sequence also.

```
int n = s.size();
    vector<int> dp(n + 1, 0);
    vector<vector<bool>> isPalindrome(n, vector<bool>(n, false));
    for (int i = n - 1; i >= 0; i--) {
        int mini = INT_MAX;
        for (int j = i; j < n; j++) {
            if (s[i] == s[j] && (j - i <= 2 || isPalindrome[i + 1][j - 1])) {
                isPalindrome[i][j] = true;
                int min_step = 1 + dp[j + 1];
                mini = min(mini, min_step); } }
        dp[i] = mini;
    }
    return dp[0] - 1;
```

1043. Partition Array for Maximum Sum

Approach: Front Partitioning DP

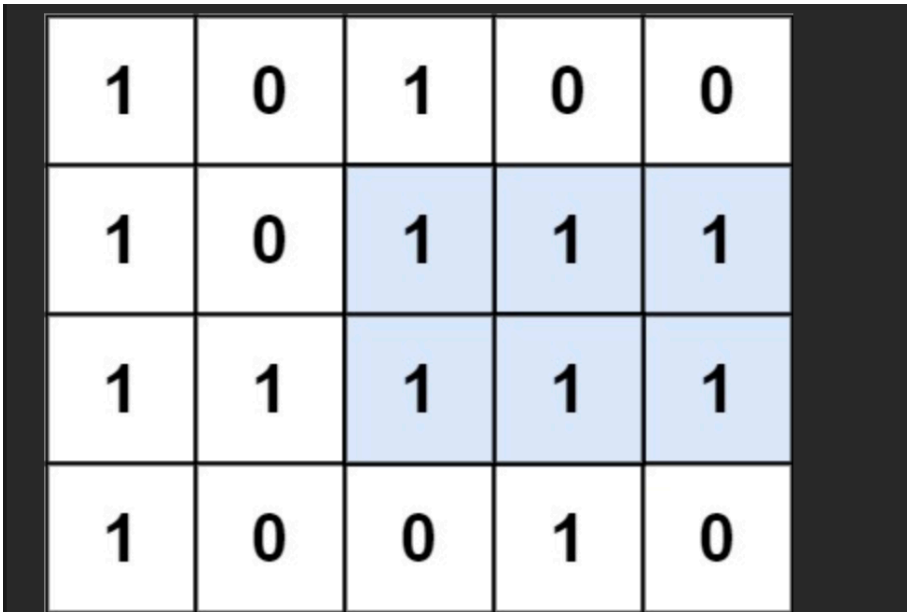
When you take any element in array please go until the k and find the maximum element on it maintain one counter also and then for that position make sum = max element till now * number of count and then remaining partition of the array. And finally return the best possible answer.
Memoization algo:

```
int solve(int idx, int k, int n, vector<int>& arr, vector<int>&dp){
    if(idx == n){
        return 0;
    }
    if(dp[idx] != -1) return dp[idx];
    int cnt = 0;
    int ans = INT_MIN;
    int maxi = INT_MIN;
    for(int i=idx; i<min(idx+k, n); i++){
        cnt++;
        maxi = max(maxi, arr[i]);
        int sum = maxi * cnt + solve(i+1, k, n, arr, dp);
        ans = max(ans, sum);
    }
    return dp[idx] = ans;
}
```

Time complexity is O(n * k) space is O(n) with O(n) auxiliary space.
And again the conversion of Memoization to Tabulation is very easy.

85. Maximal Rectangle

Approach : Maximum Rectangle in Histogram.
To solve this question the prerequisite is our previous famous question called Maximum Rectangle in Histogram. We solved that question using the left smaller and right smaller array using the stack approach. Can we relate this question to that? So ans is yes. Then How?



In this if we treat each row as our histogram then it is simple to solve this question. Till each row add that column if there is no 0 is occurred in that row. And then apply our original logic. For example in the first, second and third row if we add them then the array for histogram array is: [3, 1, 3, 2, 2] which is our maximum answer.
Algo:

```
sint largestRectangleArea(vector<int>& heights) {
        stack<int> st;
        int ans = 0, n=heights.size();
        for(int i=0;i<=n;i++){
            while(!st.empty() && (i==n || heights[st.top()] >= heights[i])){
                int height = heights[st.top()];
                st.pop();
                int width;
                if(st.empty()) width = i;
                else width = i - st.top() - 1;
                ans = max(ans, (height*width));
            }
            st.push(i); }
        return ans; }
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        int n = matrix[0].size();
        vector<int>temp(n, 0);
        int ans = 0;
        for(auto it: matrix){
            for(int i=0; i<n;i++){
                if(it[i] == '1'){
                    temp[i] += 1;
                }
                else{
                    temp[i] = 0; } }
         ans = max(ans, largestRectangleArea(temp));
        }
        return ans;
```

Time complexity is O(n * (m + n)) and space is O(n) for extra array + O(n) monotonic stack.

1277. Count Square Submatrices with All Ones

Approach: Dynamic Programming
If you try the Brute in such type of questions then definitely it can be accepted but the code of brute force method is very complex so we have to come up with such solution so our code will become very easy.
When the square is formed when our bottom right guy will be a 1 and for that bottom right guy we have to calculate the number of square it can form so

how to do this? We know for first column and first row they can able to perform only 1 square if they have 1 for sure. Now what about another guys. So for that we take upper guy upper left diagonal and left guy and then take it minimum and add 1 for our guy itself. Why we take minimum because we have to find out the square not rectangle if the question is changed and we have to find out the rectangle then definitely we take the maximum of that three guys.

Algo:

```
vector<vector<int>> dp(n, vector<int>(m,0));
        int ans = 0;
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(i == 0 || j==0){
                    dp[i][j] = matrix[i][j];
                }
                else{
                    if(matrix[i][j] == 1){
                        dp[i][j] = min(dp[i-1][j-1], min(dp[i][j-1], dp[i-1][j])) + 1;
                    }
                }
                ans += dp[i][j];
            }
        }
        return ans;
```

Time complexity is O(n * m) space is also same because we store the number of squares for each element.

152. Maximum Product Subarray

Approach: prefix and suffix multiplication

The brute method will runs in O(n^2) time to solve this question but how to solve this question in O(n). Ans is take 2 variable called prefix and suffix and start from first and last parallel. Multiply prefix with nums[I] and suffix with nums[n-i-1]. And update the maxi. In case if prefix or suffix will 0 then again make it 1.

Algo:

```
Int n = nums.size();
Int prefix=1, suffix = 1, maxi = INT_MIN;
for(int I=0;i<n;i++){
if(prefix == 0) prefix = 1;
if(suffix == 0) suffix = 1;
Prefix *= nums[I];
Suffix *= nums[n-i-1];
Maxi = max(maxi, max(suffix, prefix));
}
```

Time complexity is O(n).

Egg Dropping Puzzle (GFG):

Approach: Dynamic Programming

This is a famous puzzle which can solved by the recursion. Why recursion? Because we have to try out all the possible approaches. And that's why we have to use the recursion. And at each point we have two possibilities first is egg is break or another is egg is not break so we can able to use the pick and not pick approach of our dynamic programming. In question it is given that when k =1, k=0, and n=1 ans is k so those three are our base cases.

Memoization algo:

```
int solve(int n, int k, vector<vector<int>>&dp){
    if(k == 1 || k== 0 || n == 1){
        return k;
    }
    if(dp[n][k] != -1) return dp[n][k];
    int mini = INT_MAX;
    for(int i=1; i<=k; i++){
        int broke = solve(n-1, i-1, dp);
        int notbroke = solve(n, k-i, dp);
        mini = min(mini, max(broke, notbroke));
    }
    return dp[n][k] = mini + 1;
}
```

Time complexity is O(n * k * k) space is O(n * k) with auxiliary stack space.

Tabulation algo:

```
for(int i=0; i<=n;i++){
            dp[i][0] = 0;
            dp[i][1] = 1;
        }
        for(int i=0;i<=k;i++){
            dp[1][i] = i;
        }
        for(int idx = 2; idx<= n;idx++){
            for(int fl=2; fl<=k ;fl++){
                int mini = INT_MAX;
                for(int i=1; i<=fl; i++){
                    int broke = dp[idx-1][i-1];
                    int notbroke = dp[idx] [fl-i];
                    mini = min(mini, max(broke, notbroke));
                }
                dp[idx][fl] = mini+1; } }
        return dp[n][k];
```

139. Word Break

Approach: front partitioning Dp

The simple approach is take the entire dictionary into set so it will don't contains the 2 or more occurrences of the same word. Now build a recursion which can start from 0 to string size. In that iterate with inner loop from idx to String size and then take that substring if the substring is present in the dictionary then return true and call the recursion for I+1 again otherwise at the end of the day return false.
Memoization algo:

```
bool solve(string s, int pos, set<string>& st,vector<int>& dp){
        if(pos == s.size()) return true;
        if(dp[pos] != -1) return dp[pos];

        for(int i=pos;i<s.size();i++){
            if(st.count(s.substr(pos,i-pos+1))){
                if(solve(s, i+1, st,dp)) {
                    return dp[pos] = true;
                    }
                }
        }
        return dp[pos] = false;
        }
```

Time complexity is almost O(n^2) and space complexity is O(n) with auxillary stack space if you want to reduce that stack space also then convert this memoization into tabulation which Is very easy.

208. Implement Trie (Prefix Tree)

Algo: Link
Time complexity is O(n) for insertion, O(n) for Search a Word and O(n) for search a prefix word.

Implement Trie - 2 (Coding Ninja):

All the other stuff of the original trie is same but the only changes is in that we used the bool flag for identify the end of the word. But here we take the int endpoint to count how many words are there which can reach the ending of word and we also take the another count called preffic count which gives the prefix count at each stage. And all the other thing are same.
Link
Time complexity is O(n) for each operation.

Complete String (Coding Ninja):

Approach: Trie Data structure
If you know trie very well then this is a very easy question for you. Why let's understand step by step.

Step->1: put all the characters of array of string in the trie. And remember at the end of each word from array you have to make the flag of a trie as true.
Step2-> make one function which can gives whether the given word is present or not in the trie.
Step3-> iterate our original array of string again and check if that word is present or not in the trie. And parallely check for the lexicographical order also. And return the target possible word which is present in the trie.
*Note that you have to follow the standard procedure of creating the trie and insertion on it with class and it's object and use that object in your function it is a plus point for you.
Algo:
Link
Time complexity is O(n * avg length of words in array) for insertion + O(n * avg length of words in array) for again iterating and finding our longest string in trie. And it is a ideal to say about space because each node will contains the 26 child node on it in worst case.

Count Distinct Substrings (Coding Ninja):

Approach: Trie
The very first approach is come in our mind is set.because our CP mind will tell us whenever you see the count distinct then use set and off course it will works. So what is the procedure.
Step1 : iterate from 0 to n-1
Step 2: iterate from I to n-1
Step 3: Take each substring and put it into set.
Step4: since we know from question we have to add 1 for empty string also, so return st.size() +1.
This will take O(n^2 * log n) time and extra space for set. Why long because set will take long time for insertion.
Now this approach will accepted for sure. But our motive is to learn the trie for this question.
So what is intuition for that:
Step1-> make a trie with only list[26] size, here we do not required flag element because we have to return only distinct count. And make the int insert function which can takes the word and insert it into the trie and at the end it will returns. How many characters it will entered in trie.
Step 2 -> Take all the substrings of the string and gives to the insert function.
Step 3-> add all the counts which will returned the insert function.
Step 4 -> return the count + 1.
Algo:
Link
Time complexity is O(n^2) and ideal space for Trie.

Power Set (GFG):

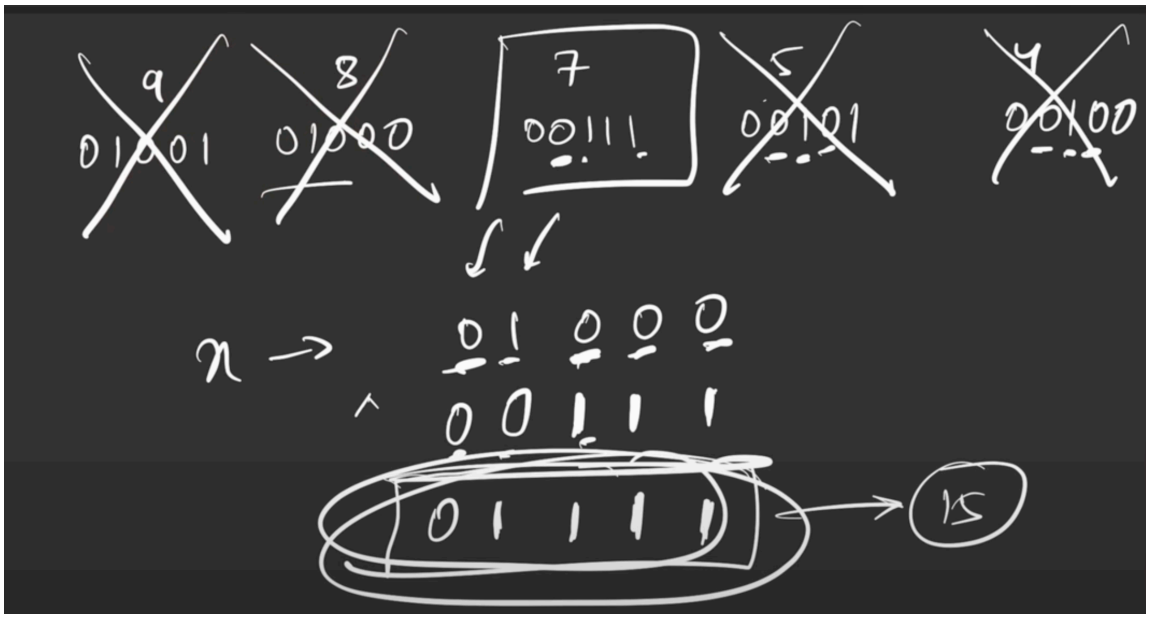
Approach: power set;
Apply the power set approach. Since the answer is accepted in lexicographically sorted order so please sort the resultant array.

```
vector<string> AllPossibleStrings(string s){
        vector<string>ans;
        int n = s.size();
        for(int i=0; i<(1<<n); i++){
            string str = "";
            for(int j=0; j<n;j++){
                if(i & (1<<j)){
                    str += s[j]; } }
            if(!str.empty()) ans.push_back(str);
        }
        sort(ans.begin(), ans.end());
        return ans;
```

Time complexity is O(n^2) to generate all possible combinations.

421. Maximum XOR of Two Numbers in an Array

Approach: Trie
This is one of the most important question to understand all the stuff like bit manipulation , XOR and Trie with XOR operation. Now the important question is hoe to apply those stuffs in this question?
We know the truth table of Xor is: 1-0 = 1, 0-1= 1, 0-0 = 0, 1 -1 = 0. Means even no of 1's then 0 and odd no of 1's then 1. Let's for example the number 9 has the binary numbers like 0 1 0 0 1. How to maximise the xor of 9 so for that we start from left side if we found the 0 then we are looking 1 and if we found the 1 then we are looking for 0 because only opposite bits are gives the maximum xor according to the xor truth table.
Refer the following image for that:



So, what if we put all the numbers of an array into the trie in the form off it's binary and then solve this question? Ans is yes, you can do it. It will makes your task become so easier.
You can use the 32 bits integer to solve this particular question so in future it will not creates the trouble.
Algo:
Step1-> build a trie which has two function one for insert the number in the form of it's binary bits. And another function which takes the integer number and return the maximum possible xor from trie.
Step2- > now put all the elements of a array in trie.
Step3-> iterate again entire array and give all the numbers to the getmax function of trie it will gives you the maximum possible xor.
Step4-> return the maximum answer.
Link:

Time complexity is O(n * 32) + O(n * 32) for iterate two times array and trie. Again we do not able to tells the exact space complexity in the trie.

This all logic will works for single array what if we have the task to solve for two array and task to find the maximum xor for both the array. Then the slight changes is creates a trie for first array and again instead of iterate the first array and call the getmax function of trie you can iterate the second array, Which will gives you the correct answer.

1707. Maximum XOR With an Element From Array

Approach: Offline Queries and Trie
If we think this problem as previous then we are wrong. Because in that we first putted all the element and then finding the maximum xor. But here there is a different case. Because we have to find the max xor for some queries. So how to approach this question.
Let's think different what if our trie contains the elements <= our query element then our job to be done. So for that we follow the given steps.
Step1-> sort the nums array also create one ans vector for our final answers.
Step2-> create a vector pair of pair which contains the query mi value, xi value and index of that query.
Step->3 now sort the offline query vector according to it's mi value.
Step4-> now again iterate that vector pair of pair and take each query and while the elements of nums array will lesser or equal to the mi value insert those elements in trie and then find the max xor and put it into the ans vector.
Link:

Time complexity is O(qlength log qlength) + O(qlength * 32 + n * 32) where q length is length of the query array. Now some of you are thinking that why O(qlength * 32 + n * 32) because all the elements of array will inserted only once in the trie.

* * *