Coding School.

Selected Problems to Crack Interview.

Array: -

Problem 1 - Number Of Triangles.

Write a program to count the number of possible triangles that can be formed with three elements from a given unsorted array of positive integers.

Solution -

How to approach this problem🤔🤔??

Step 1 - In the problem he is asking about the triangle. So first think about what is the condition for a triangle.
Suppose a,b,c are the sides of a triangle. Then these conditions must be followed…(Sum of two any two sides must be greater than the third side).
a+b>c
b+c>a
a+c>b.

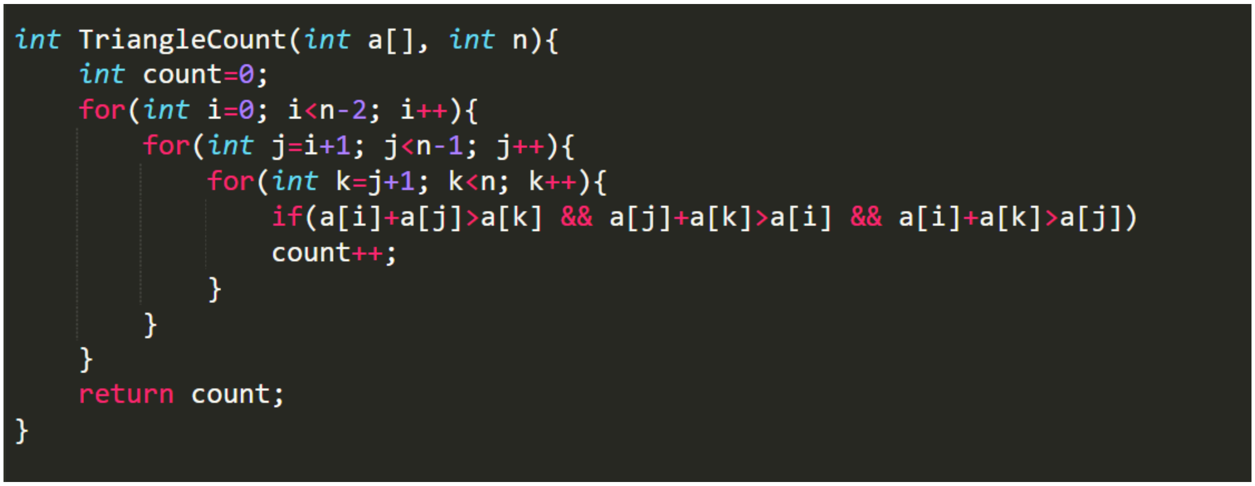
Step 2 - Think about the brute force approach.
Here we need three numbers. So, we consider every combination of three numbers in the given array and check if they satisfy the triangle inequality. And with this we also keep a count of the number of combinations that satisfy this condition.
Now, how to write code for this approach??

We have to keep track of three numbers, so we need three loops.

- Outermost loop runs from i=0 to n-3 to select the first side of the triangle, a[i].
- The middle loop runs from j=i+1 to n-2 to select the second side, a[j].
- The innermost loop runs from k=j+1 to n-1 to select the third size, a[k].

With this we have to check if the three sides form a valid triangle. If they do, then we increase the count by 1.
When all iterations are completed we will get our answer.

So, let's Code - 🧑‍💻🧑‍💻



```
int TriangleCount(int a[], int n){
    int count=0;
    for(int i=0; i<n-2; i++){
        for(int j=i+1; j<n-1; j++){
            for(int k=j+1; k<n; k++){
                if(a[i]+a[j]>a[k] && a[j]+a[k]>a[i] && a[i]+a[k]>a[j])
                count++;
            }
        }
    }
    return count;
}
```

Time Complexity Analysis -

The outer loop iterates (n-2) times. - O(n).
Middle for loop iterates (n-i-2) times. - O(n).
Inner for loop iterates (n-j-1) times. - O(n).
Overall complexity - O(n)*O(n)*O(n) = O(n^3).

From another perspective, we are exploring all possible combinations of all three sides of a triangle. So the total number of possible combinations = The number of ways to pick three elements among n elements = nC3 = n*(n-1)*(n-2)/3. So, this is O(n^3).

Space Complexity - We are using constant extra space, so space complexity is O(1).

Step 3 - Try to reduce the complexity. Means optimize your code.
Now, check is it possible to preprocess the array and efficiently identify all possible Triangles?
One idea is to sort the given array. Why sort🤔🤔??
Because, if we consider a triplet (a, b, c) such that a<=b<=c, we only need to check that a+b>c in order to determine if the triangle formed by them is valid.

The idea is simple: if c>=a and c>=b, then adding a number to c will always produce a sum that is greater than either x or y. Therefore the inequalities c+a>b and c+b>a are satisfied implicitly by the property a<=b<=c.

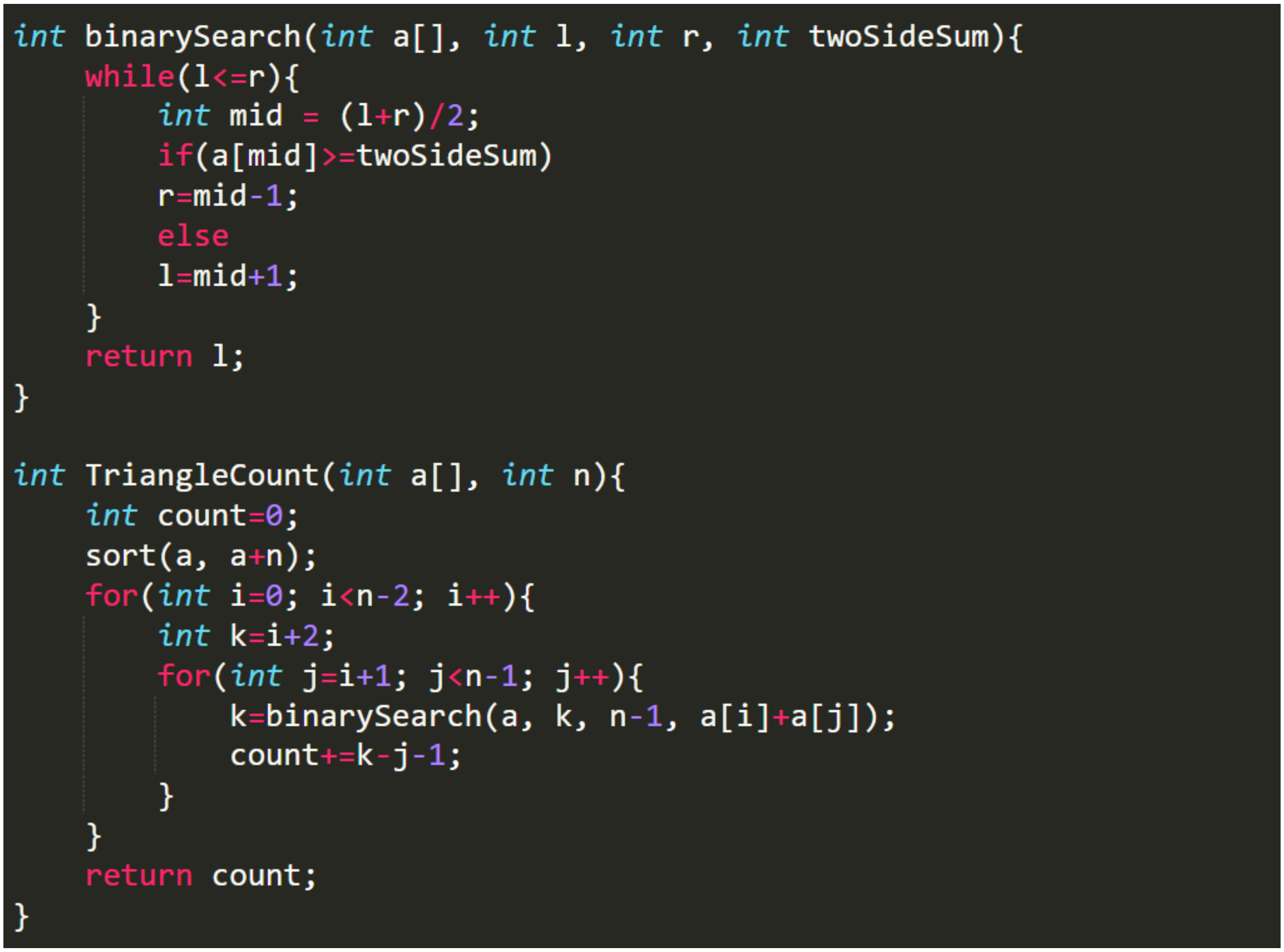
To determine the number of elements that satisfy the inequality a[k]<a[i] + a[j], we first sort the array a[]. We then consider each pair of elements (a[i], a[j]) such that j>i.
Since the array is sorted, as we traverse towards the right to select the index k, the value of a[k] can only increase or remain the same. This means that there is a right limit on the

value of k such that the elements satisfy a[k] < a[i] + a[j]. Any elements beyond this value of k will not satisfy the inequality.

How to find the right limit value of k???

We can use binary search, since the array is sorted. Once we find the right limit value of k, we can conclude that all the elements in the range (j+1, k-1) (inclusive) satisfy the required inequality.
So, the count of elements satisfying the inequality will be
(k-1) - (j+1) + 1 = k-j-1.

So let’s Code! 🧑‍💻🧑‍💻



```
int binarySearch(int a[], int l, int r, int twoSideSum){
    while(l<=r){
        int mid = (l+r)/2;
        if(a[mid]>=twoSideSum)
        r=mid-1;
        else
        l=mid+1;
    }
    return l;
}

int TriangleCount(int a[], int n){
    int count=0;
    sort(a, a+n);
    for(int i=0; i<n-2; i++){
        int k=i+2;
        for(int j=i+1; j<n-1; j++){
            k=binarySearch(a, k, n-1, a[i]+a[j]);
            count+=k-j-1;
        }
    }
    return count;
}
```

Time Complexity Analysis -
Outer for loop - O(n)
Inner for loop - O(n)
Binary Search - O(logn).
Overall Complexity - O(n^2*logn).

Further analysis, How can we reduce the number of operations in Binary Search???
When we find the right limit index (k) for a specific pair (i, j), we don't need to start searching for the right limit (k) again from the index j+2 when we choose a higher value of j for the same value of i.
Instead, we can simply continue directly from where we left off for the last chosen value of j, at index k.
This can help us to reduce the range of our binary search for k to shorter values, which in turn can improve the performance of our algorithm.

😊Can we further optimize our algorithm🤔??
Let's Optimize our algorithm……..
In the above approach, we are applying binary search O(n^2) times to find the right limit of the index k for every pair of indices (i, j).
Can we think of improving this further??
Here is a better idea!!

As the array is sorted, we can find the right limit of k by traversing linearly from k=i+2 and stopping at the first value of k that does not satisfy the inequality a[i] + a[j] > a[k].

The count of elements a[k] satisfying this inequality for a particular pair (i, j) is equal to k-j-1.

Algo -

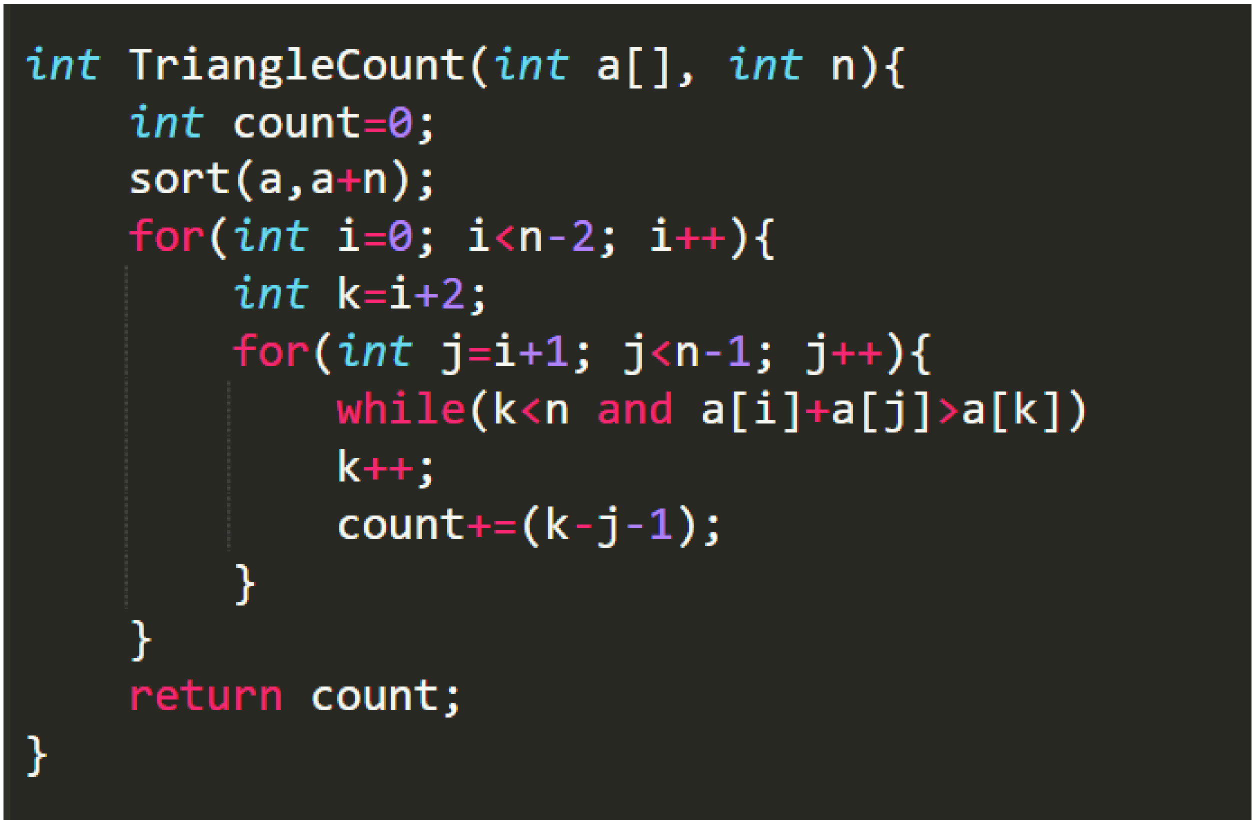
- Outer loop from 0 to n-3 to pick the first side a[i] of the triangle.
- Inner loop from i+1 to n-2 to pick the second side a[j] of the triangle.

What about the third side................???

- Inside the inner loop we define a variable k to track the third side a[k] of the triangle. We initialize it to i+2.
- We run a while loop that continues as long as k <n and a[i]+a[j] > a[k]. This loop helps us find the count of the third sides which satisfy the conditions of a triangle.
- In other words, we find the farthest index k of the third side whose length is less than the sum of the other two sides.
- After the end of the while loop, we increase the value of count by k-j-1.

Finally return Count. Yeah we got it finally! 🤗

So, let's Code!🧑‍💻🧑‍💻



```
int TriangleCount(int a[], int n){
    int count=0;
    sort(a,a+n);
    for(int i=0; i<n-2; i++){
        int k=i+2;
        for(int j=i+1; j<n-1; j++){
            while(k<n and a[i]+a[j]>a[k])
            k++;
            count+=(k-j-1);
        }
    }
    return count;
}
```

Time Complexity Analysis -

The code contains three nested loops, which may lead someone to believe that its complexity is O(n^3).

Really, is it O(n^3)......?? Think about it!

No boss, it's not O(n^3).

Let's examine deeply, you will realize that the complexity is actually better.

It can be observed that k is initialized only once in the outermost loop. The innermost loop executes at most O(n) time for every iteration of the outermost loop, because k starts from i+2 and goes up to n for all values of j.

So overall complexity of inner for loop and while loop will be O(n).
Finally, Outer for loop complexity = O(n).
Inner for loop and while loop complexity = O(n).
Therefore, the time complexity is O(n)*O(n) = O(n^2).

One more solution is also available for this problem, using a two pointer algorithm.
Link - Solution using Two pointer algorithm

Note: - We will discuss Two pointers algorithm in next pages.

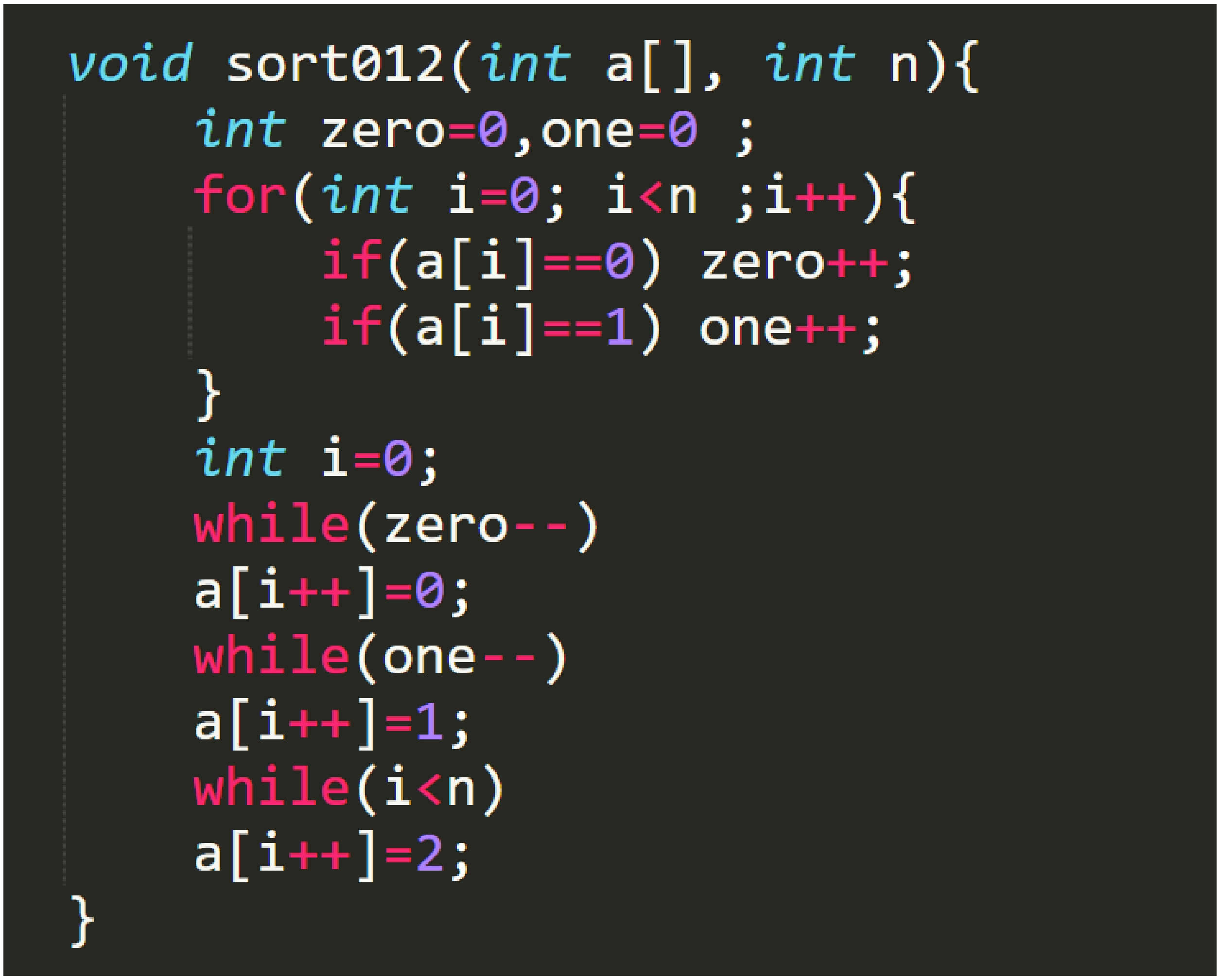
Problem 2: - Sort an array of 0s, 1s, and 2s.

Solution: -
Brute force method: -
Simply sort the array using the Sorting algorithms.
Time Complexity - Merge Sort O(nlogn).

Can we optimize this further……….????????
Key point - Here we have only three numbers: 0, 1, and 2.
So this is an advantage for us.
What will we do??

- Simply count the number of 0s, 1s, and 2s in the array.
- Then put all the 0s at the starting of the array, after that put all the ones, and then put all 2s.

So let’s Code!🧑‍💻🧑‍💻🧑‍💻



```
void sort012(int a[], int n){
    int zero=0,one=0 ;
    for(int i=0; i<n ;i++){
        if(a[i]==0) zero++;
        if(a[i]==1) one++;
    }
    int i=0;
    while(zero--)
    a[i++]=0;
    while(one--)
    a[i++]=1;
    while(i<n)
    a[i++]=2;
}
```

Time Complexity Analysis -
For loop - O(n).
While loop - from i=0 to n-1, so complexity - O(n)
Overall complexity = O(n) + O(n) = O(n).
Boom! Complexity Reduced.

Space complexity - O(1).

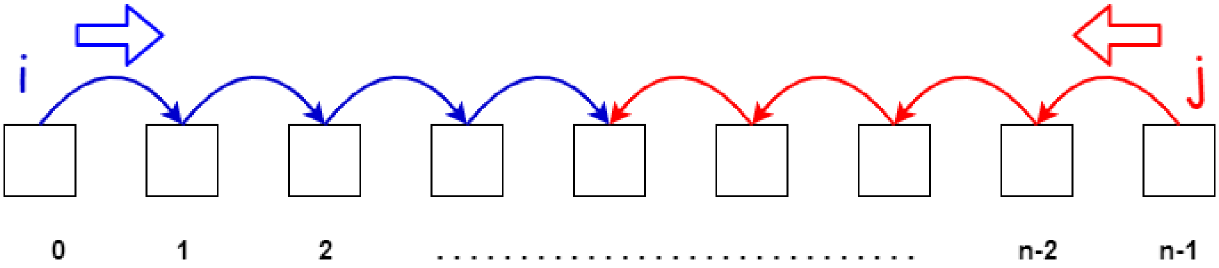
....Two Pointers....

When we have to use a Two pointers algorithm.....??

Two pointer algorithms are typically used in situations where you have a collection of items, such as an array or a list, and you need to find a specific subset of those items that meet certain conditions.
For example, two pointer algorithms are often used to find pairs of elements in an array that sum to a given value, or to find a sub-array with a given sum.

They can also be used to find the longest increasing subsequence in an array, or to find the smallest window in a string that contains all characters of another string.

NOTE: - Two pointer algorithms are efficient because they typically only iterate through the collection of items once, and they can be useful in solving both easy and complex problems.



Here we will solve some problems on Two pointer algorithm.

Problem 1: - Remove Duplicates.

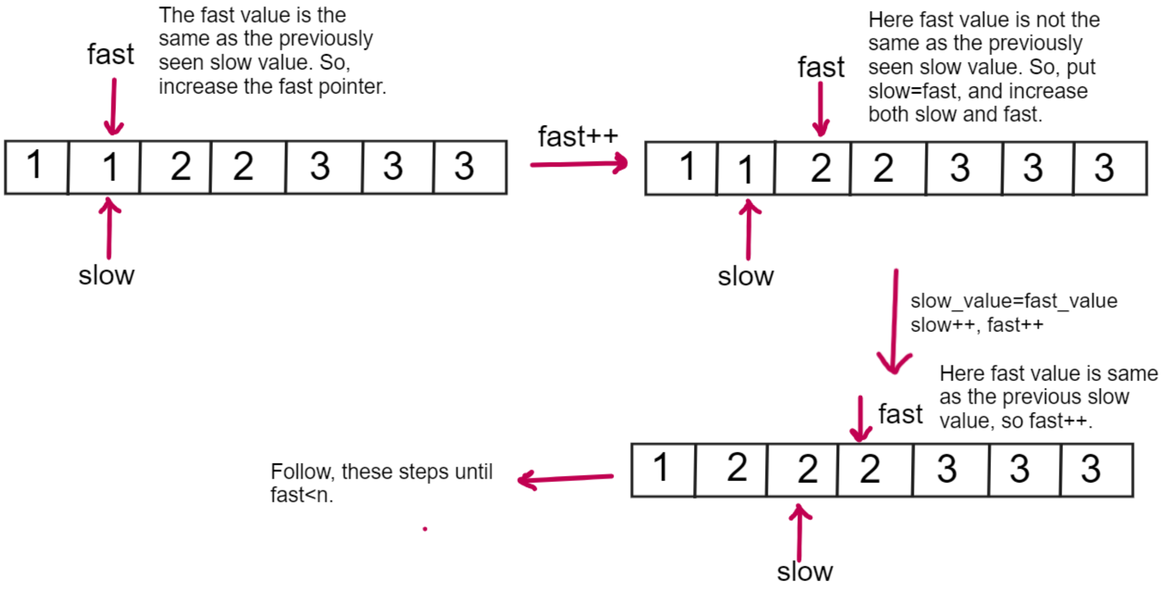
Given an integer array, sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once.
The relative order of the elements should be kept the same.

Ex. Input - [0, 0, 1, 1, 1, 2, 2, 3, 3, 3]
Expected Output - [0, 1, 2, 3, _, _, _, _, _, _].

Solution -
Step 1 -
First define the Goal -
The goal here is to remove all duplicates from the array, then return the new end index of the array.

Step 2 -
Now, how to do this…???
To do this, we iterate over the array with two pointers.
1st Pointer - Iterates sequentially.(Slow Pointer).
2nd Pointer - Skips over duplicates.(Fast Pointer).

Step 3 -
Just copy the current value to the slow pointer if the current value doesn’t equal the previous value seen.



The fast value is the same as the previously seen slow value. So, increase the fast pointer.

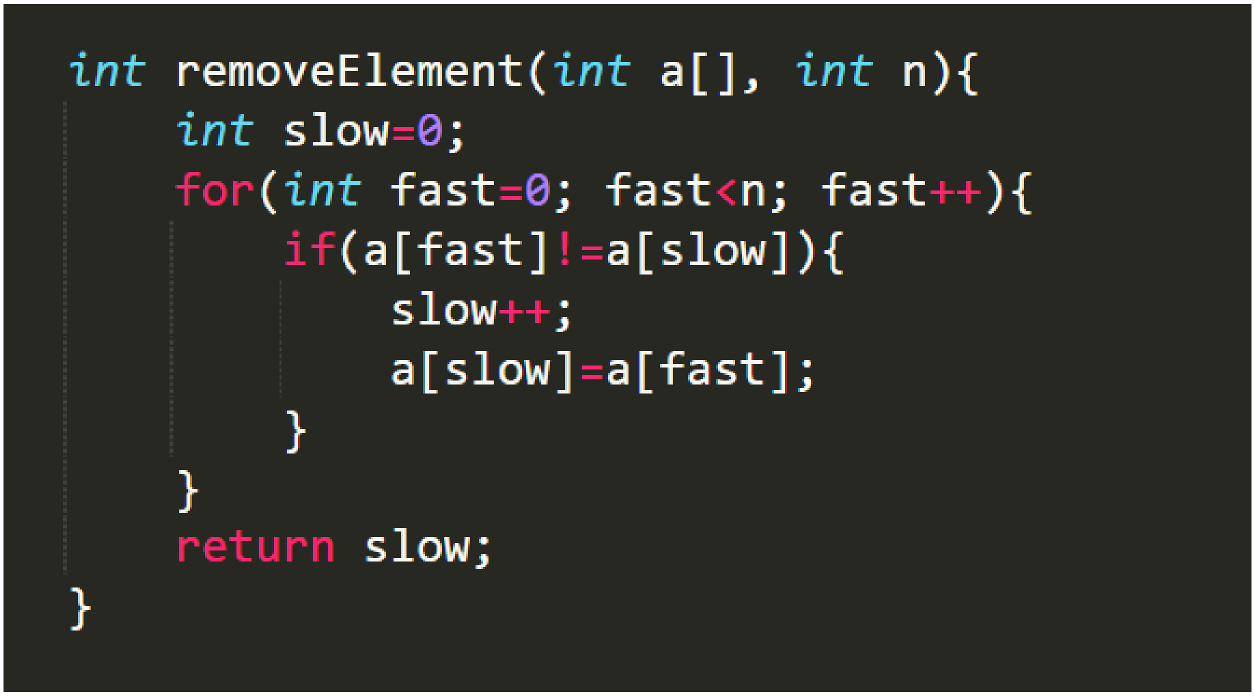
Here fast value is not the same as the previously seen slow value. So, put slow=fast, and increase both slow and fast.

slow_value=fast_value
slow++, fast++

Here fast value is same as the previous slow value, so fast++.

Follow, these steps until fast<n.

So, let's Code!!🧑‍💻🧑‍💻🧑‍💻🧑‍💻



```
int removeElement(int a[], int n){
    int slow=0;
    for(int fast=0; fast<n; fast++){
        if(a[fast]!=a[slow]){
            slow++;
            a[slow]=a[fast];
        }
    }
    return slow;
}
```

Time Complexity Analysis - As we are running only one for loop, so complexity will be - O(n).
Space complexity - O(1).

Read ……📖📖

- Two Pointer Technique

Problems on 2 Pointer algorithm -

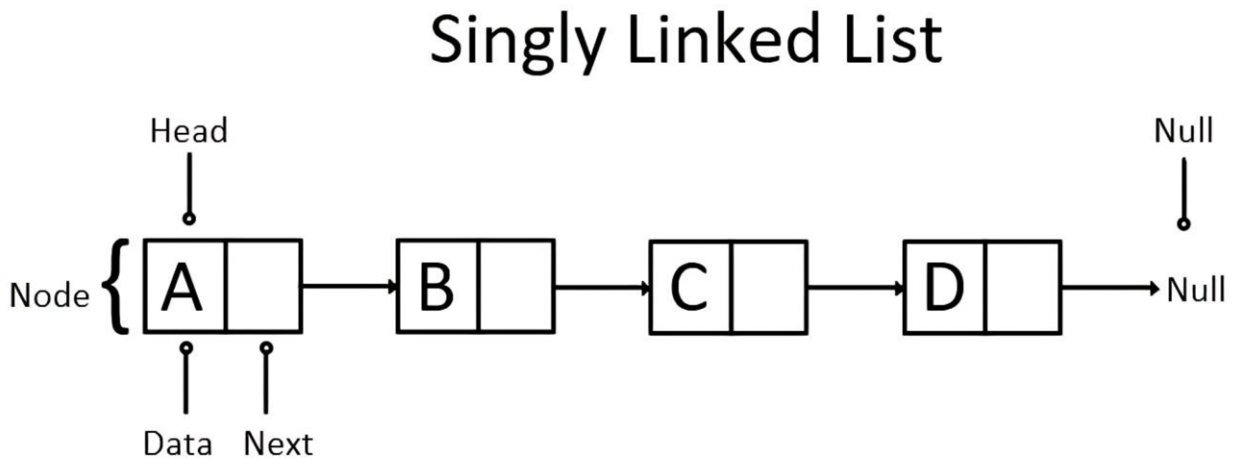
- Middle of the linked list
- Reverse String
- Sort array by parity II
- Valid Palindrome
- Container with most water

…Linked List…

Introduction: -

Linked list is a data structure which stores objects in nodes in a snake-like structure. Instead of an array (where we have a simple list to store something) we have nodes in the linked list. It’s the same thing though, you can store whatever you want - objects, numbers, strings, etc.

The only difference is in the way it’s represented. It’s like a snake: with a head and tail, and you can only access one thing at a time - giving its own advantages and disadvantages. So if you want to access the 5th thing, you can’t do linked_list[5], instead -> you would have to iterate over the list from the beginning and then stop when the number hits 5.



Problem 1: - Find the Middle node of the linked list.

Solution - Before going forward first pick a notebook and a pen, and then draw a linked list. And try it yourself.

Ok, so if you tried and you solved it, then “Great” now you are master in both “Two pointer” and “linked list”.

Wait wait, you didn’t solve this problem using two pointers…🤔🤔??

If you solved it using two pointers, then you have learned how to mix two concepts to solve any problem.

Most of the people try to solve this problem, like - First count the number of nodes in the linked list, and then again start from the head node, and then move to the next node (totalNodes/2) times, and you get the middle node.
This solution is also correct.

But, you are following “Coding School”, so think smartly.
Ok let’s start using our amazing brain.

Approach -
We take two-pointers to traverse the linked-list, where one pointer is moving with double the speed of another pointer.

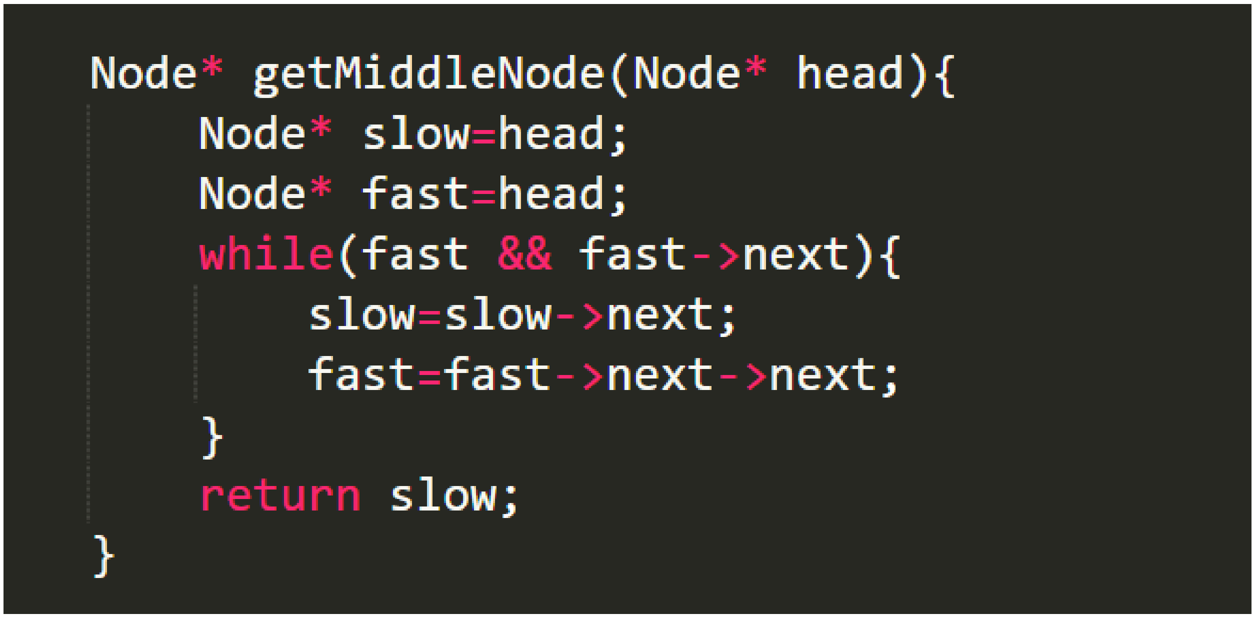
So when the fast pointer reaches the end of the linked list then the slow pointer must be present at the middle node.

The idea is very straightforward where we can get the middle node in a single scan of the linked list.

Algorithm -

- Initialize slow and fast pointers with the head node.
- Now run a loop until fast or fast->next becomes NULL.
- At each iteration of the loop, we move the slow pointer by 1 and the fast pointer by 2 steps forward.
- By the end of the loop, the slow pointer will point to the middle node and we return it.

So let’s Code!!🧑‍💻🧑‍💻🧑‍💻



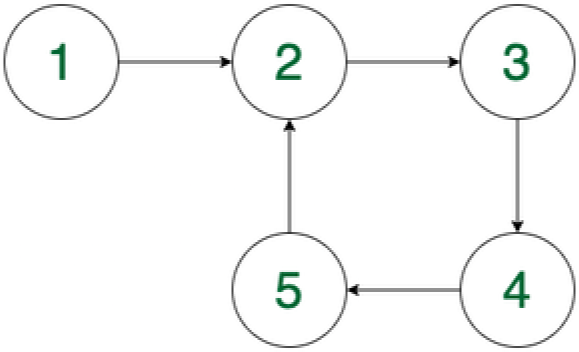
```
Node* getMiddleNode(Node* head){
    Node* slow=head;
    Node* fast=head;
    while(fast && fast->next){
        slow=slow->next;
        fast=fast->next->next;
    }
    return slow;
}
```

Time Complexity Analysis - As we traversed the linked list only once, so complexity - O(n).

Space complexity - We are not using any extra space, so space complexity will be - O(1).

Problem 2 - Detect loop in linked list. (Floyd's cycle detection algorithm).

Key Point - This is an excellent linked list problem to learn problem-solving using fast and slow pointers.



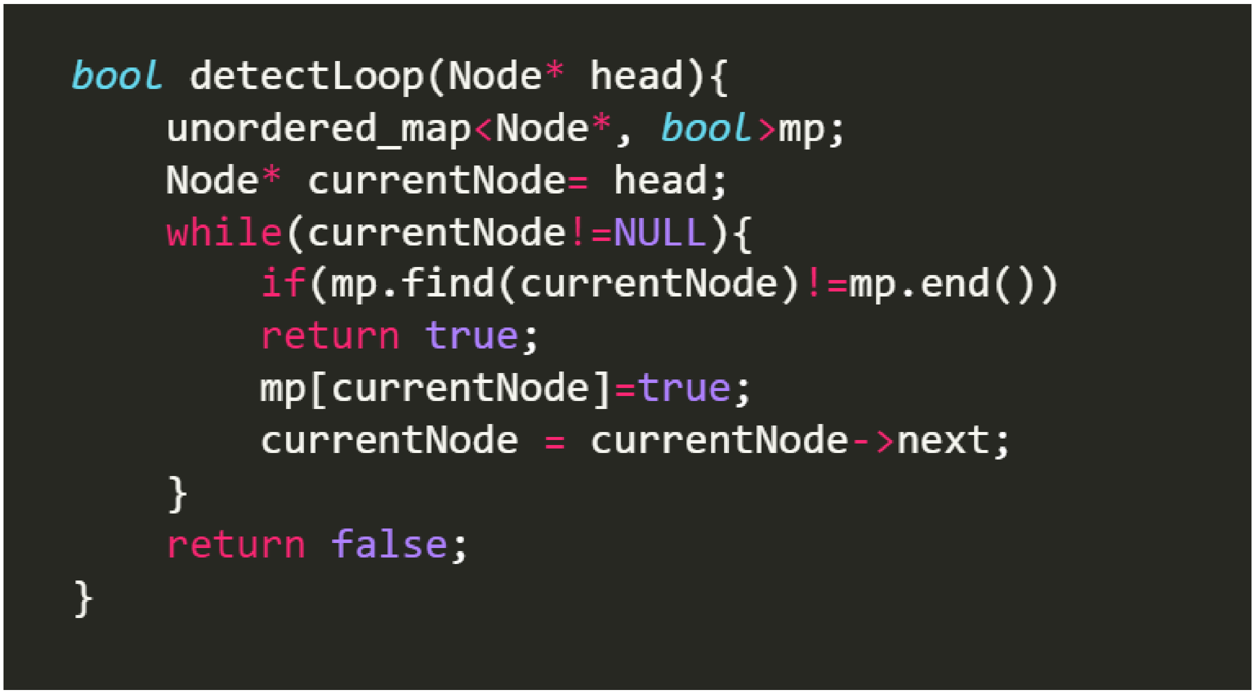
How to approach this problem…???

Step 1 - First understand what happens when a linked list is circular.

- You will not be able to find the end node.
- One of the nodes will be next to the two different nodes. In the above example node 2 is next to node 1 and node 5.

So, now using these properties try to build the algorithm.
As one node is repeating twice, in case of loop. So, we can use hashmap here, to check if there is any repeating address or not.

So let's Code!!🧑‍💻🧑‍💻🧑‍💻



```
bool detectLoop(Node* head){
    unordered_map<Node*, bool>mp;
    Node* currentNode= head;
    while(currentNode!=NULL){
        if(mp.find(currentNode)!=mp.end())
        return true;
        mp[currentNode]=true;
        currentNode = currentNode->next;
    }
    return false;
}
```

Time Complexity Analysis - As we are traversing the list only once. So, complexity - O(n).

Space Complexity - We are using a hash map, so space will be - O(n).

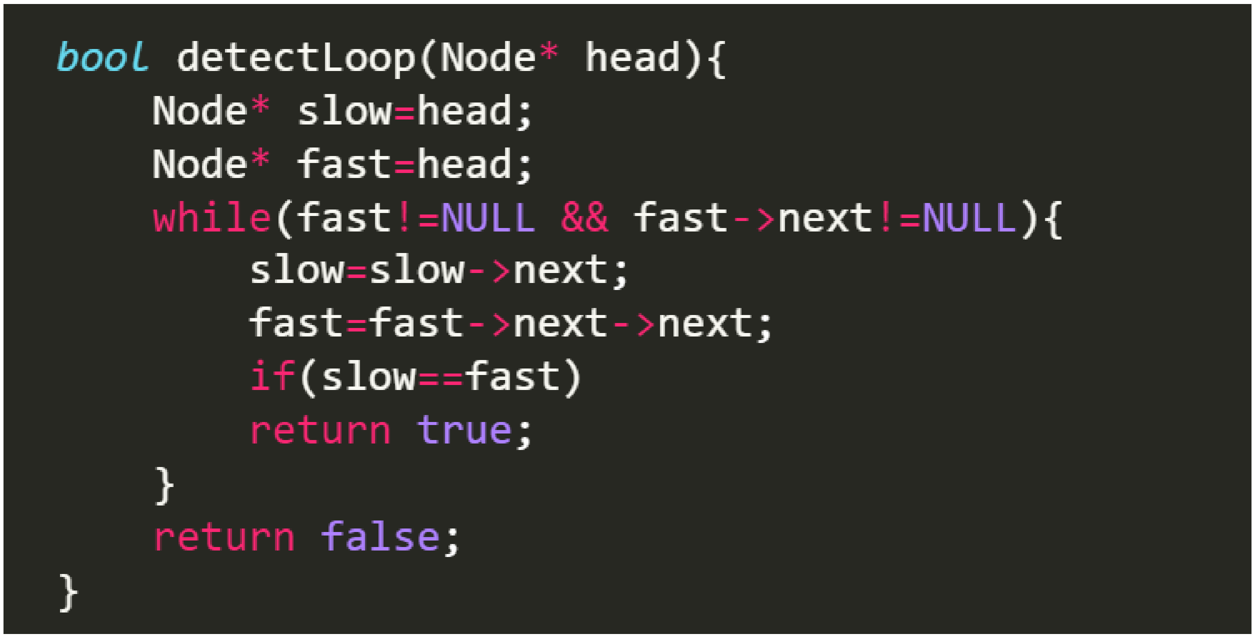
Can we optimize our space complexity……..????
Using Two pointers we can optimize our space complexity.

For this we have a famous algorithm “Floyd’s cycle finding algorithm” or “The tortoise and hare algorithm”.
This algorithm uses two pointers -

- Slow pointer - Moves one step.
- Fast pointer - Moves two steps.

If the two pointers ever meet, it means there is a cycle in the linked list.

So let’s Code!!🧑‍💻🧑‍💻🧑‍💻



```
bool detectLoop(Node* head){
    Node* slow=head;
    Node* fast=head;
    while(fast!=NULL && fast->next!=NULL){
        slow=slow->next;
        fast=fast->next->next;
        if(slow==fast)
        return true;
    }
    return false;
}
```

Time Complexity Analysis - We are traversing linked-list only once, so complexity - O(n).

Space Complexity - No extra space is used, so space complexity - O(1).

Problem 3 - Delete a node without a head Pointer.

You cannot iterate to reach that node, you just have the reference to that particular node itself.

There’s a catch here -> we can’t delete the node itself, but change the value reference. So you can change the next node’s value to the next one and delete the next one. Like this:

```
node->data = node->next->data;
node->next = node->next->next;
```

Important Tip - If you want to solve any problem of linked list, first visualize what’s happening, and then your problem will automatically be solved.

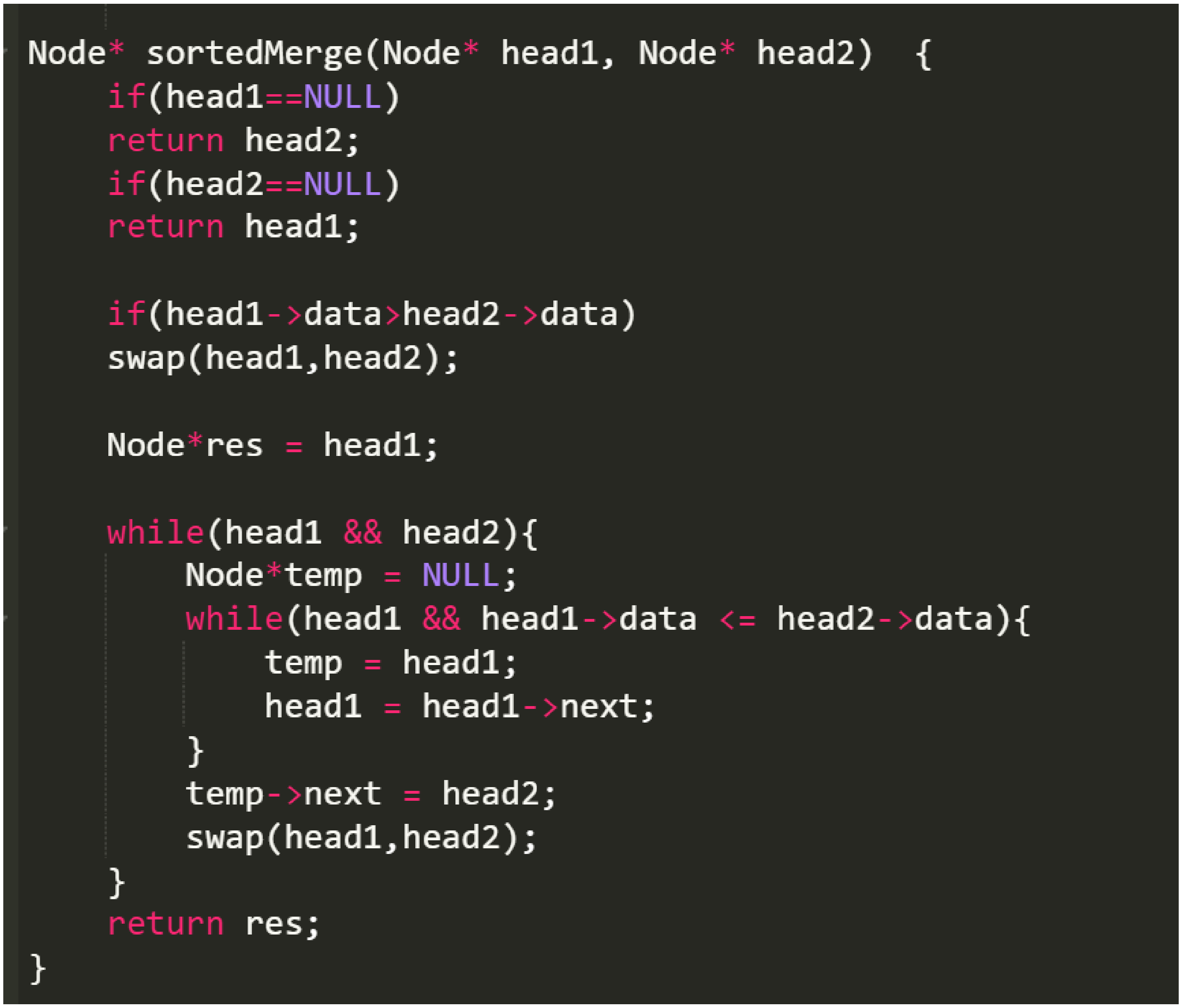
Problem 4 - Merge Two sorted Linked Lists.

This problem is basically similar to the Merge sort.
You just have to implement merge sort on the linked list.

It is given that lists are sorted, so we just start traversing the lists and compare node values of these lists.

How to merge…??
Compare the current node value of the linked list, add the minimum node to the new list. And move to the next node.

So let’s code!!🧑‍💻🧑‍💻🧑‍💻🧑



```
Node* sortedMerge(Node* head1, Node* head2)  {
    if(head1==NULL)
    return head2;
    if(head2==NULL)
    return head1;

    if(head1->data>head2->data)
    swap(head1,head2);

    Node*res = head1;

    while(head1 && head2){
        Node*temp = NULL;
        while(head1 && head1->data <= head2->data){
            temp = head1;
            head1 = head1->next;
        }
        temp->next = head2;
        swap(head1,head2);
    }
    return res;
}
```

Time Complexity Analysis - We are traversing both lists only once - O(n+m).

Space Complexity - We are not using any extra space, so space complexity is - O(n).

More Problems for practice -

- Remove the Nth node from the End of the list.
- Intersection of two linked lists.
- Rotate list.

...Sliding Window...

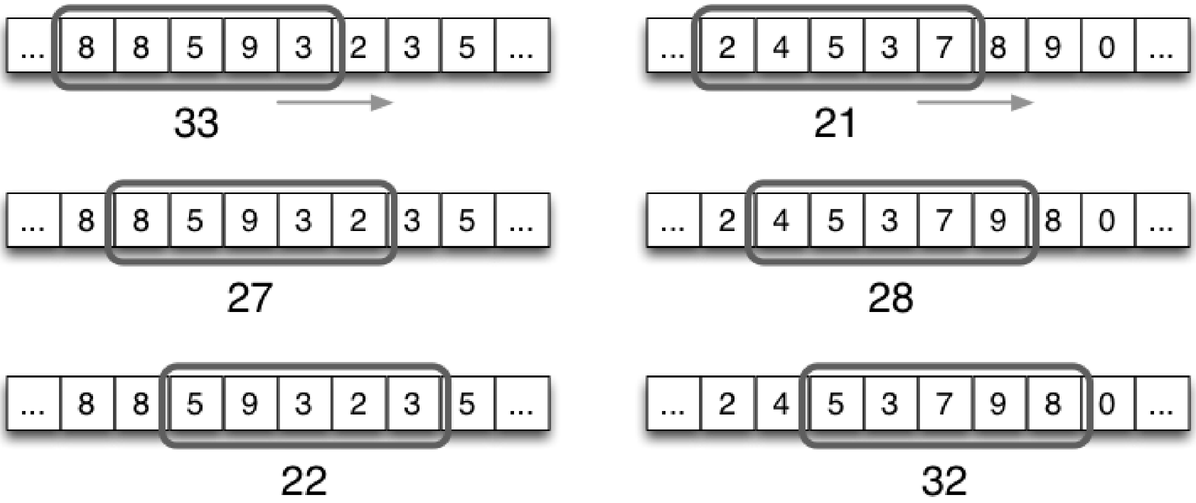
Intro -

Sliding window algorithms are very useful when you have questions regarding sub-strings and sub-sequences for arrays and strings.

Think of it as a window which slides to the left or right as we iterate through the string/array.

Key point - We already studied, 2 pointer algorithm. So, the sliding window also uses a two pointer algorithm.

Where first pointer is the start of window, and second pointer is end of the window.



Problem 1: - Max sum for consecutive k elements.

You are given an array of n integers, you have to find the maximum sum of k consecutive integers.

Solution -

How will you approach....???

What comes to your mind when you read this problem statement.

I think most of the people automatically think about the sliding window algorithm, it may be that they are not aware of this algorithm. But actually you are thinking about sliding windows only.

But how to implement this....???

Let's see.......!! 😊😊First Try it yourself. Try to visualize this one. You will get to know how to implement it.

Ok, so let's come to the point - (implementation).

We just have to pick subarrays with the length k, and find its sum, and compare it with the maximum sum. That's it.

Don't understand.....😒😒...???

OK ok, let's understand deeply, how we will implement algorithms for this problem.

How a sliding window would work here: -

- We start with a window of 'k' from the left.
- We plan to move it to the right until the very end.
- We remove the leftmost element (from the window) and add the right one as we move to the left.
- We store the sum for every window and then return the max at the very end.

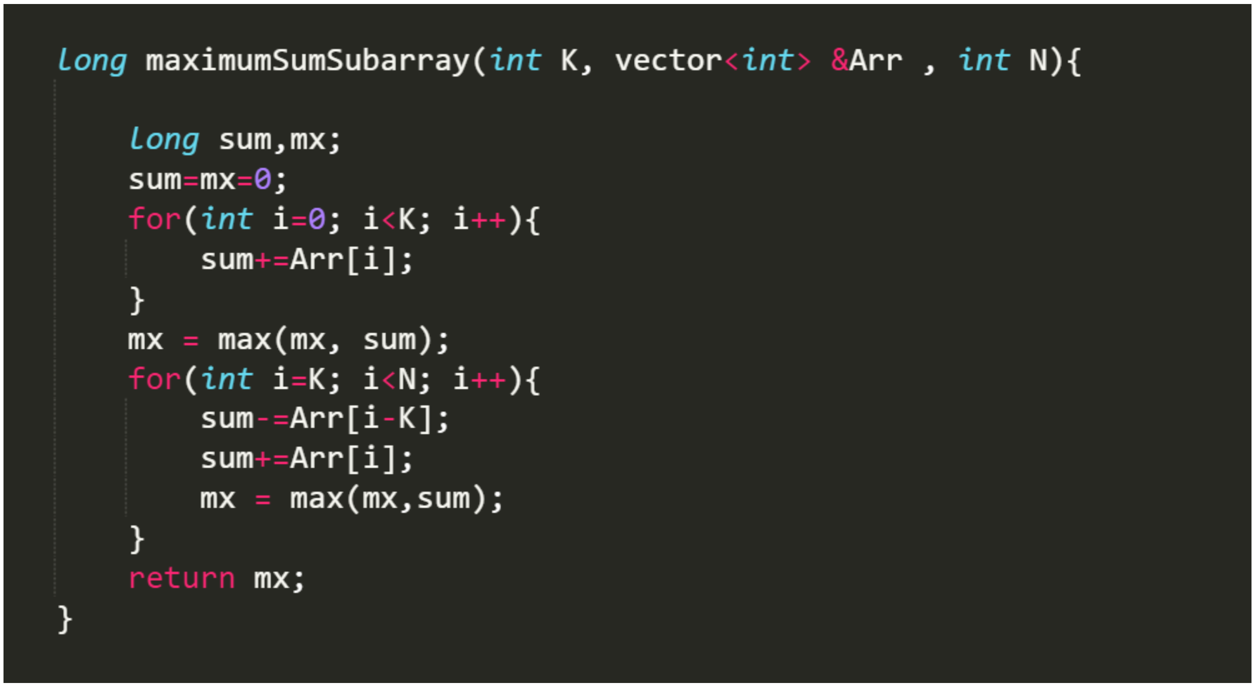
Best way to store the sum……???
Can you think about that?……..Try to think!

Now let’s see……..
Storing the sum -

- You can either calculate the sum every time which will be expensive.
- Or we can just find the sum of the window the first time.
- And then subtract the leftmost element and add the right element as we go, storing the maximum sum till the end.

So, I hope you got it!

So let’s Code!!🧑‍💻🧑‍💻🧑‍💻



```
long maximumSumSubarray(int K, vector<int> &Arr , int N){

    long sum,mx;
    sum=mx=0;
    for(int i=0; i<K; i++){
        sum+=Arr[i];
    }
    mx = max(mx, sum);
    for(int i=K; i<N; i++){
        sum-=Arr[i-K];
        sum+=Arr[i];
        mx = max(mx,sum);
    }
    return mx;
}
```

Time complexity Analysis - We are traversing the array only once - O(n).
Space complexity - O(1).

Problem 2 - Fruits into basket.

I am adding the problem link here - Fruit into basket.
Because the problem statement is bit large.

I hope you read and understand the problem statement very well.

So, this is the time to use your brilliant mind and think about a solution approach.

Super interesting problem, let's learn something cool from it. This is a sliding window problem where we want to keep the maximum of 2 unique fruits at a time.

- We begin with 2 pointers, start and end.
- We move the end pointer when we're exploring stuff in the array -> this is the fast pointer moving ahead.
- We move the start pointer only when we're shrinking the window.

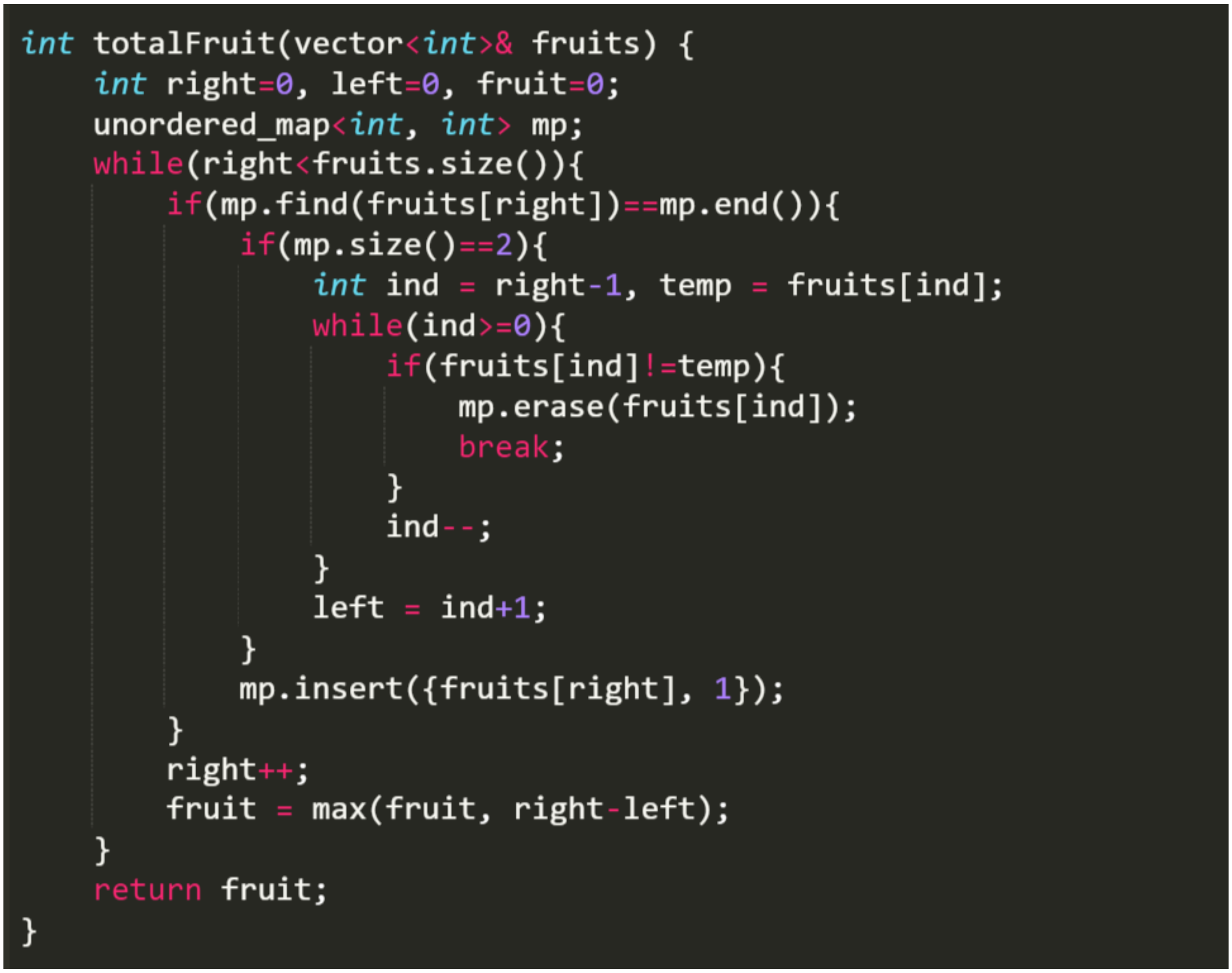
Think of this as expanding the window and shrinking it once we go out of bounds.

Now, how do we expand or shrink here? No clue to be honest, bye.
Haha kidding, let's do it. We expand when we're exploring, so pretty much always when we add an element to our search horizon - we increase the end variable. Let's take this step by step. We have 2 pointers, start/end, and a map -> we add elements with the 'end' pointer and take out elements with the start pointer (shrinking).

Let’s take the end pointer till the array length, add the element to the map and then while the number of unique fruits are more than 2, remove the element from the map

Now, we want to store the maximum window size at all times after the second loop has exited and we’re in the nice condition -> maximum 2 unique fruits. The conditions can be changed here easily according to the number of unique fruits or min/max given to us

So, this is time to Code!!.......You write your own code.
And then compare it with my Code!.😊😊



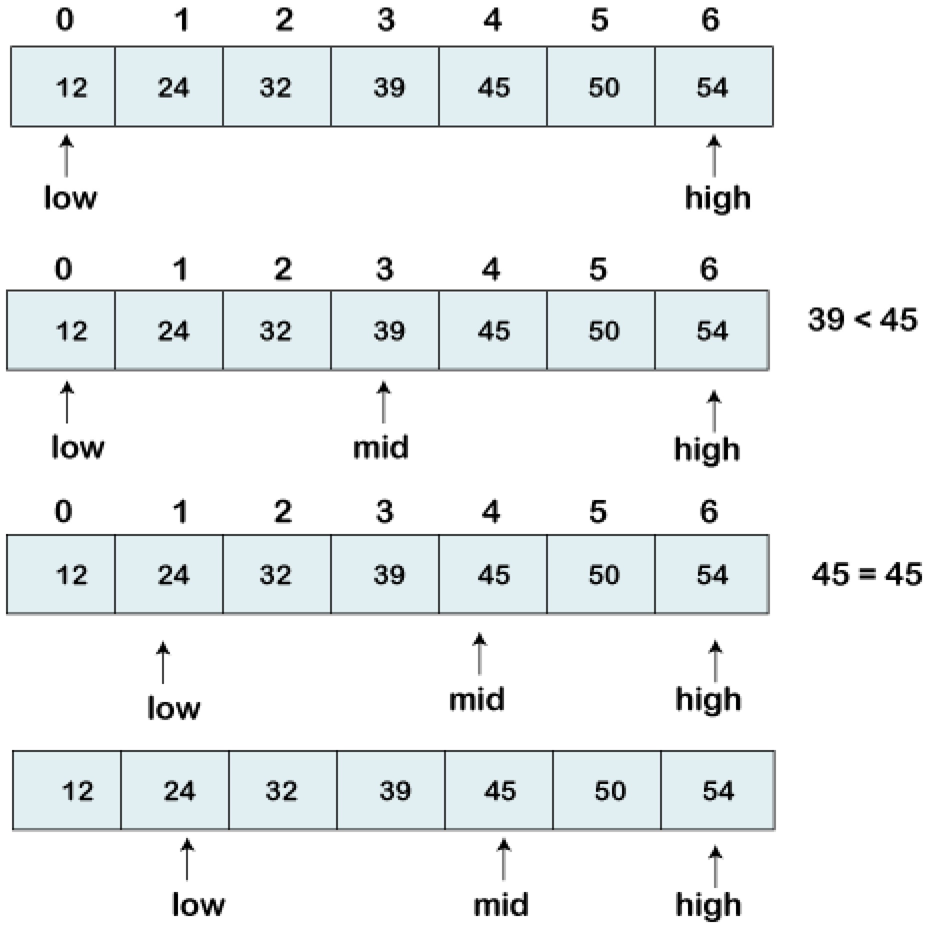
```
int totalFruit(vector<int>& fruits) {
    int right=0, left=0, fruit=0;
    unordered_map<int, int> mp;
    while(right<fruits.size()){
        if(mp.find(fruits[right])==mp.end()){
            if(mp.size()==2){
                int ind = right-1, temp = fruits[ind];
                while(ind>=0){
                    if(fruits[ind]!=temp){
                        mp.erase(fruits[ind]);
                        break;
                    }
                    ind--;
                }
                left = ind+1;
            }
            mp.insert({fruits[right], 1});
        }
        right++;
        fruit = max(fruit, right-left);
    }
    return fruit;
}
```

I hope your code was better than my code!!

Wow! Congrats… you are growing.

“Binary Search”

We use binary search to optimize our search time complexity when the array is sorted (min, max) and has a definite space. It has some really useful implementations, with some of the top companies still asking questions from this domain.

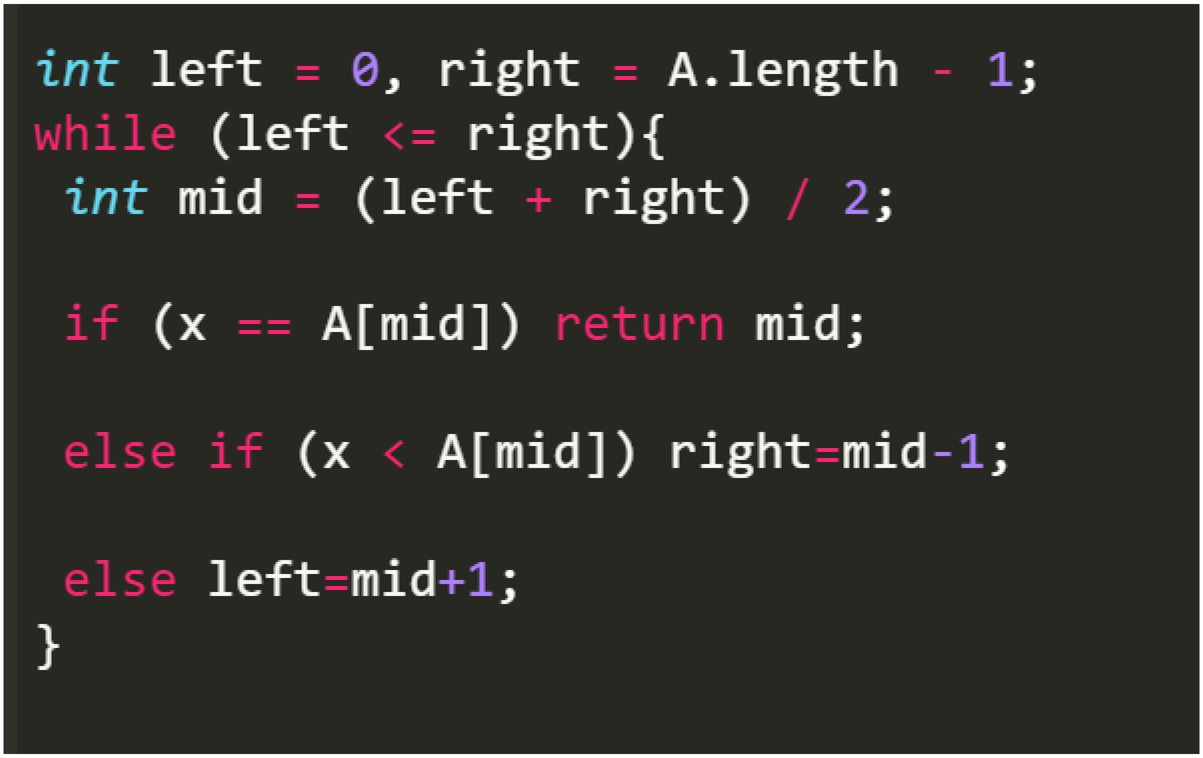


The concept is: if the array is sorted, then finding an element shouldn’t require us to iterate over every element where the cost is O(N). We can skip some elements and find the element in O(logn) time

Algorithm -

- We start with 2 pointers by keeping a low and high.
- Finding the mid and then comparing that with the number we want to find.
- If the target number is bigger, we move right, as we know the array is sorted.
- If it's smaller, we move left because it can't be on the right side, where all the numbers are bigger than the mid value.

Here's an iterative way to write the Binary search algorithm:

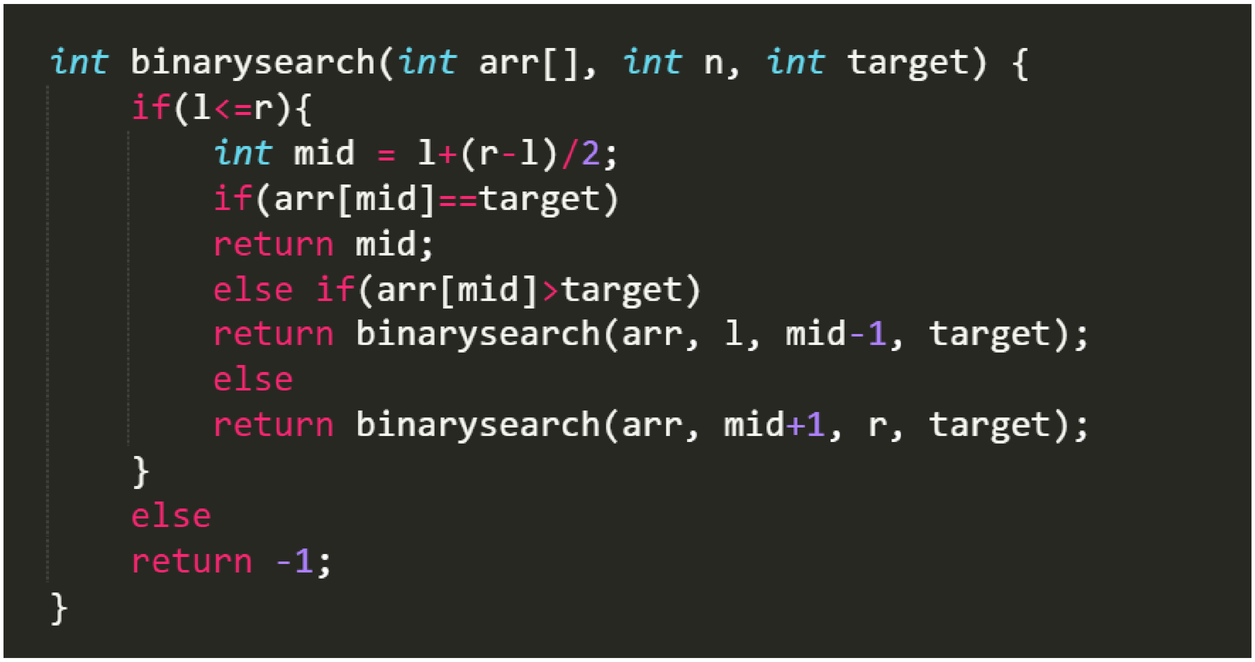


```
int left = 0, right = A.length - 1;
while (left <= right){
 int mid = (left + right) / 2;

 if (x == A[mid]) return mid;

 else if (x < A[mid]) right=mid-1;

 else left=mid+1;
}
```

Let's understand the recursive solution now:

We call the function for the left side and right side if the mid doesn't match our target.

We can either change the left/right pointers through the arguments or through case arguments looks like an easier way.
If we want to move to the right, we change the left pointer to mid+1, and if we wanna go left, we change the right pointer to mid-1.



```
int binarysearch(int arr[], int n, int target) {
    if(l<=r){
        int mid = l+(r-l)/2;
        if(arr[mid]==target)
        return mid;
        else if(arr[mid]>target)
        return binarysearch(arr, l, mid-1, target);
        else
        return binarysearch(arr, mid+1, r, target);
    }
    else
    return -1;
}
```

Problem 1 - Max font size (Google Intern).

Google likes to test you on word problems with core principles. So even if they ask you a binary search question, it will be framed like a real life thing so that it's much harder to understand.
They also test OOPS sometimes, by asking you to create classes and functions to display different things.
Here's the question: Given:

- Height and width of a screen where you have to type.

- Height and width of each character you type on the screen.
- Min and max range of the font size of each character.

Find the maximum font size such that the characters fit inside the screen Once you understand the question, it’s trivial to think of a brute force problem: explore all the possible font sizes and then see what fits at the end.
Return that. Thinking a little more, we see that we have a range (sorted) and we don’t really have to check for each font before choosing the maximum one.

Problem 2: Search in a rotated sorted array.
Problem Link - LeetCode Problem.

Array is sorted but rotated. [4,5,1,2,3] -> 4 came to the front instead of the back.

A brute force is just iterating and finding the element -> O(N).

Can you do it better…???……..Yes You can because you are following “Coding School”.

This is a very interesting problem, because there are a couple of nice optimized solutions and there’s a 50% chance you’ll see one of those (wow, I’m so smart).
I’ve seen a few questions based on this, so it’s important to understand this before moving forward.

Now this problem can be solved with the two logics……

1. Find the pivot and then use Binary Search to find the element.
2. Direct binary search on array without finding pivot.

Finding pivot where rotation has happened.
Here is the idea to solve this problem -
The idea is to find the pivot point, divide the array into two sub-arrays and perform a binary search.

The main idea for finding a pivot is –

For a sorted (in increasing order) and rotated array, the pivot element is the only element for which the next element to it is smaller than it.
Using binary search based on the above idea, pivot can be found.
It can be observed that for a search space of indices in range [l, r] where the middle index is mid,
If rotation has happened in the left half, then obviously the element at l will be greater than the one at mid.
Otherwise the left half will be sorted but the element at mid will be greater than the one at r.
After the pivot is found, divide the array into two sub-arrays.
Now the individual sub-arrays are sorted so the element can be searched using Binary Search.

Now how to implement the above idea…???

Here is the implementation -

Algorithm -

Find out the pivot point using binary search. We will set the low pointer as the first array index and high with the last array index.

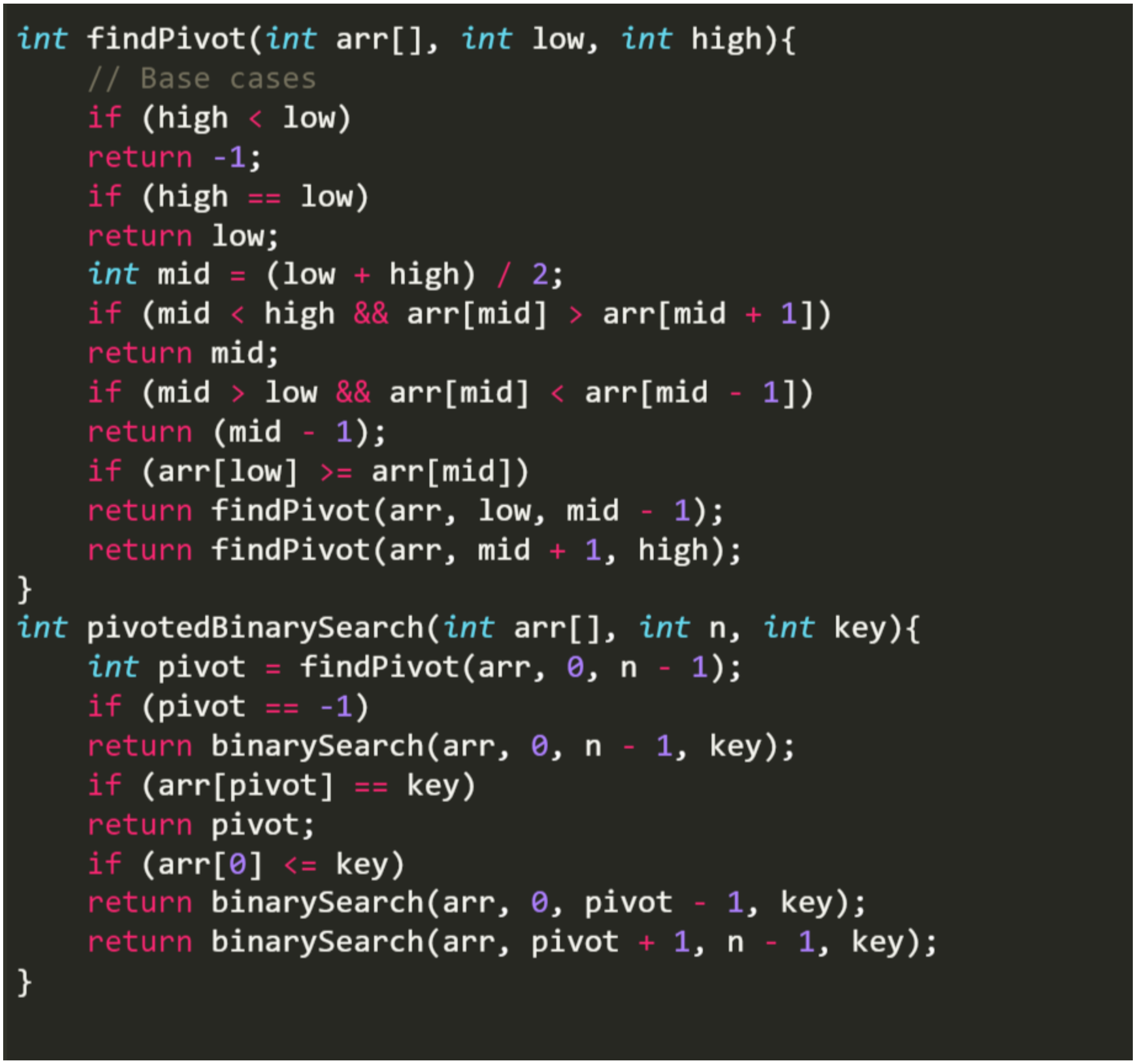
- From the high and low we will calculate the mid value.
- If the value at mid-1 is greater than the one at mid, return that value as the pivot.
- Else if the value at the mid+1 is less than mid, return mid value as the pivot.
- Otherwise, if the value at low position is greater than mid position, consider the left half. Otherwise, consider the right half.

Divide the array into two sub-arrays based on the pivot that was found.
Now call binary search for one of the two sub-arrays.

- If the element is greater than the 0th element then search in the left array
- Else search in the right array.

If the element is found in the selected sub-array then return the index
Else return -1.

So Let’s Code!!🧑‍💻🧑‍💻🧑‍💻



```
int findPivot(int arr[], int low, int high){
    // Base cases
    if (high < low)
    return -1;
    if (high == low)
    return low;
    int mid = (low + high) / 2;
    if (mid < high && arr[mid] > arr[mid + 1])
    return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
    return (mid - 1);
    if (arr[low] >= arr[mid])
    return findPivot(arr, low, mid - 1);
    return findPivot(arr, mid + 1, high);
}
int pivotedBinarySearch(int arr[], int n, int key){
    int pivot = findPivot(arr, 0, n - 1);
    if (pivot == -1)
    return binarySearch(arr, 0, n - 1, key);
    if (arr[pivot] == key)
    return pivot;
    if (arr[0] <= key)
    return binarySearch(arr, 0, pivot - 1, key);
    return binarySearch(arr, pivot + 1, n - 1, key);
}
```

binarySearch() - this function we have discussed already.

Time Complexity - O(logn).

2nd Solution - Without pivot.

The idea is to create a recursive function to implement the binary search where the search region is [l, r]. For each recursive call:

- We calculate the mid value as mid = (l + h) / 2
- Then try to figure out if l to mid is sorted, or (mid+1) to h is sorted
- Based on that decide the next search region and keep on doing this till the element is found or l overcomes h.

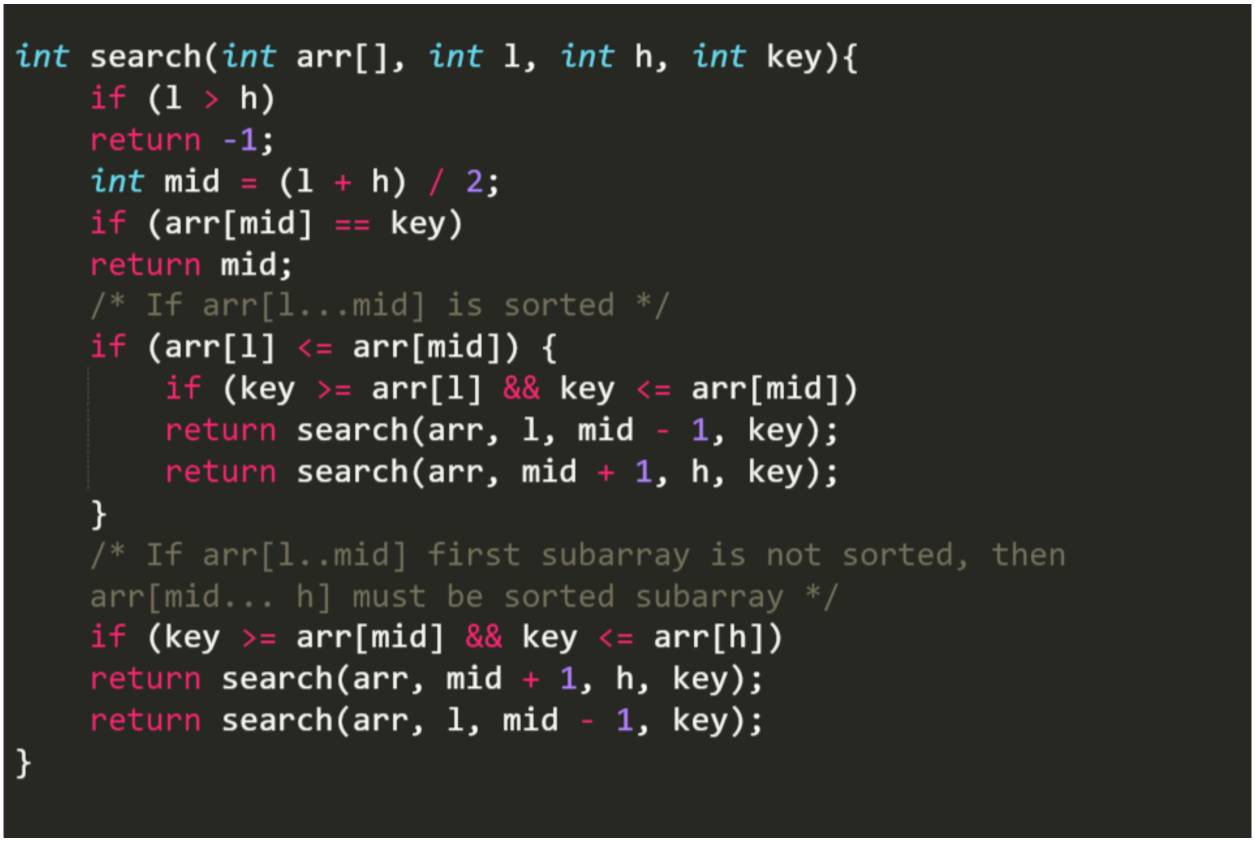
Use a recursive function to implement binary search to find the key: -
Find middle-point mid = (l + h)/2
If the key is present at the middle point, return mid.
Else if the value at l is less than the one at mid then arr[l . . . mid] is sorted

- If the key to be searched lies in the range from arr[l] to arr[mid], recur for arr[l . . . mid].
- Else recur for arr[mid+1 . . . h]

Else arr[mid+1. . . h] is sorted:

- If the key to be searched lies in the range from arr[mid+1] to arr[h], recur for arr[mid+1. . . h].
- Else recur for arr[l. . . mid].

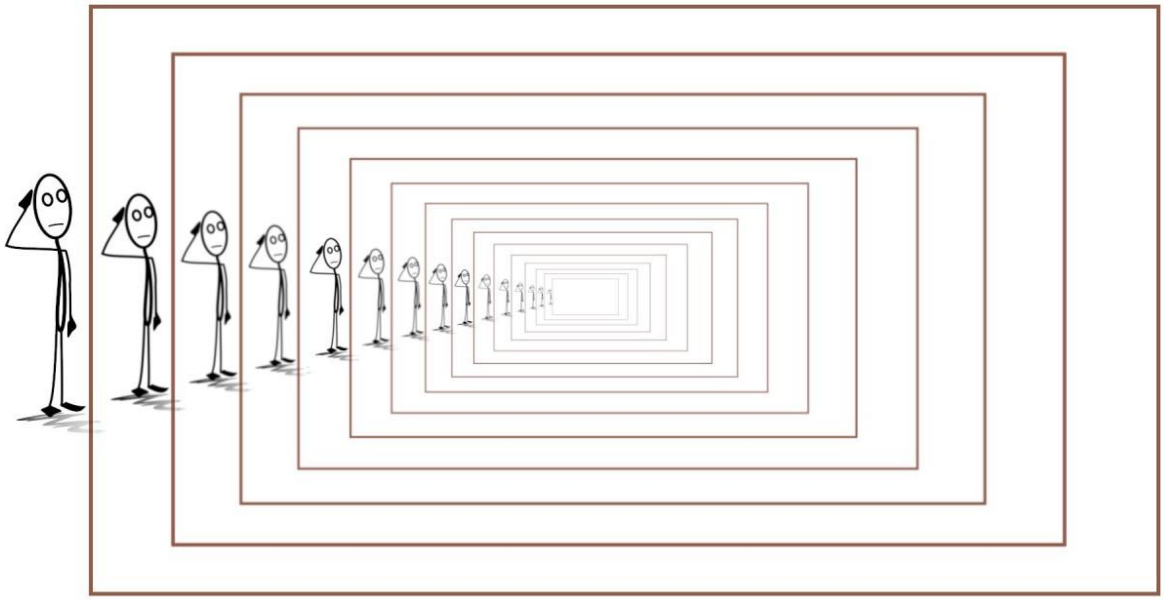
So let’s Code…….🧑‍💻🧑‍💻🧑‍💻



```
int search(int arr[], int l, int h, int key){
    if (l > h)
    return -1;
    int mid = (l + h) / 2;
    if (arr[mid] == key)
    return mid;
    /* If arr[l...mid] is sorted */
    if (arr[l] <= arr[mid]) {
        if (key >= arr[l] && key <= arr[mid])
        return search(arr, l, mid - 1, key);
        return search(arr, mid + 1, h, key);
    }
    /* If arr[l..mid] first subarray is not sorted, then
    arr[mid... h] must be sorted subarray */
    if (key >= arr[mid] && key <= arr[h])
    return search(arr, mid + 1, h, key);
    return search(arr, l, mid - 1, key);
}
```

“Recursion”

Think of it as solving smaller problems to eventually solve a big problem.
So if you want to climb Mount Everest, you can recursively climb the smaller parts until you reach the top.
Another example is that you want to eat ‘15 butter naan’, so eating all of them at once won’t be feasible. Instead, you would break down those into 1 at a time, and then enjoy it on the way.



If you become expert in recursion, then you will be also become master in solving problems related to these three important topics-

- Recursion. (“Ye to h hi” 😂😂).
- Backtracking.
- Dynamic Programming.

I have found a superb video for you to understand the recursion deeply and you will feel how recursion works actually.
Link - Recursion Amazing Video.

Wait wait……Video pura dekha na…???

Achha dekh liya, bhut bdia…….ok now you are familiar with the recursion. Maybe I can say now you are almost a master in recursion.

Now let's move further and remove the word "almost" from "almost master"…….because you are following "Coding School" so it's my duty to take you to the expert level of thinking.

So let's Move……

These are some questions I have when I look at a recursive question/solution, you probably have the same.
Let's try to figure them out -

- What happens when the function is called in the middle of the whole recursive function?
- What happens to the stuff below it?
- What do we think of the base case?
- How do we figure out when to return ?
- How do we save the value, specially in the true/false questions?
- How does backtracking come into place, wrt recursion?

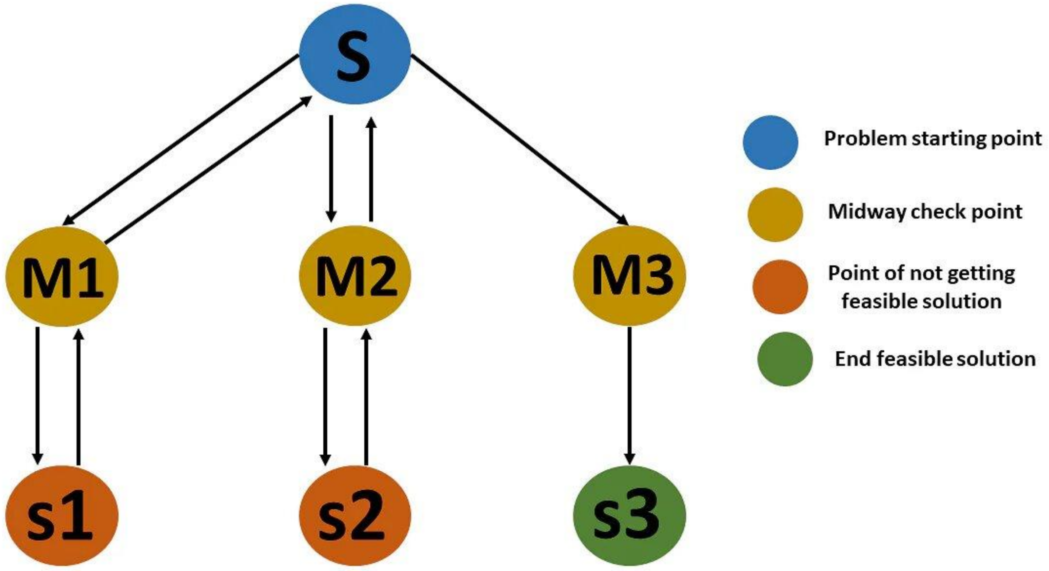
Let's try to answer these one by one. A recursive function means that we're breaking the problem down into a smaller one. So if we're saying function(x/2) -> we're basically calling the function again with the same parameters.

So if there's something below the recursive function -> that works with the same parameter. For instance, calling function(x/2) with x=10 and then printing (x) after that would print 10 and then 5 and so on. Think of it as going back to the top of the function, but with different parameters.

The return statements are tricky with recursive functions.
You can study about those things, but practice will help you get over it.
For instance, you have fibonacci, where we want to return the sum of the last 2 elements for the current element -> the code is something like fib(n) + fib(n-1) where fib() is the recursive function.
So this is solving the smaller problem until when? -> Until the base case. And the base case will return 1 -> because eventually we want the fib(n) to return a number.
This is a basic example, but it helps you gain some insights on the recursive part of it.

Something complex like dfs or something doesn't really return anything but transforms the 2d matrix or the graph.

Note - Backtracking is nothing but exploring all the possible cases by falling back or backtracking and going to other paths.



BackTracking.

Problem 1 - Generate Parentheses.

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Don’t understand what the problem actually is …….??
Ok don’t worry….I am here …..for you to make you understand.

So, for better understanding let’s see an example.
Input n=3.
Output - “((()))”, “(()())”, “(())()”, “()(())”, “()()()”.
So, here are total of 5 well-formed combinations of parentheses.

I think now you got it……….ok……but now think about the solution.

You are trying to think about how to solve this problem using recursion….because we are learning recursion, so as you are smart so you just start thinking about recursion.

But it’s not a good practice…..it may be that your mind can solve this problem with another method.
So first forget about recursion, and then try to think as normal.

Ok, now let’s come to our goal…….that is recursion.

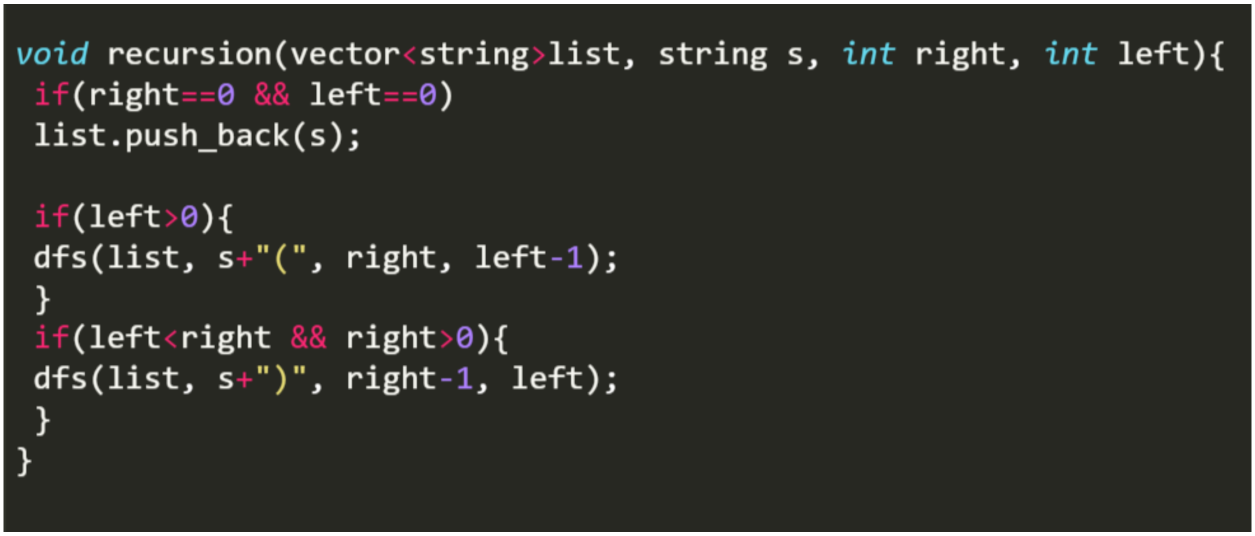
In simple words, we want to print out all the possible cases -> valid parentheses can be generated.
One thing which strikes me is -> we need a way to add “(” and “)” to all possible cases and then find a way to validate so that we don’t generate the unnecessary ones.

The first condition is if there are more than 0 open / left brackets, we recurse with the right ones. And if we have more than 0 right brackets, we recurse with the left ones. Left and right are initialized at N - the number given.

There’s a catch. We can’t add the “)” everytime we have right>0 cause then it will not be balanced. We can balance that with a simple condition of left cause we’re subtracting one as we go down to 0.

So, now this is time to code………go and code first….and then compare your code with my code…..i hope you will write a better code than me!!

So let’s Code…..🧑‍💻🧑‍💻🧑‍💻



```
void recursion(vector<string>list, string s, int right, int left){
 if(right==0 && left==0)
 list.push_back(s);

 if(left>0){
 dfs(list, s+"(", right, left-1);
 }
 if(left<right && right>0){
 dfs(list, s+")", right-1, left);
 }
}
```

Problem 2 - Reverse Linked List.

Note - This problem cover two topics -

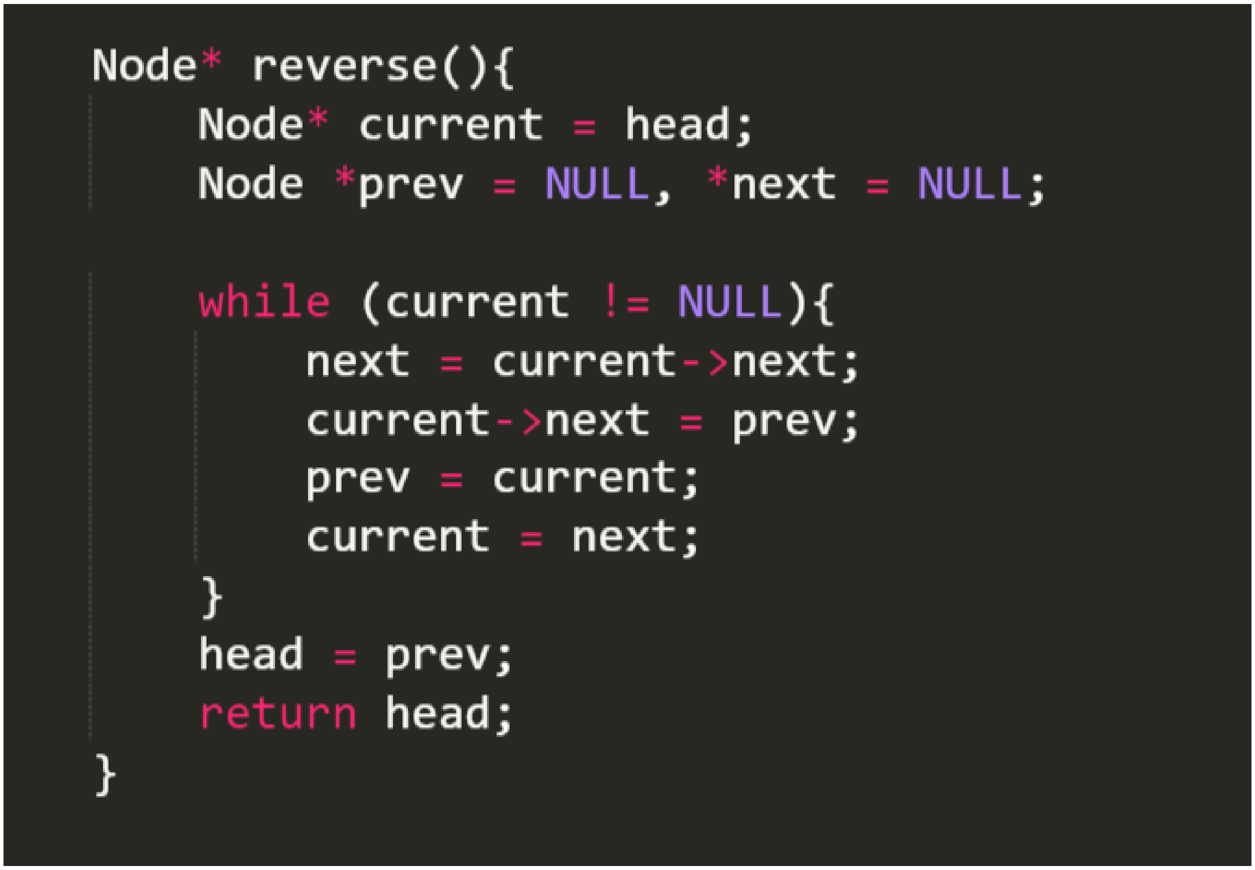
- Linked list.
- Recursion.

Although this requires linked list knowledge, this is more of a recursion question.

Let’s try to solve this both iteratively and recursively to see what really is going on.
Let’s discuss a short iterative way of doing this.

- Move ahead with a pointer.
- Point the current to previous -> curr.next = prev.
- Move the prev by changing it to curr.

So let’s code!!🧑‍💻🧑‍💻🧑‍💻

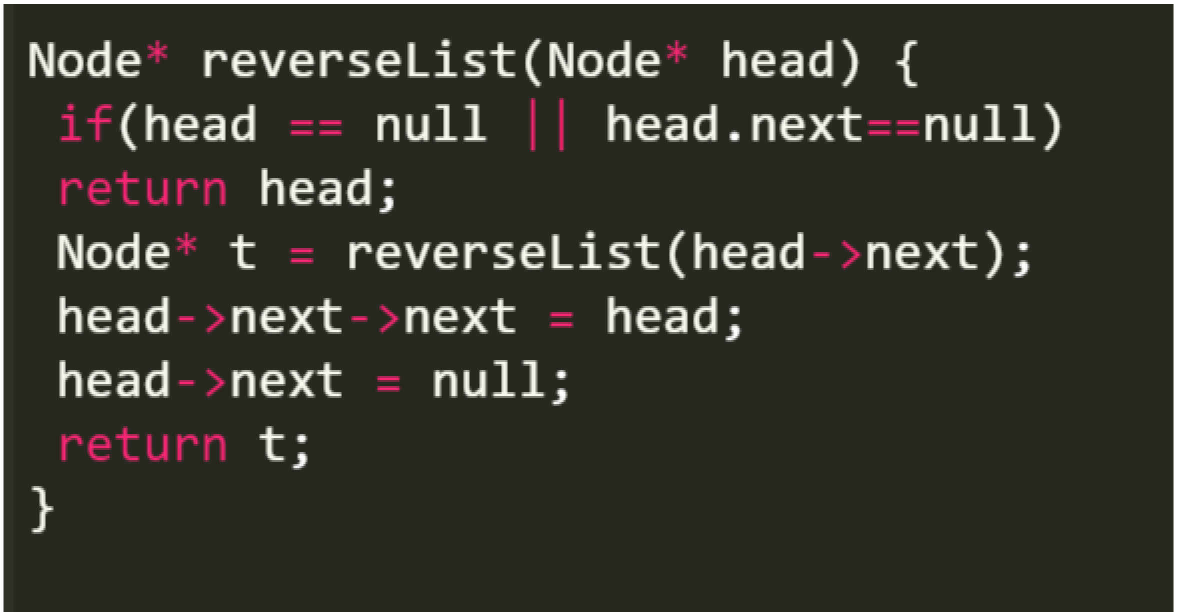


```
Node* reverse(){
    Node* current = head;
    Node *prev = NULL, *next = NULL;

    while (current != NULL){
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
    return head;
}
```

We can also solve this one using recursion.
Here is the implementation -

- Store the recursive call in a node -> This takes the pointer to the end.
- Point the curr's next pointer to that.
- Point head's next to null -> this will be the tail (at every instance)

So let's code....



```
Node* reverseList(Node* head) {
 if(head == null || head.next==null)
 return head;
 Node* t = reverseList(head->next);
 head->next->next = head;
 head->next = null;
 return t;
}
```

I am adding a video here for you to understand this problem very well - Reverse list.

“BackTracking”

Backtracking can be seen as an optimized way to brute force. Brute force approaches evaluate every possibility.
In backtracking you stop evaluating a possibility as soon as it breaks some constraint provided in the problem, take a step back and keep trying other possible cases, see if those lead to a valid solution.

Think of backtracking as exploring all options out there, for the solution. You visit a place, there’s nothing after that, so you just come back and visit other places.

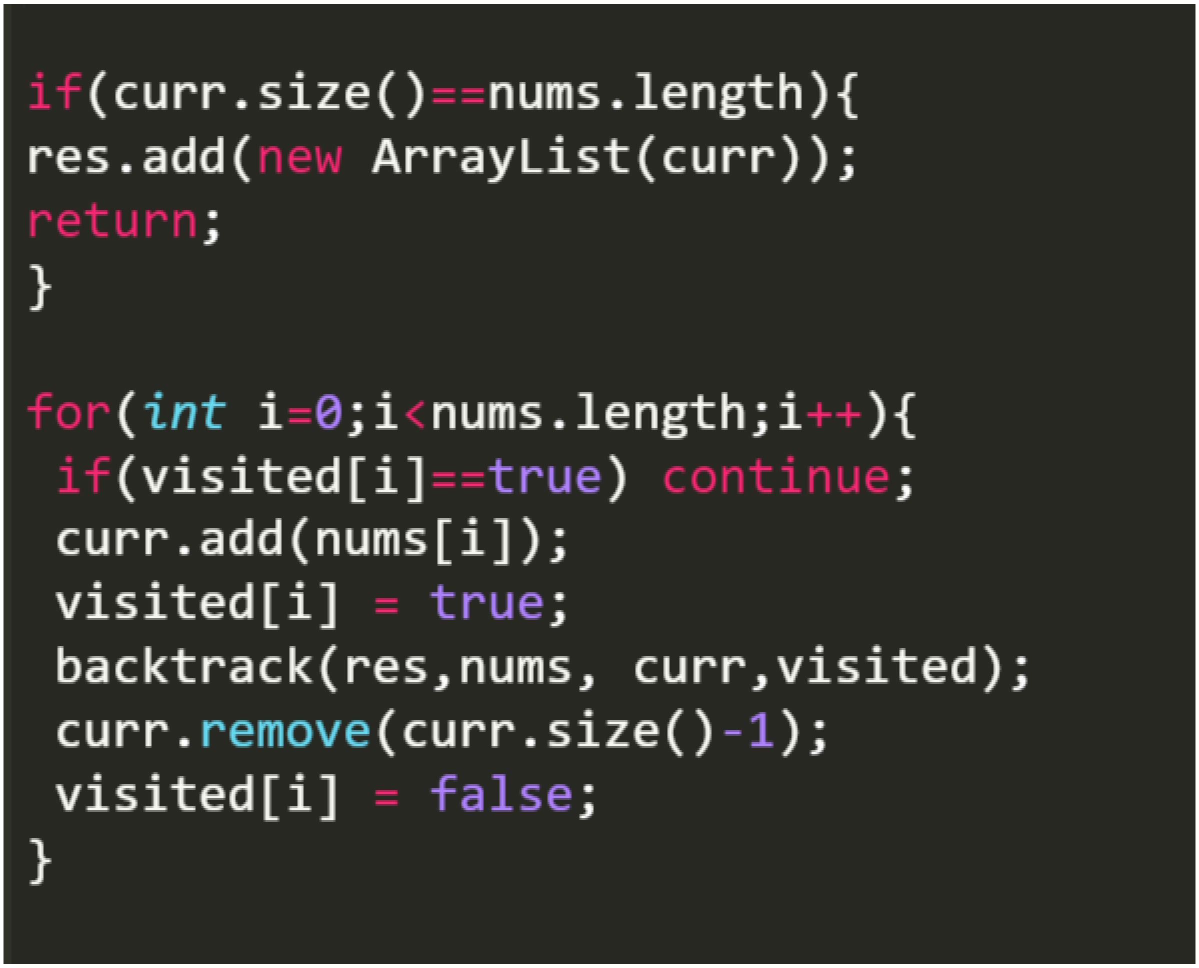
Here’s a nice way to think of any problem:

- Recognize the pattern.
- Think of a human way to solve it.
- Convert it into code.

Problem 1 - Permutations.

We have an array [1,2,3] and we want to print all the possible permutations of this array.
The initial reaction to this is - explore all possible ways -> somehow write 2,1,3, 3,1,2 and other permutations.
Second step, we recognize that there’s a pattern here. We can start from the left - add the first element, and then explore all the other things with the rest of the items.
So we choose 1 -> then add 2,3 and 3,2 - > making it [1,2,3] and [1,3,2]. We follow the same pattern with others.
How do we convert this into code?

- Base case.
- Create a temporary list.
- Iterate over the original list.
  Add an item + mark them visited.
  Call the recursive function.
  Remove the item + mark them univisited.



```
if(curr.size()==nums.length){
res.add(new ArrayList(curr));
return;
}

for(int i=0;i<nums.length;i++){
 if(visited[i]==true) continue;
 curr.add(nums[i]);
 visited[i] = true;
 backtrack(res,nums, curr,visited);
 curr.remove(curr.size()-1);
 visited[i] = false;
}
```

There are also other solutions to problems like this one, where you can modify the recursive function to pass in something else.

We can pass in something like this: function(array[1:]) -> to shorten the array every time and then have the base case as len(arr) == 0.

Problem 2 - N Queens.
Problem Link - N-Queens.

We want to place 8 queens such that no queen is interacting with each other.
We see a similar pattern, where the thinking goes like this -> we want to explore all possible ways such that eventually we find an optimal thing, where queens don't interact with each other.

We start by placing the first, then second… until there's a conflict.
We then would have to come back, change the previous queens, until we find the optimal way.
We would have to go back to the very start as well, and maybe try the whole problem again.

How to convert this into code?

Similar to most backtracking problems, we will follow a similar pattern:

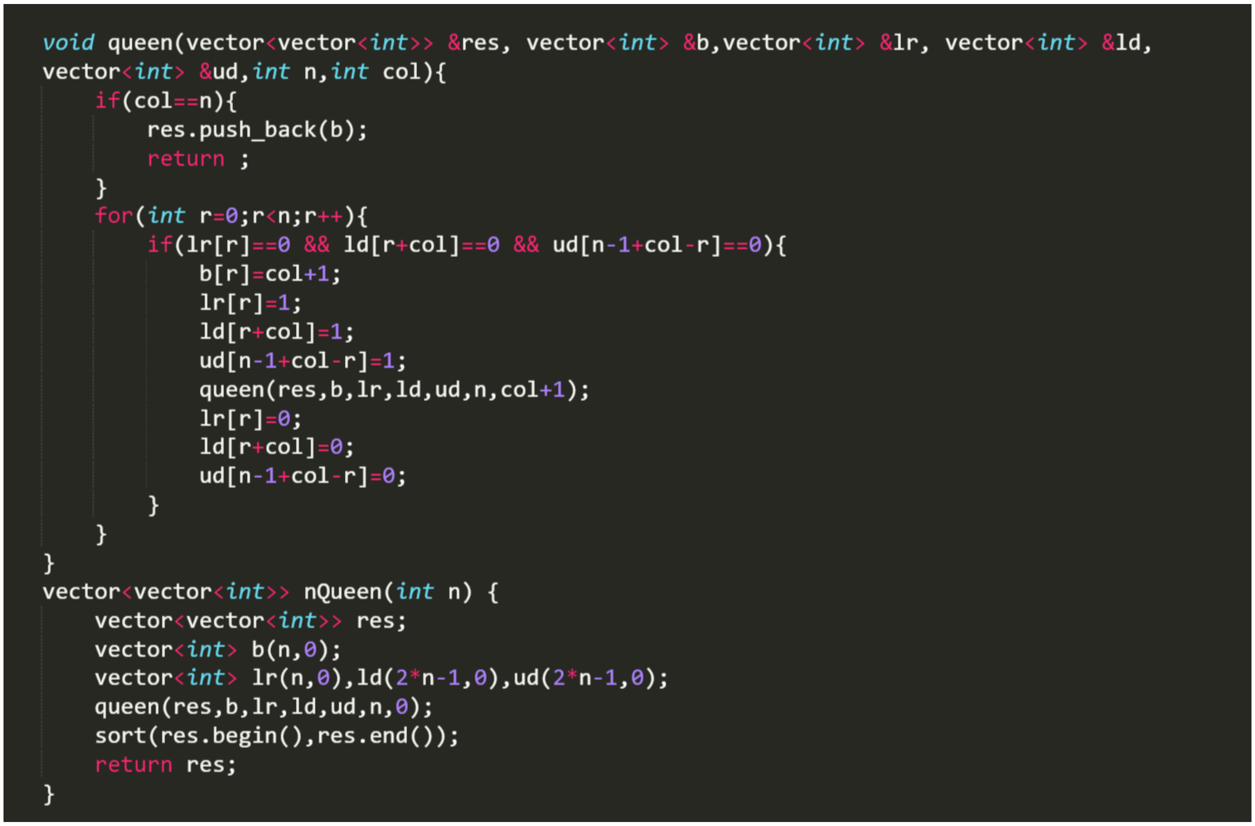
- Place the queen on a position.
- Check if that position is valid === Call the recursive function with this new position.
- Remove the queen from that position.

Make sure to think about the base cases, recursive calls, the different parameters, and validating functions.

I am adding an algo visualizer link here, for better visualization.
Link - [Visualize the Solution](#).

Summary -
You can solve this problem using a backtracking approach.
The idea is to place queens one by one in different columns, and for each queen, check if it is safe to place it in that row of that column.
If it is safe, place the queen and move to the next column. If not, backtrack and try placing the queen in the next row of the same column.
Once all the queens are placed in all the columns, add the current configuration.

So let's Code!!🧑‍💻🧑‍💻🧑‍💻



```cpp
void queen(vector<vector<int>> &res, vector<int> &b,vector<int> &lr, vector<int> &ld,
vector<int> &ud,int n,int col){
    if(col==n){
        res.push_back(b);
        return ;
    }
    for(int r=0;r<n;r++){
        if(lr[r]==0 && ld[r+col]==0 && ud[n-1+col-r]==0){
            b[r]=col+1;
            lr[r]=1;
            ld[r+col]=1;
            ud[n-1+col-r]=1;
            queen(res,b,lr,ld,ud,n,col+1);
            lr[r]=0;
            ld[r+col]=0;
            ud[n-1+col-r]=0;
        }
    }
}
vector<vector<int>> nQueen(int n) {
    vector<vector<int>> res;
    vector<int> b(n,0);
    vector<int> lr(n,0),ld(2*n-1,0),ud(2*n-1,0);
    queen(res,b,lr,ld,ud,n,0);
    sort(res.begin(),res.end());
    return res;
}
```

Memoization -

Memoization means storing a repetitive value, so that we can use it for later.

A really nice example here:

- If you want to climb Mount Everest, you can recursively climb the smaller parts until you reach the top. The base case would be the top, and you would have a recursive function climb() which does the job.
- Imagine if there are 4 camps to Mount Everest, your recursive function would make you climb the first one, then both 1 and 2, then 1-2-3 and so on. This would be tiring, cost more, and a lot of unnecessary work. Why would you repeat the work you've already done? This is where memoization comes in.

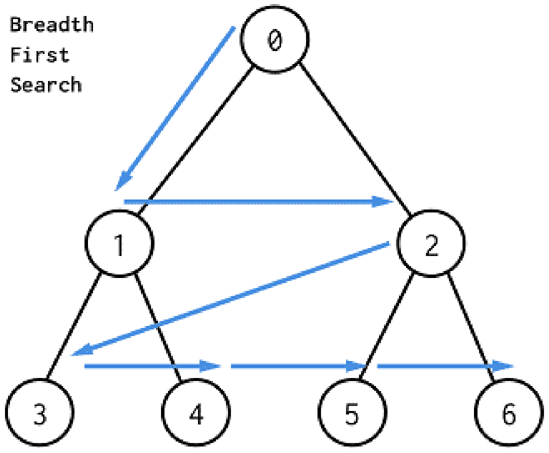
- If you use memoization, you would store your camp ground once you reach it, so the next time your recursive function works, it'll get the camp ground value from the stored set.

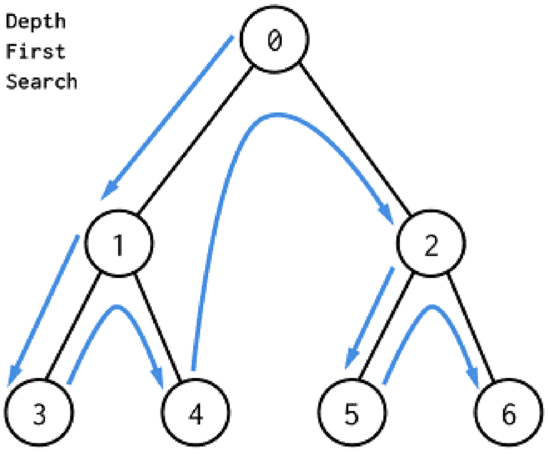
Dynamic programming is Backtracking + Memoization. That's it. Every problem is a part of this algorithm -> explore all possible ways and then optimize them in such a way that we don't explore already explored paths.
Stop solving dynamic programming problems the iterative way. Practice tons of recursion + backtracking problems, and then go the iterative way.

“BFS, DFS”

These are searching techniques to find something.
It’s valid everywhere: arrays, graphs, trees, etc.
A lot of people try to confuse this with being something related to graphs, but no -> this is just a technique to solve a generic search problem.

Here I am adding a BFS/DFS visualizer, you can see how BFS/DFS works - Visualize BFS/DFS.





Try to understand the iterative way of solving a DFS or BFS question and how things work.
There are some basic things: -

- Push the first node.
- Iterate over all nodes (first time it’s just the root).
- Pop the top element.
- Add the neighbors.
- Repeat (Usually through the while or for loop).

BFS Implementation -

- Declare a queue and insert the starting vertex.
- Initialize a visited array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
  Remove the first vertex of the queue.
  Mark that vertex as visited.
  Insert all the unvisited neighbors of the vertex into the queue.

In BFS first we traverse all the neighbors of a particular vertex. And then we move to the next vertex.

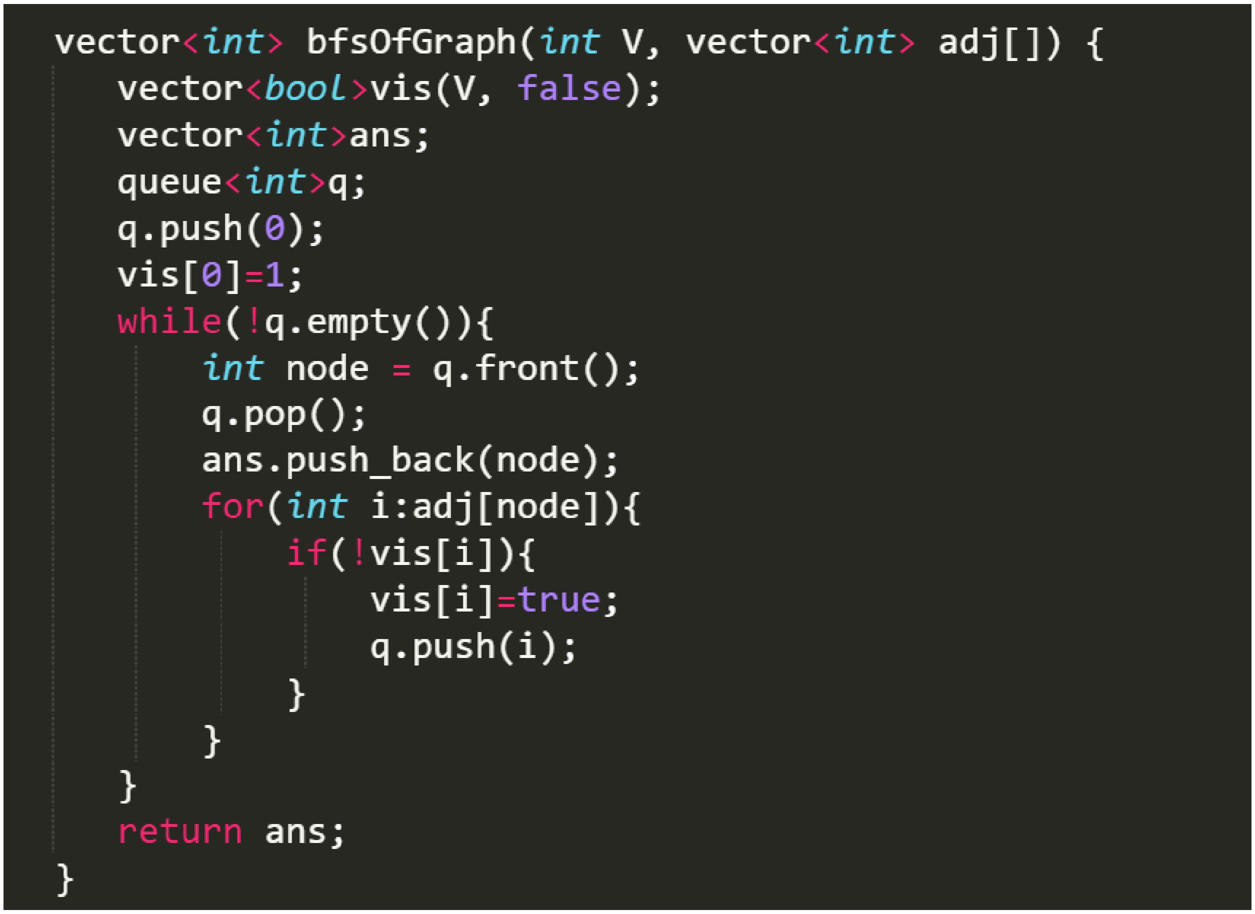
But in DFS we traverse a path completely, and then go to the next neighbor.

So, now I think you are not confused with BFS and DFS.
BFS means - Traverse level by level.
Like in a tree, first traverse its first level and then go to the next level.

But in DFS we have to traverse the selected path completely, like in a tree if we select a path, then before moving to any other node, first traverse this path completely.

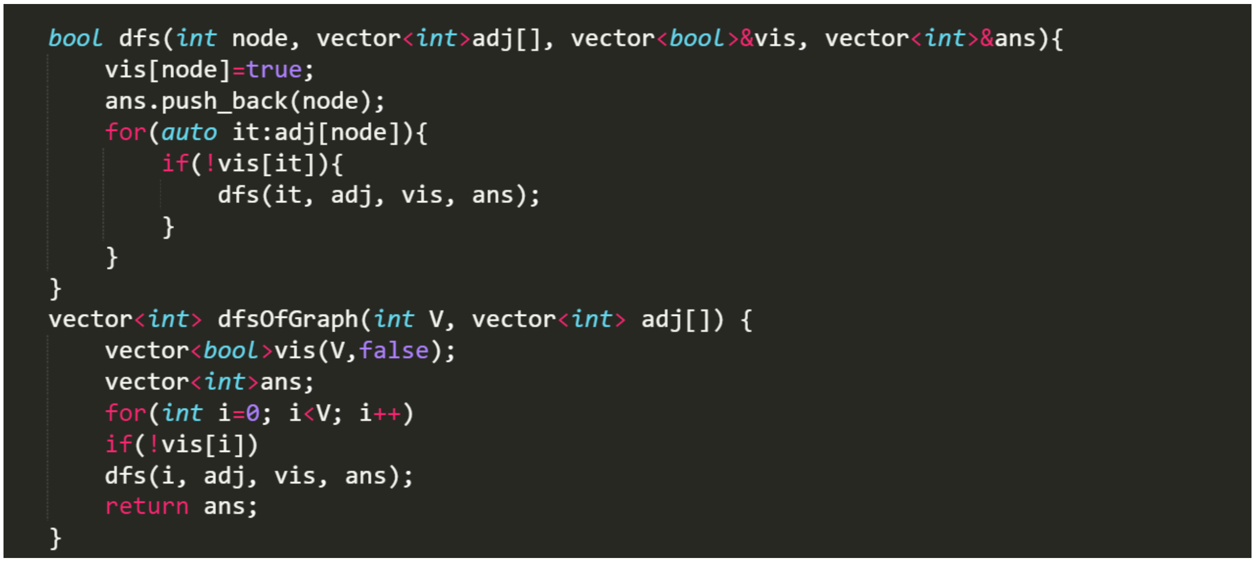
We use a visited array, initially all the nodes are unvisited. And if we visit any node, then we mark it as visited. Such that we will not traverse the same node.

Now Let’s see the codes…….



```
vector<int> bfsOfGraph(int V, vector<int> adj[]) {
    vector<bool>vis(V, false);
    vector<int>ans;
    queue<int>q;
    q.push(0);
    vis[0]=1;
    while(!q.empty()){
        int node = q.front();
        q.pop();
        ans.push_back(node);
        for(int i:adj[node]){
            if(!vis[i]){
                vis[i]=true;
                q.push(i);
            }
        }
    }
    return ans;
}
```

DFS Implementation -



```
bool dfs(int node, vector<int>adj[], vector<bool>&vis, vector<int>&ans){
    vis[node]=true;
    ans.push_back(node);
    for(auto it:adj[node]){
        if(!vis[it]){
            dfs(it, adj, vis, ans);
        }
    }
}
vector<int> dfsOfGraph(int V, vector<int> adj[]) {
    vector<bool>vis(V,false);
    vector<int>ans;
    for(int i=0; i<V; i++)
    if(!vis[i])
    dfs(i, adj, vis, ans);
    return ans;
}
```

Why are we discussing the implementations for a simple search algorithm?

Because this is the basic thing that you need for a lot of problems.
A lot of graph problems require you to know dfs, bfs and this is one of those things, which is usually used with a combination of things.
For instance, you have a 2D matrix with something inside it, and you want the shortest path -> boom, BFS.
Or maybe you have a graph where you want to find the vertex of it -> boom, DFS/BFS.
So it comes in many forms, and it's very important to understand it completely before moving forward.

Here are some implementations and use cases for DFS, BFS:
DFS: -

- Find connected components in a graph.
- Calculate the vertex or edges in a graph.
- Whether the graph is strongly connected or not.
- Wherever you want to explore everything or maybe go in depth.

BFS -

- Shortest path algorithms and questions.
- Ford Fulkerson algorithm.
- Finding nodes in a graph.
- Wherever there is a shortest thing, finding something quickly, etc

Problem 1 - Number of Islands.

Problem link - No. of islands.

Understanding this will definitely open your eyes about the visualization that happens in a search algorithm.

Let's go!

We have a 2d matrix, with 0's and 1's or some other symbols.

We want to find the islands -> where one island is one of more grid nodes which are connected together.

Here's an example:

Input: grid =

```
[ ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"] ]
```

Output: 3.

Because there are 3 islands(with connected ones).

All green ones(4) - 1 Island.

All pink ones(1) - 1 Island.

All blue ones(2) - 1 Island.

So a total 3 Island.

We want to connect all the 1's together, so that we form an island and then count those islands. A human way to do this is just count the connected 1's and then keep a track of those.

How do we code it?

The core principle of DFS kicks in -> we start from the 1st node, pop it, mark it visited, explore all the neighbors, and then repeat.
Once this exploration is done, we start with another 1, explore all of its connected 1's and then mark those visited.
Every time we explore a new node island, we increase the count by 1 and eventually return that number.
Sounds easy?

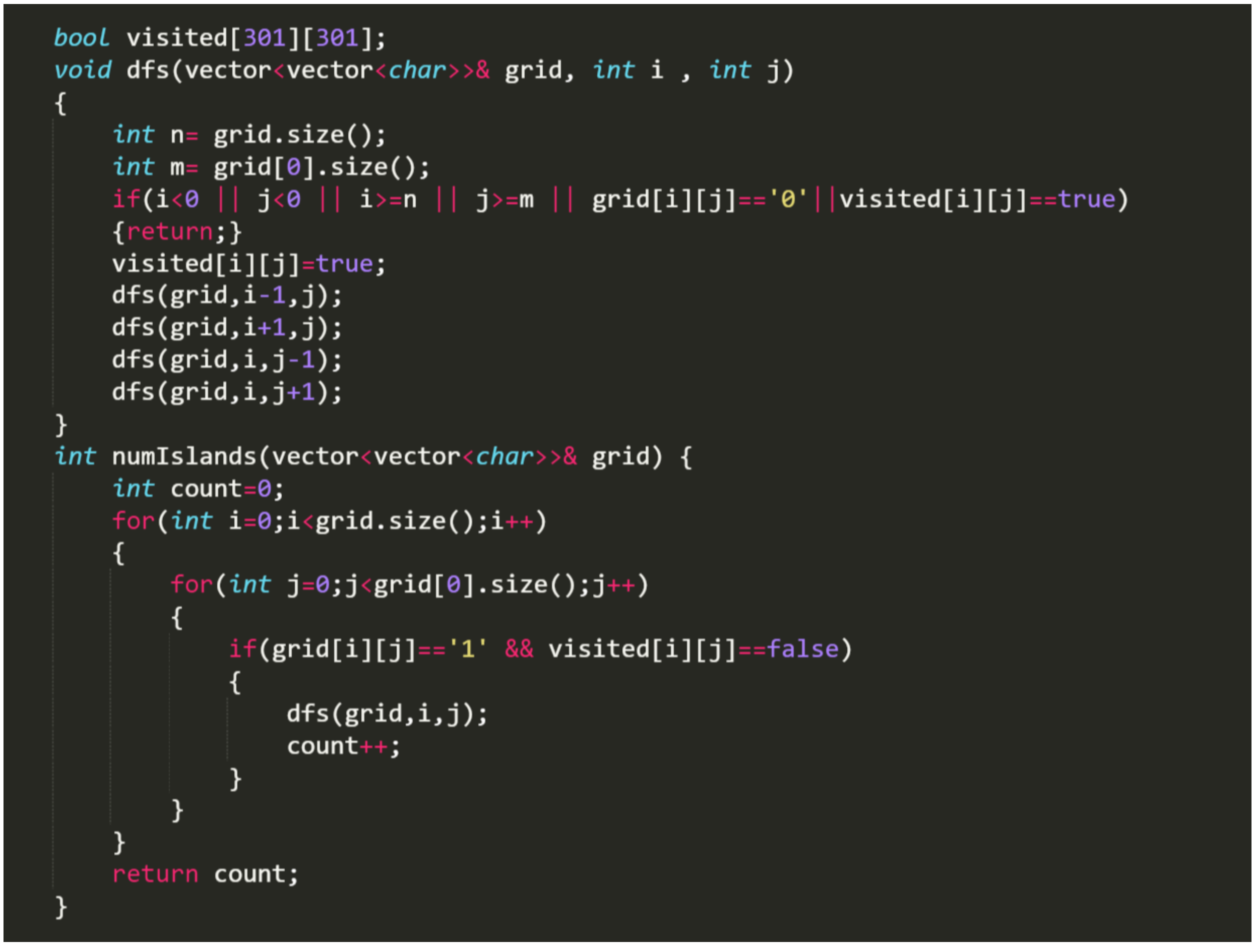
Go code it first... I'm waiting.

Glad you're back.

Here's a recursive implementation:

- We explore every element in the grid.
- If we see a 1, we call the dfs function on it, which counts the connected nodes, and turns those into something other than 1.
- We also increase the count every time we see a NEW node with 1.
- In the dfs method below, we take in the grid element, explore all the sides (top, right, bottom, left), and mark the node to something else every time.

- We’re not counting anything in the dfs function, just exploring all sides, changing the digit, and basically COVERING the island up.



```
bool visited[301][301];
void dfs(vector<vector<char>>& grid, int i , int j)
{
    int n= grid.size();
    int m= grid[0].size();
    if(i<0 || j<0 || i>=n || j>=m || grid[i][j]=='0'||visited[i][j]==true)
    {return;}
    visited[i][j]=true;
    dfs(grid,i-1,j);
    dfs(grid,i+1,j);
    dfs(grid,i,j-1);
    dfs(grid,i,j+1);
}
int numIslands(vector<vector<char>>& grid) {
    int count=0;
    for(int i=0;i<grid.size();i++)
    {
        for(int j=0;j<grid[0].size();j++)
        {
            if(grid[i][j]=='1' && visited[i][j]==false)
            {
                dfs(grid,i,j);
                count++;
            }
        }
    }
    return count;
}
```

I am adding a video for you here - you can watch for more explanation - No. of Islands.

Problem 2 - Level Order Traversal of Binary Tree.

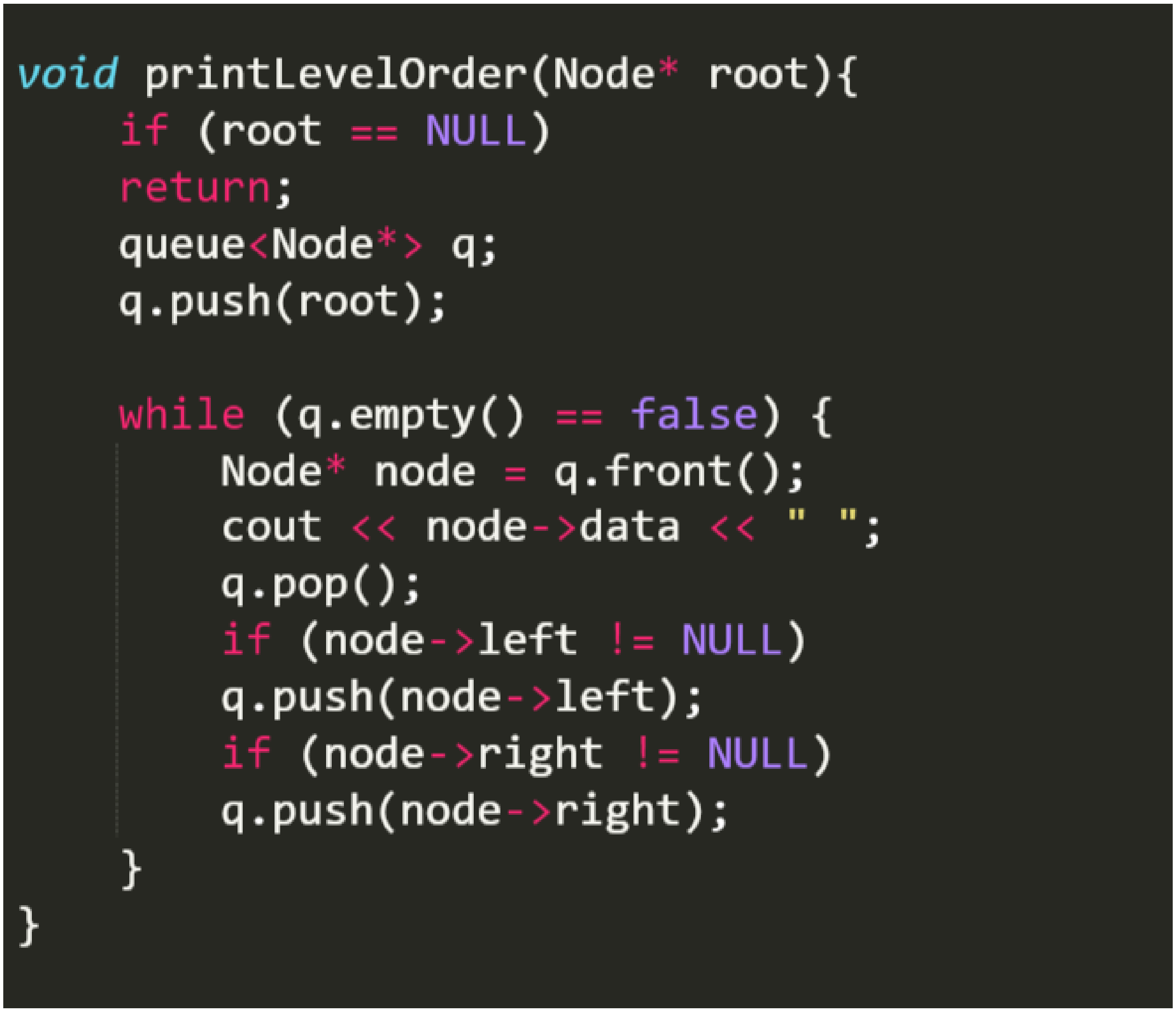
Problem Link - Level Order Traversal.

Print the tree in a level order -> left to right, level by level.

- Try to visualize before writing code.
- How can we get the levels at once?
- How does the core of DFS/BFS/Stack/Queue work?

Here is the implementation -
Follow the below steps to Implement the above idea:

- Create an empty queue q and push root in q.
- Run While loop until q is not empty.
  Initialize temp_node = q.front() and print temp_node->data.
  Push temp_node's children i.e. temp_node -> left then temp_node -> right to q
  Pop front node from q.



```
void printLevelOrder(Node* root){
    if (root == NULL)
    return;
    queue<Node*> q;
    q.push(root);

    while (q.empty() == false) {
        Node* node = q.front();
        cout << node->data << " ";
        q.pop();
        if (node->left != NULL)
        q.push(node->left);
        if (node->right != NULL)
        q.push(node->right);
    }
}
```

Now, we are going to start with your most favorite topic.

Can you guess…….???

Dynamic Programming……..

😂😂😂What??? It is not your favorite topic?

Don't worry, after this you will learn it very well.

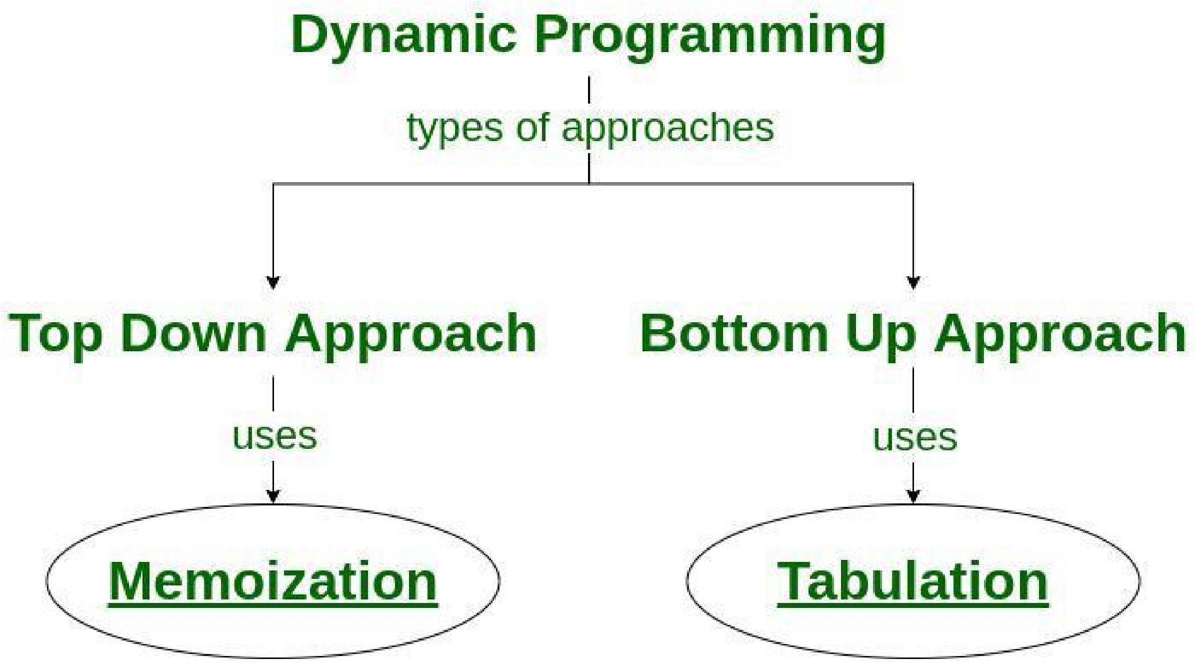
“Dynamic programming”

Dynamic programming is nothing but **recursion + memoization**.
If someone tells you anything outside of this, share this resource with them.

The only way to get good at dynamic programming is to be good at recursion first.
You definitely need to understand the magic of recursion and memoization before jumping to dynamic programming.
The day when you solve a new question alone, using the core concepts of dynamic programming -> you’ll be much more confident after that.

So if you’ve skipped the recursion, backtracking, and memoization section -> go back and complete those first!

If you’ve completed it, keep reading.
You will only get better at dynamic programming (and problem solving in general) by solving more recursion (logical) problems.



Top-down vs Bottom-up approach in Dynamic Programming

There are two ways to solve and implement dynamic programming problems:
1) The top-down approach and
2) The bottom-up approach.
Both approaches perform similarly in one way:
They use extra memory to store the solution to sub-problems, avoid recomputation and improve performance by a huge margin.
On the other hand, both of them are different in so many ways, and understanding this difference would help us to make critical decisions during problem-solving.

How does the top-down approach work?

In the top-down approach, we implement the solution naturally using recursion but modify it to save the solution of each subproblem in an array or hash table.
This approach will first check whether it has previously solved the subproblem.
If yes, it returns the stored value and saves further calculations. Otherwise, top-down approach will calculate sub-problem solutions in the usual manner.
We say it is the memoized version of a recursive solution,

i.e., it remembers what results have been computed previously.

How does the bottom-up approach work?

On the other hand, the bottom-up approach is just the reverse but an iterative version of the top-down approach.
It depends on a natural idea: solution of any subproblem depends only on the solution of smaller subproblems.
So the bottom-up approach sorts the subproblems by their input size and solves them iteratively in the order of smallest to largest.

In other words, when solving a particular subproblem, the bottom-up approach will first solve all of the smaller subproblems its solution depends upon and store their values in extra memory.

Problem 1 - Climbing Stairs Problem.

Key Point - An excellent problem to learn problem-solving using DP and application of the Fibonacci series in problem-solving. One can find a lot of similar DP problems asked during the coding interview.

Problem Statement - There is a staircase of n steps and you can climb either 1 or 2 steps at a time. We need to count and return the total number of unique ways to reach the n'th stair. The order of steps matters.

Brute force approach using recursion.

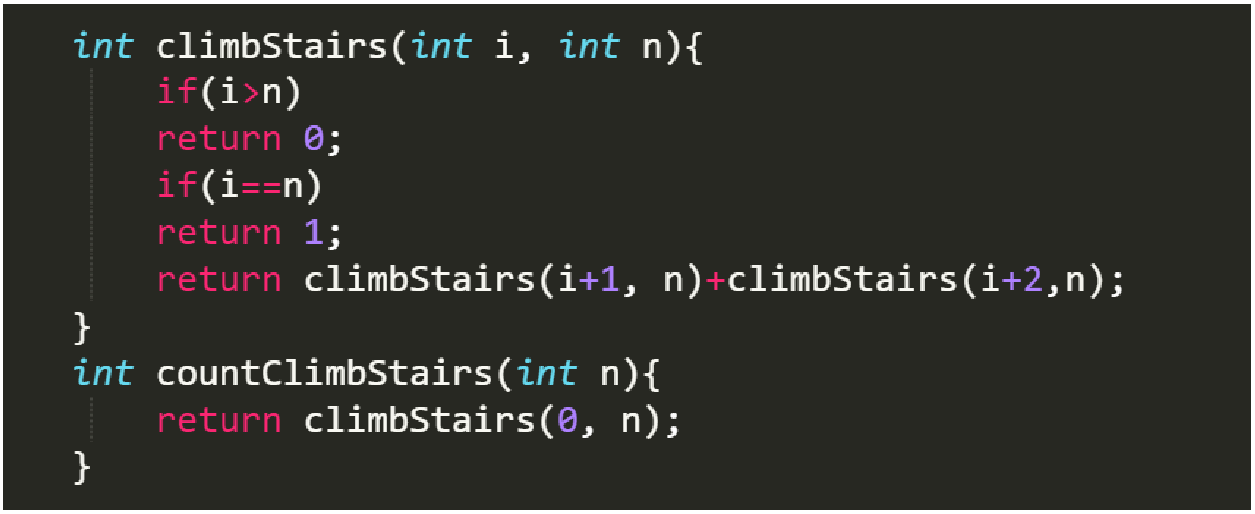
Solution -
This is a counting problem where we need to count all unique ways to reach the nth stair, but the critical question is: How do we count all possibilities?

We have two different choices at the start: Either climb 1st stair or climb 2nd stair because we can only jum 1 or 2 steps at a time.
If we take one step from ground, then the smaller sub-problem = Climbing nth stair from 1st stair.
If we take two-step from ground, then the smaller sub-problem = Climbing nth stair from 2nd stair.

So, we can solve given counting problem recursively by adding the solutions of sub-problems:
climbStairs(0, n) = climbStairs(1, n) + climbStairs(2, n).

We can also think with another analogy: We can reach the nth stair from (n-1)th stair and (n-2)th stair by taking and 2 steps respectively.
So total number of ways to reach nth stair = Total number of ways to reach(n-1)th stair + Total number of ways to reach (n-2)th stair.
climbStairs(n) = climbStairs(n-1) + climbeStaris(n-2).

So Let's Code!!🧑‍💻🧑‍💻🧑‍💻



```
int climbStairs(int i, int n){
    if(i>n)
    return 0;
    if(i==n)
    return 1;
    return climbStairs(i+1, n)+climbStairs(i+2,n);
}
int countClimbStairs(int n){
    return climbStairs(0, n);
}
```

Time complexity analysis -
Input size of the original problem = n, Input size of the 1st subproblem = n-1, Input size of the 2nd sub-problem = n-2.

So we write the recurrence relation of the time complexity in terms of the input size i.e T(n) = T(n-1) + T(n-2),
where T(1)=1 and T(2)=2.

The above recurrence relation is similar to the recurrence relation of the Fibonacci sequence.
Time complexity = O(2^n).

There is another rough analogy: We have two choices with each stair (excluding 0th and nth) i.e either we include it in the solution or exclude it.
Total number of possibilities is ~ 2^n, so time complexity = O(2^n).

Let's optimize the solution, using the Bottom-up approach of Dynamic Programming.

Since overlapping subproblems are present in this scenario, we can optimize the solution using dynamic programming.
Here we solve each sub-problem once, store the results in extra memory, and retrieve their results in O(1) instead of calculating again.
We need to take care of the following steps to solve the problem using a bottom-up approach.

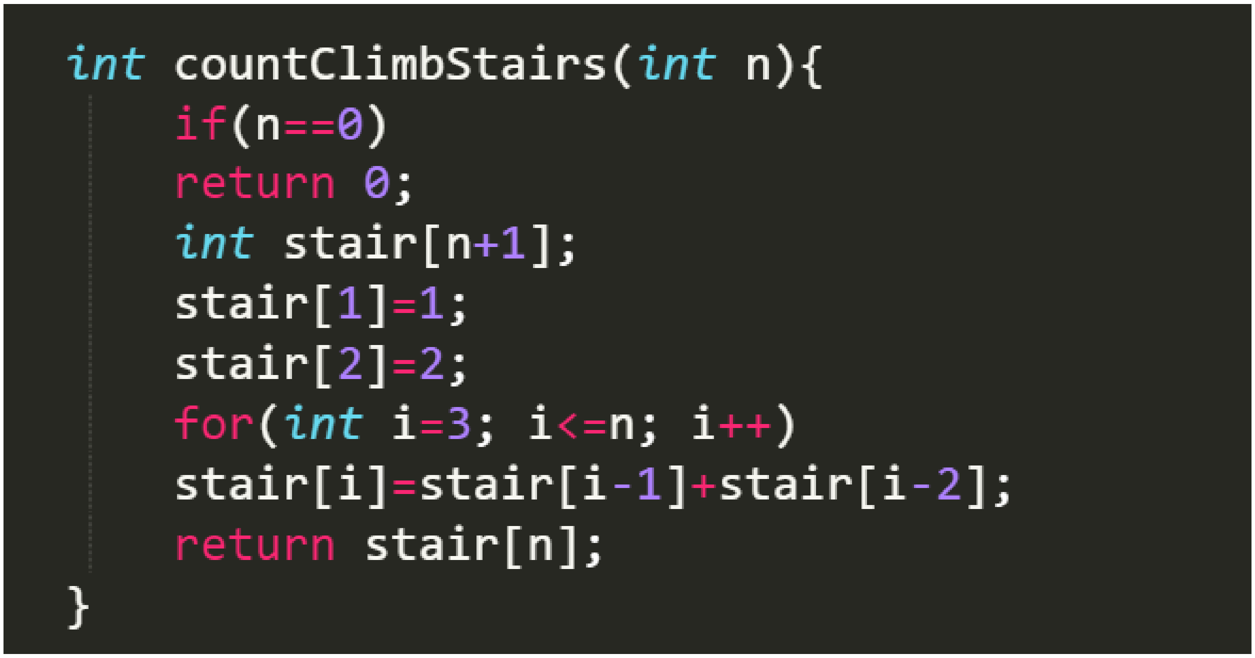
- **Table size and structure**: There are (n+1) different subproblems with input sizes from 0 to n. So we need to take a stair[] table of size n+1 to store the solution to these sub-problems.

- **Table Initialisation**: Before building a solution in a bottom-up manner, we initialize the table by using the basic version of the solution. We can get this idea from the base case of a recursive solution. Stair[1]=1, Stair[2]=2.

- **Iterative structure to fill the table**: Now we define an iterative structure to fill tables or build solutions in a bottom-up manner.
  We can get this idea from the recursive structure of a recursive solution. Here is an insight: Recursion is coming top-down and solving larger problems via solution of smaller sub-problems. In similar but reverse fashion, we need to go bottom-up and combine smaller problems to build solutions for the larger problem.

The total number of ways to reach (i)th stair = Total number of ways to reach (i-1)th stair + Total number of ways to reach (i-2)th stair.
Stair[i] = Stair[i-1] + Stair[i-2].

Returning the final solution: After storing results in the table, our final solution gets stored at the last index of the array i.e. return Stair[n].



```
int countClimbStairs(int n){
    if(n==0)
    return 0;
    int stair[n+1];
    stair[1]=1;
    stair[2]=2;
    for(int i=3; i<=n; i++)
    stair[i]=stair[i-1]+stair[i-2];
    return stair[n];
}
```

Time and space complexity analysis -

We are running a single loop and doing constant operations at each step of the iteration, so time complexity = O(n).

Space complexity = O(n), because we are using extra space of size n+1.

Problem 2: 01 Knapsack.

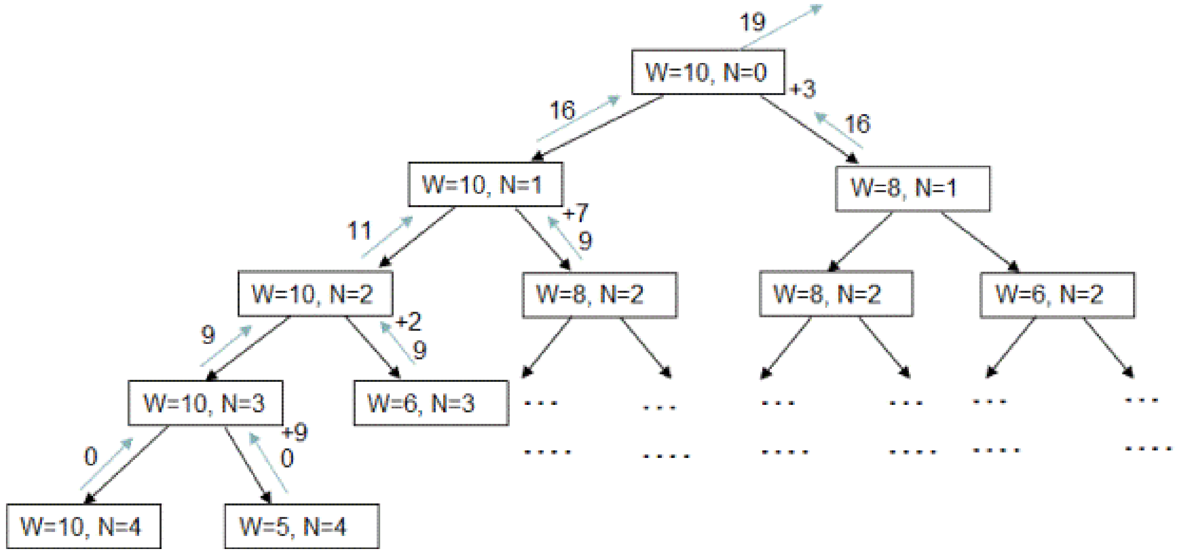
This is the core definition of dynamic programming. Understanding this problem is super important, so pay good attention.

Every problem in general, and all DP questions have a CHOICE at every step.

We have a weights array and a values array, where we want to choose those values which will return us the maximum weight sum (within the limit). There is a max weight given, which we have to take care of. Just from the first glance, I see that maxWeight will help us with the base case. At every step, we have 2 CHOICES.

- Include the value: Take value from values[index] + move ahead
- Exclude the value: Just move ahead

It's also important to think about what your recursive function would look like.
What values to pass, how will we iterate over the array, how will we use the base case through those arguments.



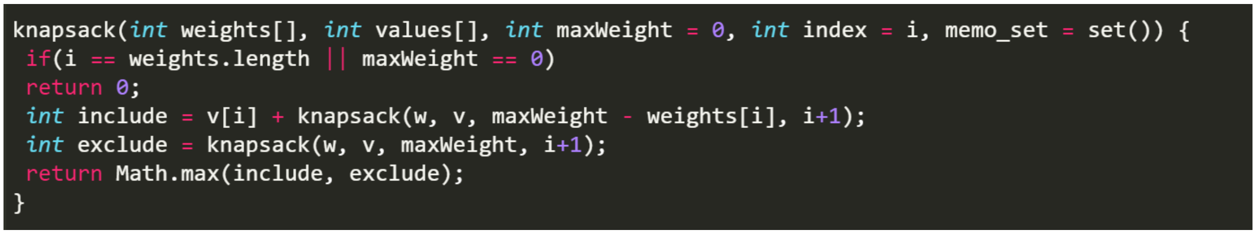
Recursion tree for 0-1 Knapsack problem

Thinking about the arguments, a good recursive function would be passing in the weights, values, index, and the remaining weight?

That way remaining_weight == 0 can be our base case. You can absolutely have other recursive functions with different arguments, it’s about making things easier.

```
/*include the ith item*/
int include = v[i] + knapsack(w, v, maxWeight - weights[i], i+1);

/* don't include*/
int exclude = knapsack(weights, values, maxWeight, i+1);
```

We think of the base case now.
A straightforward one looks like maxWeight === 0, which is also the REMAINING weight as we’re subtracting the weight every time we’re iterating with the included item.
The second one and the most usual one is when you reach the end of the array, so index === weights.length.
Can also be values.length as they’re the same.



```
knapsack(int weights[], int values[], int maxWeight = 0, int index = i, memo_set = set()) {
 if(i == weights.length || maxWeight == 0)
 return 0;
 int include = v[i] + knapsack(w, v, maxWeight - weights[i], i+1);
 int exclude = knapsack(w, v, maxWeight, i+1);
 return Math.max(include, exclude);
}
```

This is just a pseudo code.

There is a problem here, we’re doing a lot of repetitive work here, do you see it?

No?
Go wash your eyes. Just Kidding……try to find out.

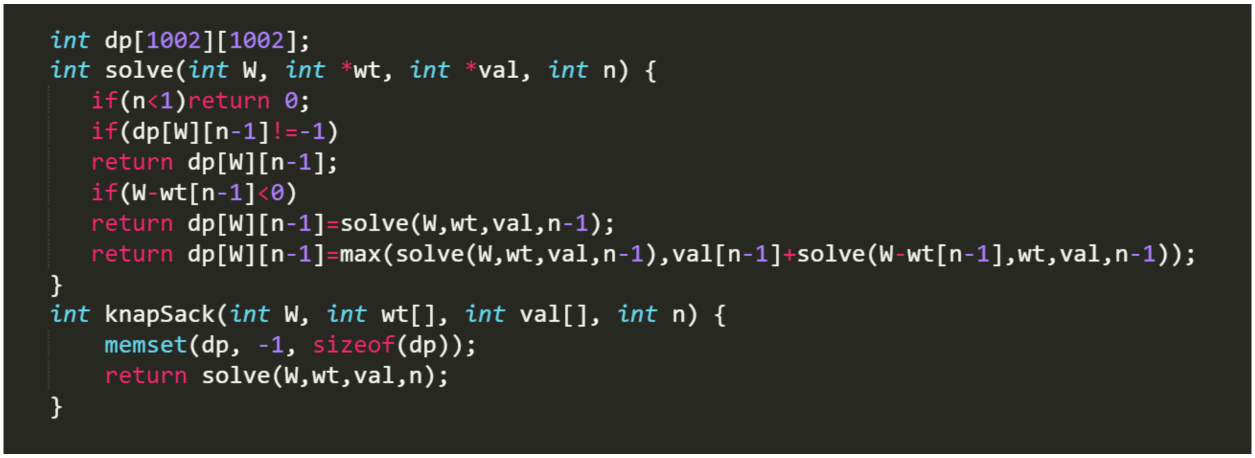
We’re re-calculating a lot of states -> where the value maxWeight - weights[i] value is something.
For example 5 is 8-3 but it’s also 9-4.
So we don’t want to do this, how can we stop this?

MEMOIZATION Simply store the max value and return it with the base case. You can think of memoization as your SECOND base case.

We set the key and max value in the set, and then use that in the base case to return when that condition is reached.

This is the recursive approach and once you’ve understood how the basis of this works, you can go to the iterative version. It’s very important to solve and understand it recursively before moving forward.

Let’s Code…..



```
int dp[1002][1002];
int solve(int W, int *wt, int *val, int n) {
   if(n<1)return 0;
   if(dp[W][n-1]!=-1)
   return dp[W][n-1];
   if(W-wt[n-1]<0)
   return dp[W][n-1]=solve(W,wt,val,n-1);
   return dp[W][n-1]=max(solve(W,wt,val,n-1),val[n-1]+solve(W-wt[n-1],wt,val,n-1));
}
int knapSack(int W, int wt[], int val[], int n) {
    memset(dp, -1, sizeof(dp));
    return solve(W,wt,val,n);
}
```

That’s all about DP from my side……….I hope you understand Dp very well.

Now you will not be afraid of any DP problem.

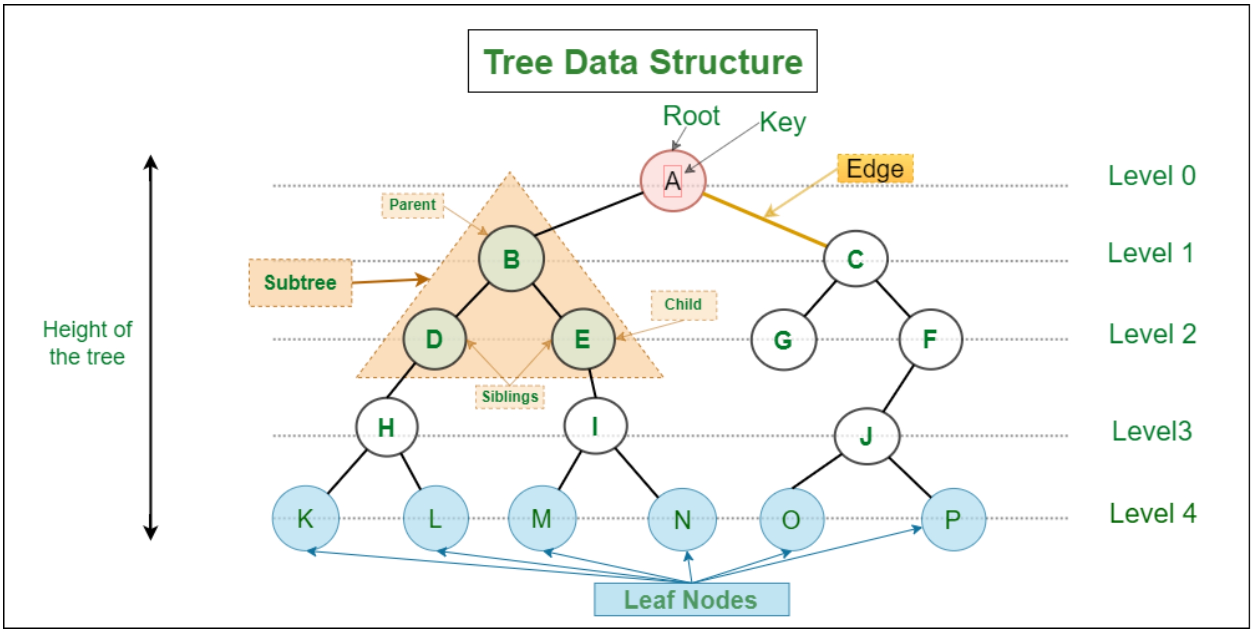
Ok……so now let’s crack the next Topice…….

“Trees”

I love trees, but actual ones - not these.
Just kidding, I love all data structures.

Let’s discuss trees. They’re tree-like structures (wow) where we can store different things, for different reasons, and then use them to our advantage.

Here’s a nice depiction of how the actually look:



Recursion is a great way to solve a lot of tree problems, but the iterative ones actually bring out the beauty of them.

Making a stack and queue, adding and popping things from that, exploring children, and repeating this would definitely make sure you understand it completely.

You should be seeing this visually in your head, when you do it iteratively.

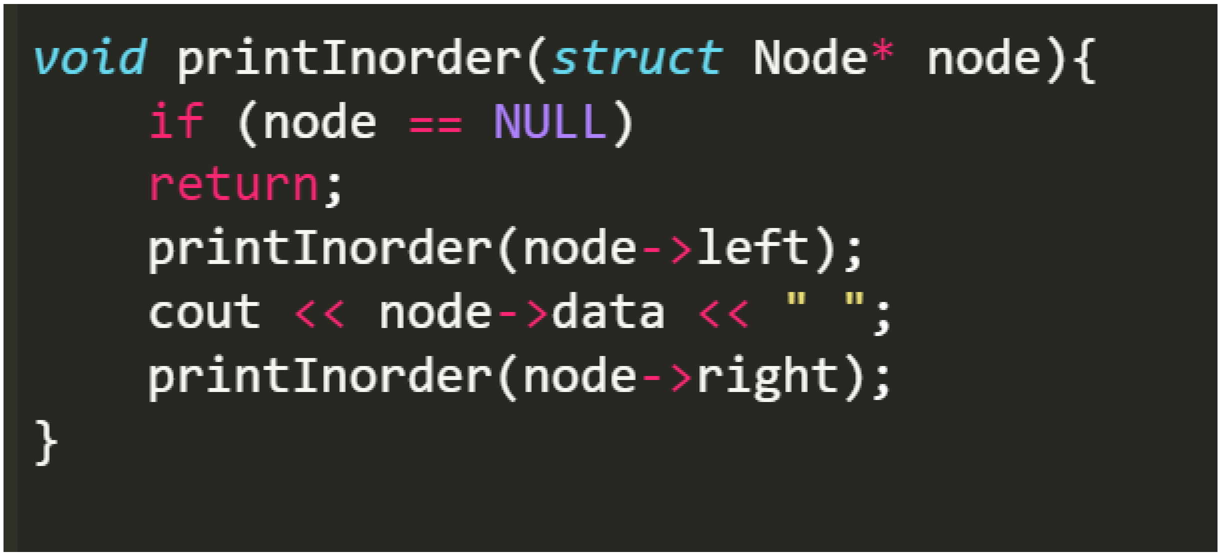
Pattern: Traversals

There are 3 major ways to traverse a tree and some other weird ones: let's discuss them all.
The most famous ones are **pre, in, and post** - order traversals.
Remember, in traversals -> it's not the left or right node (but the subtree as a whole).

Inorder traversal

- Traverse the left subtree, i.e., call Inorder(left->subtree)
- Visit the root.
- Traverse the right subtree, i.e., call Inorder(right->subtree)

Uses of Inorder Traversal:
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.
Example: In order traversal for the above-given figure is 4 2 5 1 3.



```
void printInorder(struct Node* node){
    if (node == NULL)
    return;
    printInorder(node->left);
    cout << node->data << " ";
    printInorder(node->right);
}
```

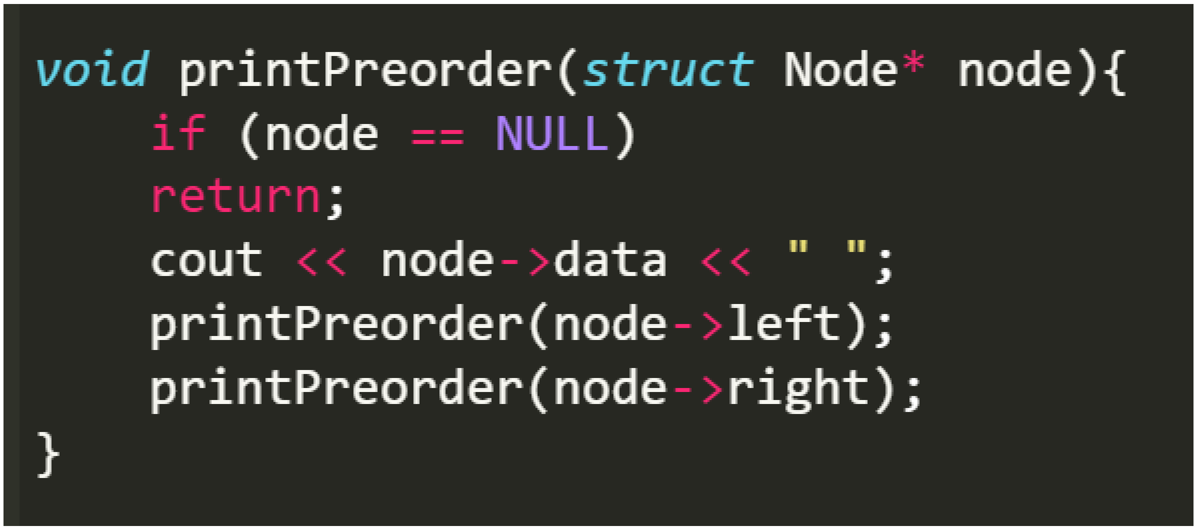
Time Complexity: O(N)
Auxiliary Space: If we don’t consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

Preorder Traversal: -

Algorithm Preorder(tree)

- Visit the root.
- Traverse the left subtree, i.e., call Preorder(left->subtree)
- Traverse the right subtree, i.e., call Preorder(right->subtree).

Uses of Preorder:
Preorder traversal is used to create a copy of the tree.
Preorder traversal is also used to get prefix expressions on an expression tree.
Example: Preorder traversal for the above-given figure is 1 2 4 5 3.



```
void printPreorder(struct Node* node){
    if (node == NULL)
    return;
    cout << node->data << " ";
    printPreorder(node->left);
    printPreorder(node->right);
}
```

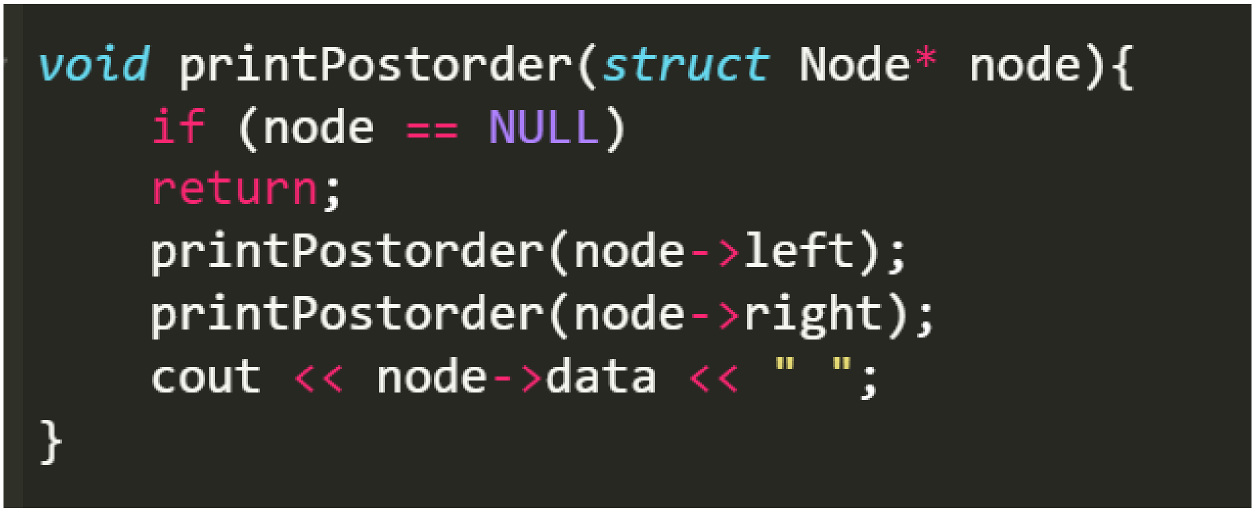
Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

PostOrder Traversal: -

Algorithm Postorder(tree)

- Traverse the left subtree, i.e., call Postorder(left->subtree)
- Traverse the right subtree, i.e., call Postorder(right->subtree)
- Visit the root.

Postorder traversal is used to delete the tree.Postorder traversal is also useful to get the postfix expression of an expression tree
Example: Postorder traversal for the above-given figure is 4 5 2 3 1.



```
void printPostorder(struct Node* node){
    if (node == NULL)
    return;
    printPostorder(node->left);
    printPostorder(node->right);
    cout << node->data << " ";
}
```

Time Complexity: O(N)
Auxiliary Space: If we don’t consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

Problem 1 - Height of a Binary Tree.

Given a binary tree, write a program to find its height.
In other words, we are given a binary tree and we need to calculate the maximum depth of the binary tree.

The height or maximum depth of a binary tree is the total number of edges on the longest path from the root node to the leaf node.
In other words, the height of a binary tree is equal to the maximum number of edges from the root to the most distant leaf node.
The height of an empty tree or tree with one node is 0.

Recursive approach: Using DFS or Post-order traversal

Solution idea -

A binary tree is a recursive structure where we can solve the problem using the solution of two smaller subproblems: left subtree and right subtree.

So how can we apply this idea to find the height of a binary tree?

Let's think!

Based on the definition, for finding the height, we should know the path length of the root to the deepest leaf node, which can be either present in the left-subtree or right sub-tree.

In other words, the longest path from the root to the deepest leaf can go through either the left or right child of the root. So the best idea would be to calculate the height of the left and right subtree recursively.

Whichever sub-tree has the maximum height, we take that value and add 1 to get the height of the tree.

Here is a clear insight:

The height of the tree = The longest path from the root node to the leaf node = 1 + max (longest path from the left-child to the leaf node, longest path from the right-child to the leaf node) = 1 + max (height of the left sub- tree, height of the right subtree)

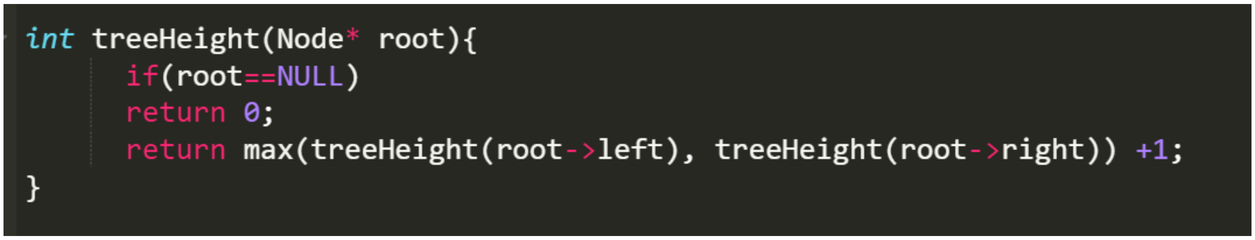
So if we know the height of the left and right sub-tree then we can easily find the height of the tree.
But one idea would be clear: Before calculating the height of the tree, we must know the height of the left and right sub-tree.
So we need to traverse tree using post-order traversal. Think!
Solution steps
Suppose we are using the function int treeHeight(TreeNode root).
If the tree is empty or tree has only one node, we return 0. This would be our base case.
Otherwise, we traverse the tree in a post-order fashion and call the same function for the left and right child to calculate the height of the left and right sub-tree.

```
int leftHeight  = treeHeight(root->left);
int rightHeight = treeHeight(root->right);’
```

Now after calculating the height of the left and right subtree, we take the maximum height among both and return the value by adding one to it.

```
return max(leftHeight, rightHeight)+1;
```

Let’s combine the code…



```
int treeHeight(Node* root){
        if(root==NULL)
        return 0;
        return max(treeHeight(root->left), treeHeight(root->right)) +1;
}
```

Solution analysis

We are doing post-order traversal for finding the height of the tree. So time complexity = O(n). Space complexity is equal to the size of the recursion call stack, which is equal to the height of the tree.

- The best-case scenario occurs when the tree is balanced, or the tree's height is O(logn). So best case space complexity = O(logn)
- The worst-case scenario occurs when the tree is highly unbalanced or skewed, i.e There is only one node at each level. In this case, the height of the tree is O(n). So worst-case space complexity = O(n)

“Graphs”

A lot of graph problems are covered by DFS, BFS, topo sort in general -> but we’re going to do a general overview of everything related to graphs.

There are other algorithms like Djikstra’s, MST, and others - which are covered in the greedy algorithms section.
A lot of graph problems are synced with other types = dynamic programming, trees, DFS, BFS, topo sort, and much more. You can think of those topics sort of coming under the umbrella of graph theory sometimes.

Problem 1: - Detect a cycle in Directed Graph.
How to approach this problem……..???
A cycle involves at least 2 nodes. The basic intuition for cycle detection is to check whether a node is reachable when we are processing its neighbors and also its neighbors’ neighbors, and so on. Let’s take the cycle in Example : 0 -> 1 -> 2 -> 3 – > 4 -> 0.

We can see that 0 is reachable through 1, 2, 3, and 4. We can use the normal DFS traversal with some modifications to check for cycles.

Approach:

Usually, we need a separate array for DFS traversal, which has information on whether a particular node has been visited or not.

For this cycle detection, we need one more array which has information on whether we are exploring one's neighbor or not.

That is, for example in Example 2, in DFS, we go from Nodes 0 to 1 to 2. Only when we are finished visiting node 2's neighbors, we can visit node 1's other neighbors. Also, only when we are finished visiting node 1's neighbors, we can go to other neighbors of 0.

Within that time if any neighbors of nodes 1 and 2, have an edge to 0, then we can say that there is a cycle. So, we need a second array to tell whether, at any point in time, we are processing a node's neighbors. Once we finish processing the node's neighbors, we can update the array.

So, in our algorithm, we are starting with node 0. This is an important step. Before going through its neighbors, we have to first set 1 to its index in the visited array. And then we have set 1 to the index in the special array(in this case: dfsVis[]), which just says that we are going to process its neighbors. As we do for DFS traversal, we check whether we have already visited a node (which is currently a neighbor of another node).

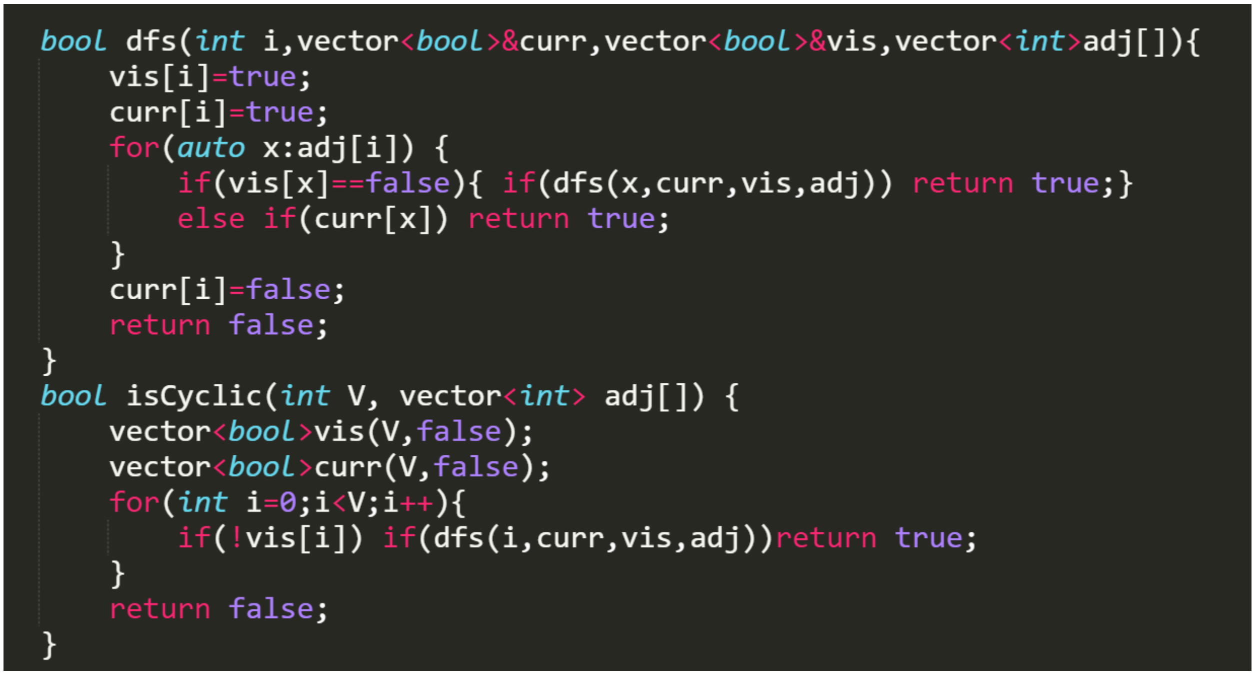
If it is already visited (this is the heart of this algorithm), then we check whether we are processing its neighbors (by checking dfsVis []). If so, then yes, we have found a cycle.

Because, we haven’t finished processing the node’s neighbor (variable “node’s” neighbor, variable “neighbor”), but we found an edge from the variable node to a variable neighbor. There is also another important thing to do.

After traversing a node’s neighbors, we should inform the dfsVis array, by setting it to 0, which means we have finished processing its neighbors.

That is our whole algorithm. If our function returns true, then we have a cycle, if it returns false, then we don’t have any cycle.

So let’s Code!!🧑‍💻🧑‍💻🧑‍💻



```
bool dfs(int i,vector<bool>&curr,vector<bool>&vis,vector<int>adj[]){
    vis[i]=true;
    curr[i]=true;
    for(auto x:adj[i]) {
        if(vis[x]==false){ if(dfs(x,curr,vis,adj)) return true;}
        else if(curr[x]) return true;
    }
    curr[i]=false;
    return false;
}
bool isCyclic(int V, vector<int> adj[]) {
    vector<bool>vis(V,false);
    vector<bool>curr(V,false);
    for(int i=0;i<V;i++){
        if(!vis[i]) if(dfs(i,curr,vis,adj))return true;
    }
    return false;
}
```

Time Complexity: O(V + E), since in its whole, it is a DFS implementation, V – vertices; E – edges;

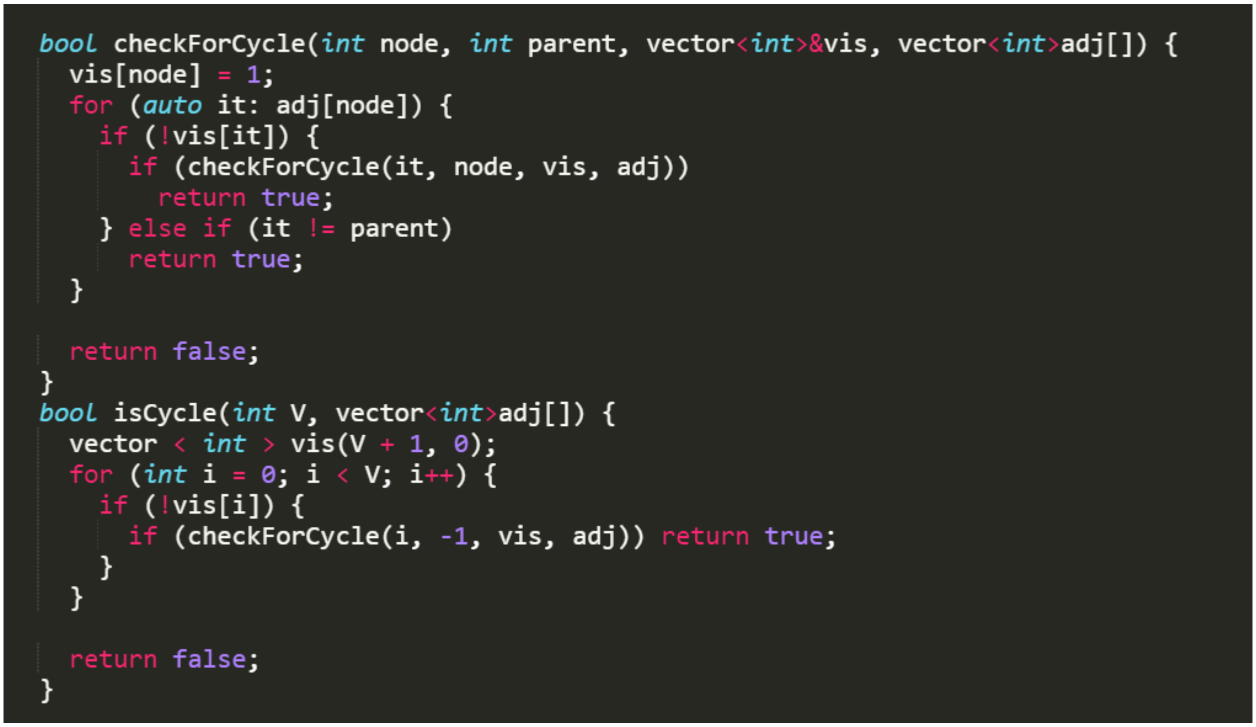
Space Complexity: O(V), because, apart from the graph, we have 2 arrays of size V, to store the required information.

Problem 2 - Detects a cycle in an Undirected Graph.

Approach: Run a for loop from the first node to the last node and check if the node is visited. If it is not visited then call the function recursively which goes into the depth as known as DFS search and if you find a cycle you can say that there is a cycle in the graph.

- Basically calling the isCyclic function with number of nodes and passing the graph
  Traversing from 1 to number of nodes and checking for every node if it is unvisited
- If the node is unvisited then call a function checkForCycle, that checks if there is a cycle and returns true if there is a cycle.
- Now the function checkForCycle has the node and previous of the parent node. It will also have the visited array and the graph that has adjacency list
- Mark it as visited and then traverse for its adjacency list using a for loop.

- **Calling DFS traversal if that node is unvisited call recursive function that checks if its a cycle and returns true**
- **If the previously visited node and it is not equal to the parent we can say there is cycle again and will return true**
- **Now if you have traveled for all adjacent nodes and all the DSF have been called and it never returned true that means we have done the DSF call entirely and now we can return false, that means there is no DSF cycle.**

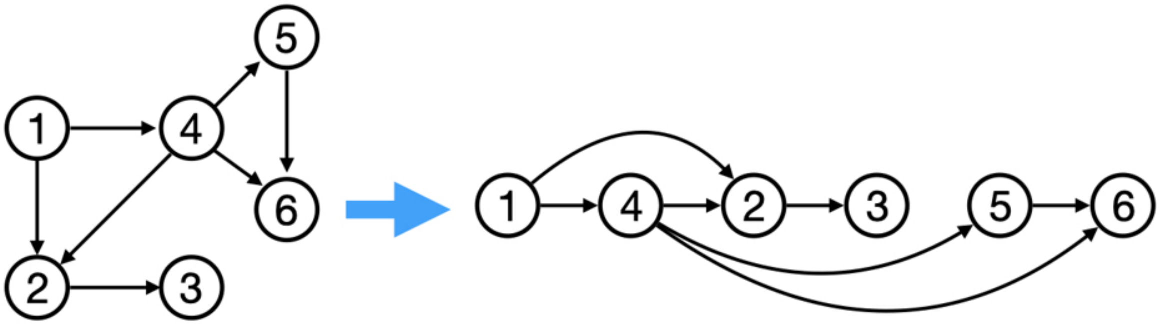


```
bool checkForCycle(int node, int parent, vector<int>&vis, vector<int>adj[]) {
  vis[node] = 1;
  for (auto it: adj[node]) {
    if (!vis[it]) {
      if (checkForCycle(it, node, vis, adj))
        return true;
    } else if (it != parent)
      return true;
  }

  return false;
}
bool isCycle(int V, vector<int>adj[]) {
  vector < int > vis(V + 1, 0);
  for (int i = 0; i < V; i++) {
    if (!vis[i]) {
      if (checkForCycle(i, -1, vis, adj)) return true;
    }
  }

  return false;
}
```

It’s not end here…………..i have discussed some more topics buddy. Let’s see. that one……..

Problem 3: - Topological Sort.

What is Topological sorting…………..???
It is basically Linear ordering of vertices such that there is an edge u->v, where u appears before v in the ordering.



Topological Sorting is applicable only for DAG(Directed Acyclic Graph). Why is it so?

Because of the following reasons:

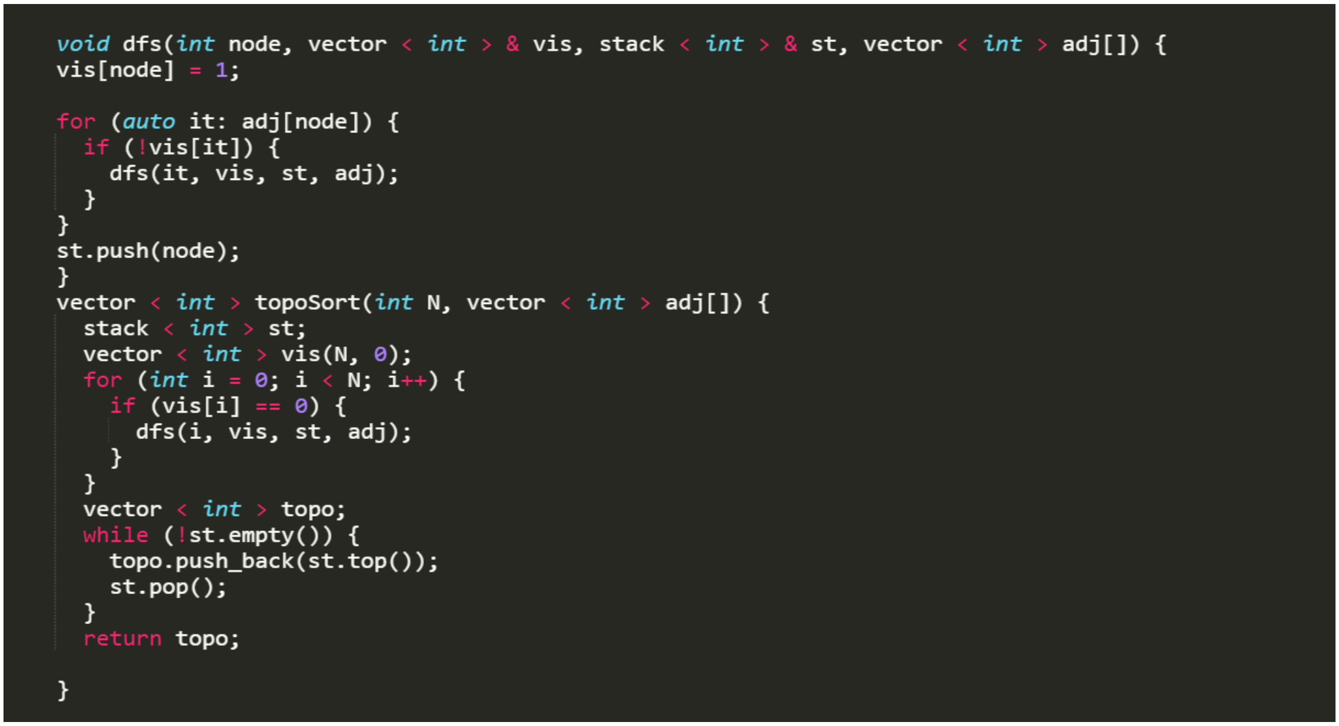
- **For Undirected graphs ,only u->v is not applicable . It cannot be sure whether the edge is between u to v or v to u (u-v) .**
- **In a cyclic graph there will always be a dependency factor . You cannot make sure that you can have linear ordering of vertices.**

Finally, now you have a clear understanding of what Topological Sorting is. We will be using the DFS(Depth First Search) method to solve the problem. What we will be doing is for each vertex we will explore its adjacent vertex. After exploring, we will store the current vertex in a data structure to maintain Topo Sort.

Topological sort in simple words - DFS with stack.

Dfs call will go from u to v. The 1st dfs (v) will get over first and then dfs(u). Here we are making sure that if u->v, then we will first push v into the stack and then u will be pushed. This is how Topological order is maintained in the Stack.

Once your dfs is finished, then you just have to pop all the elements and insert these elements into a vector and that vector will be your topological sort.



```
void dfs(int node, vector < int > & vis, stack < int > & st, vector < int > adj[]) {
vis[node] = 1;

for (auto it: adj[node]) {
  if (!vis[it]) {
    dfs(it, vis, st, adj);
  }
}
st.push(node);
}
vector < int > topoSort(int N, vector < int > adj[]) {
  stack < int > st;
  vector < int > vis(N, 0);
  for (int i = 0; i < N; i++) {
    if (vis[i] == 0) {
      dfs(i, vis, st, adj);
    }
  }
  vector < int > topo;
  while (!st.empty()) {
    topo.push_back(st.top());
    st.pop();
  }
  return topo;

}
```

That’s all............from my side..............i hope you loved this journey.

And if you need any further help, then please feel free to ask me. I will be happy to help you.

Thanks Buddy........and Bye Bye........

Are ruko jra..........abhi m or bhi questions.......leke aaunga.......ok.