Complete Data structures and Algorithms Guide

Resources, Notes, Questions, Solutions

JULY
2021

This document provides a complete guide to understanding data structures and algorithms in computer science. It contains a collection of information compiled by people who have successfully completed interviews at FAANG companies as well as a useful information, tips, graphics, etc. to help better understand the subject matter and be a better coder or to ace that interview.

As with all 30DaysCoding resources, this guide is provided to you with the mission of making the world's resources more accessible to all.

Enjoy!

Table of Contents

The document is divided as follows.

Complete Data structures and Algorithms Roadmap

Resources, Notes, Questions, Solutions

We've covered all the amazing data structures and algorithms that you will ever need to study to get that next opportunity. Hopefully, this will make you a better problem solver, a better developer, and help you ace your next technical coding interviews. If you have any questions along the way, feel free to reach out to us on 30dayscoding@gmail.com.

Our Aim

- Our aim is to help you become a better problem solver by gaining knowledge of different data structures, algorithms, and patterns
- We want you to understand the concepts, visualize what's going on, and only then move forward with more questions
- Most phone interviews require you to be a good communicator and explain your approach even before you write the solution. So it's important to understand the core concepts and then work on extra stuff.

□ □ □ There are thousands of questions which you can solve out there, but computer science and coding is much more than just doing data structures. **Building and developing** something is the core of computer science, so if you're actually interested - then most of your time should go in learning new frameworks and building stuff. Another important aspect of coding and life in general is to **explore**! The more you explore -> the closer you get to your interests, so keep exploring and learning. Good luck!

Practice

- Practicing 150-200 questions will make you confident enough to approach any new problem. This is just solving only 2 questions for 75 days, which is not a lot if you think about it!
- Being consistent is the key. Finish this guide in 75-90 days. Don't rush it from today, take your time, revisit topics after a while, read and watch a lot of videos, and eventually be comfortable with problem solving!
- Enjoy the process and start today. I'm sure you'll do great. Have fun.

Arrays

Introduction

 Informally, an array is a list of things. It doesn't matter what the things are; they can be numbers, words, apple trees, or other arrays. Each thing in an array is called an *item* or *element.* Usually, arrays are enclosed in brackets with each item separated by commas, like this: [1, 2, 3]. The elements of [1, 2, 3] are 1, 2, and 3.

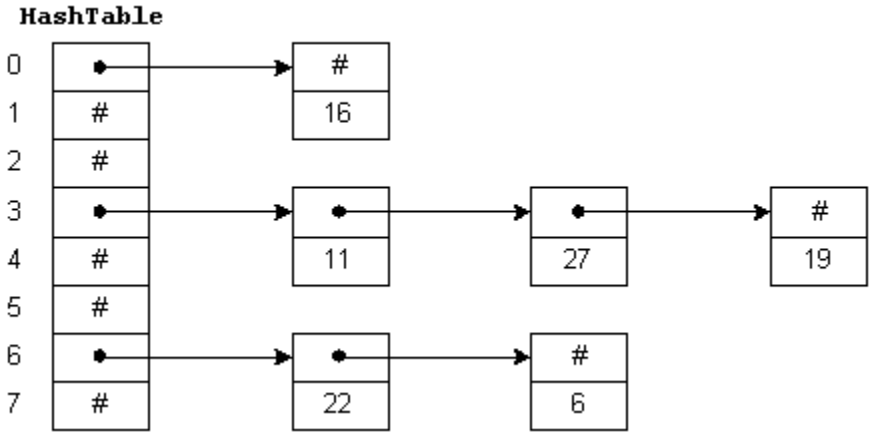
- Introduction to Arrays
- https://www.cs.cmu.edu/~15122/handouts/03-arrays.pdf
- An Overview of Arrays and Memory (Data Structures & Algorithms #2)
- What is an Array? - Processing Tutorial

Arrays are used with all different types of data structures to solve different problems, so it's kind of hard to come up with array questions with just an array logic. Let's discuss some of the most famous patterns which concern arrays most of the time.

2D matrices are also arrays and are very commonly asked in interviews. A lot of graph, DP, and search based questions involve the use of a 2D matrix and it's important to understand the core concepts there. We've discussed the common patterns in each section below so make sure to check that out.

Hash maps, tables

 A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. In other words, we can store anything in the form of key value pairs.
Example: map<string, string>, means that this is a hashmap where we store string key and value pairs.



Resources

- Hashmap and Map Powerful Guide
- Data Structure and Algorithms - Hash Table
- Leetcode discuss: Hashtable implementation

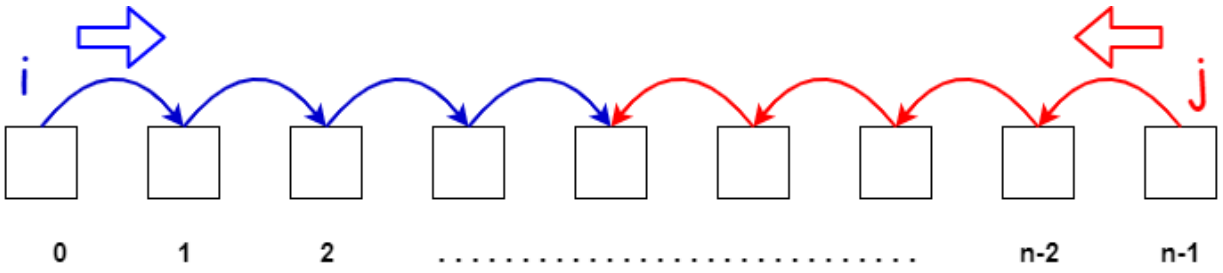
Questions

- 1. Two Sum
- 771. Jewels and Stones
- Leetcode : How Many Numbers Are Smaller Than the Current Number
- Partition Labels

---

2 Pointers

 For questions where we're trying to find a subset, set of elements, or something in a sorted array -> 2 pointers approach is super useful.



Some common questions with this approach are concerned with splitting something or finding something in the middle, eg: middle element of the linked list. This is something which you'll recognize instantly after solving some questions on it, so just try to see the template and start solving.

Here's a general code template for solving a 2 pointer approach. We move from the left and right with different conditions until there's something we want to find.

```java
/* General two pointer problem solution */
public boolean twoSumProblem(int A[], int N, int X)
{
    // represents first pointer
    int left = 0;

    // represents second pointer
    int right = N - 1;

    while (left < right) {
        // question condition match
        if(){
            // do something
            return true
        }
        // first wrong condition
        else if(){
            // close in the array from left
            left+=1;
        }
        // second wrong condition
        else{
            // close in the array from right
            right-=1;
        }
    }
    return false;
}
```

Problem 1: Remove duplicates

https://leetcode.com/problems/remove-duplicates-from-sorted-array/

 We have a value given, and we have to remove the occurrences of ‘value’ in place. The array is sorted and we have to return an integer (important information)

Damn, how do we do this? No clue.

Kidding, let’s discuss. One brute force way is to have a separate array, iterate over the original array and then add the items other than value to the new array. Now just return the new array. But we have to do this in place, so we can’t have an additional thing. How to do that?

Let's start from the beginning, if the number is something other than the 'value', let's just bring that to the beginning? And then finally return the pointer.
Think of this -> we want to shift the elements behind if they don't match the value given.

```
int removeElement(int A[], int elem) {
    int pointer = 0;
    for(int i=0; i<n; i++) {
      if(A[i]!=elem) {
            A[pointer++] = A[i];
          }
    }
    return pointer;
}
```

Part II
What if we don't have a value and just want to remove the duplicates and return the index.
We would still have 2 pointers, test if slow != fast -> move the slow pointer forward, and change the nums[slow] to the fast one -> basically pushing that element back.

```
def removeDuplicates(self, nums: List[int]) -> int:
    if len(nums) ==0 : return 0
    slow = 0
    for fast in range(1,len(nums)):
        if nums[slow] != nums[fast]:
            slow += 1
            nums[slow] = nums[fast]
    return tail+1
```

Problem 2: Two Sum + Sorted

 If we want to find 2 indices which sum up to a target and the array is sorted, we can start from left and right with 2 pointers, and move them according to the sum at every time.
Eventually we will find the target from those 2 indices, or just return -1 if we don't.

However, the logic would be more complex if the array is not sorted. We can simply store the elements in a hashmap as we go, and eventually return when we find target-nums[i] in the array as we're going forward.

```java
boolean pairSum(int A[], int N, int X)
{
    int i = 0;
    int j = N - 1;

    while (i < j) {
        if (A[i] + A[j] == X)
            return true;

        else if (A[i] + A[j] < X)
            i++;
        else
            j--;
    }
    return false;
}
```

Read □

- Article: 2 Pointer Technique
- Hands-on-Algorithmic-Problem-Solving: 2 Pointers

Videos □

- How to Use the Two Pointer Technique
- Two Pointer | Is Subsequence | LeetCode 392.

Questions □

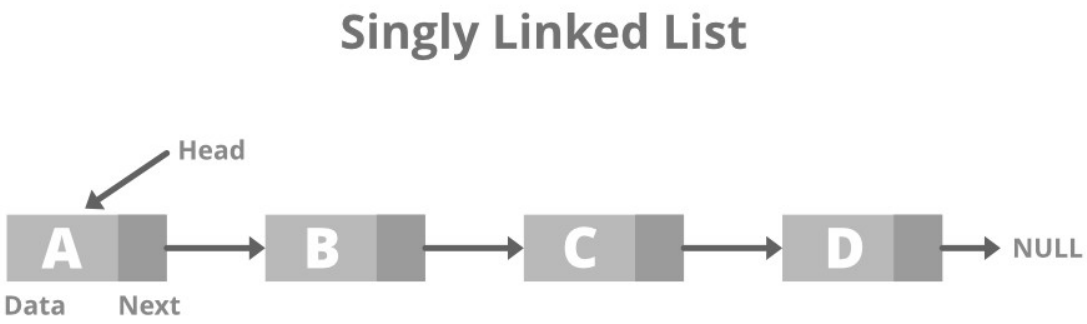
- Middle of the Linked List
- 922. Sort Array By Parity II
- Reverse String
- Valid Palindrome
- E26. Remove Duplicates from Sorted Array
- 75. Sort Colors
- 11. Container With Most Water

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: Linked list methods

Here's how a linked list looks like: Linked list: Methods

We have the basic functions of insert, delete, search, etc which basically depend on 2 simple conditions:

- We have to iterate to find the node
- And the only way we can go is forward: *node.next*

You might not see a Linked list ever in your life when you develop real things, but it's still nice to know something that exists. Along with that, there are certain questions based around linked lists, so it's important to understand those as well.

Problem 2: Linked List cycle

 We want to find if there's a cycle in a linked list. The first thing which comes to mind is -> how the F do we do this. Just kidding -> it's on the lines of -> maybe if we visit a node again, then we would find a cycle

How can we find if we visited the node again? Maybe store the nodes as you're iterating and then see if you find the node in the set again.

Using a set:

```
public boolean hasCycle(ListNode head) {
    Set<ListNode> set = new HashSet<>();
    while(head!=null){
        if(set.contains(head)){
            return true;
        }
        set.add(head);
        head=head.next;
    }
    return false;
}
```

This solution is absolutely correct, but it requires you to have a separate set (space complexity), can we do something better? Think of something on the lines of 2 pointers. Can we have a slow and fast potiner where the fast tries to catch the slower one? If it can, we find a cycle, if not -> there's no cycle.

Using slow and fast pointers:

```
while(
      head!=null && slow.next !=null
      && fast.next!=null && fast.next.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if(slow == fast) return true;
    head = head.next;
}
```

Even simpler:

```
while(runner.next!=null && runner.next.next!=null) {
        walker = walker.next;
        runner = runner.next.next;
        if(walker==runner) return true;
}
```

Problem 3: Deleting a node

 We can simply delete a node by moving the pointer from the previous node to the next node. *prev.next = node.next*. The only simple catch is that we have to iterate to reach the 'node' and there is no instance given to us.

Part II

What if we want to delete a node without the head? 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.val = node.next.val;
node.next = node.next.next;
```

Problem 4: Merge sorted lists

Merge Two Sorted Lists

 We have 2 sorted lists and we want to merge them into one.

Does sorting tell you something? The element at the head would be the smallest.

Can we compare the heads every time and add those to the new list?

Ooooo, maybe yeah. Let's try comparing and then move the counter of the bigger element.

```
if l1.val < l2.val:
      cur.next = l1
      l1 = l1.next
```

Otherwise, we do this for the other node (because that's smaller)

```
else:
    cur.next = l2
    l2 = l2.next
```

What do we do once a list is done and the other one is left? Simply move the new linked list to the next pointer -> cur = cur.next and then add all the elements of the left over list.

```
cur.next = l1 or l2 # iterate over and add all the elements
return head (the temporary new list that we made)
```

Problem 5: Merge K Sorted lists:

Leetcode 23. Merge k Sorted Lists
 We have k lists and we want to merge all of those into a big one.

1. One simple way would be to compare every 2 lists, call this function, and keep doing until we have a bigger list. Any other shorter way?
2. We can be smart about it and add all the lists into one big array or list -> sort the array -> then put the elements back in a new linked list!
3. Or maybe we can use something called the priority Queue. We can add all the items to the queue, take them out one by one and then store them in a new list. It will behave like a normal queue or list, but save us a lot of time (complexity wise). Here's more about it. Here's the priority queue solution: here.

Read

- Linked list: Methods
- How I Taught Myself Linked Lists. Breaking down the definition of linked list
- Introduction to Linked List

Videos

- Data Structures: Linked Lists
- Interview Question: Nth-to-last Linked List Element

Questions

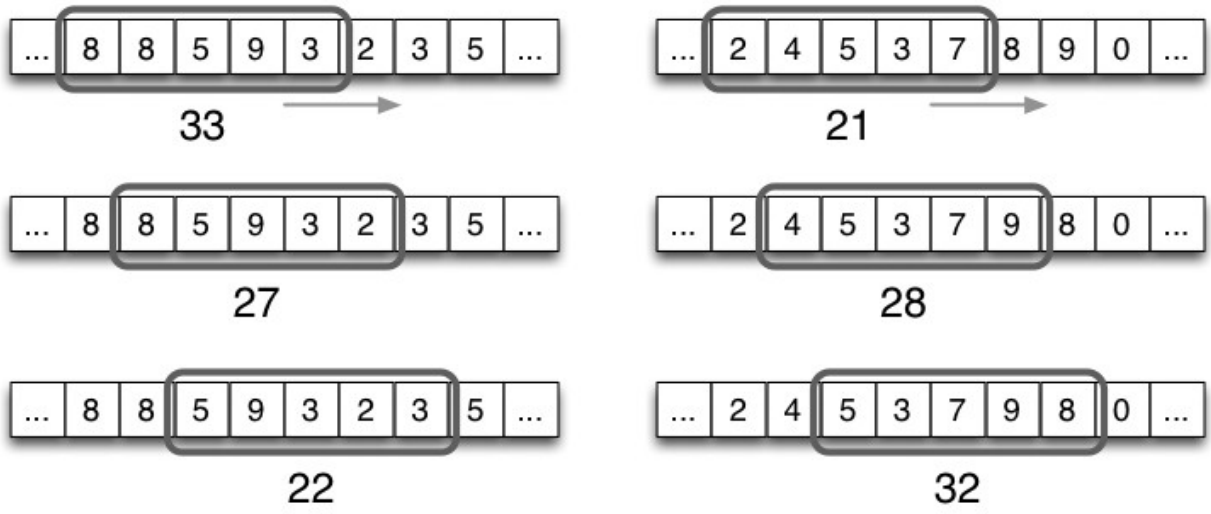
- 141. Linked List Cycle (Leetcode)
- Delete Node in a Linked List"
- 19. Remove Nth Node From End of List
- Merge Two Sorted Lists
- Palindrome Linked List

- 141. Linked List Cycle (Leetcode)
- Intersection of Two Linked Lists
- Remove Linked List Elements
- Middle of the Linked List
- lc 23. Merge k Sorted Lists
-

Sliding Window

Introduction

 This is super 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 right or left as we iterate through the string/array.



Sliding window is a 2 pointer problem where the front pointer explores the array and the back pointer closes in on the window. Here's an awesome visualization to understand it more: Dynamic Programming - Sliding Window

Problem 1: Max sum for consecutive k

 We have an array [1,2,3,2,4] and k=2, we want to return the max sum of the array with size 2.
Looking at this for the first time, I would think of a brute force way to calculate all the subarrays, find their sum, store the maximum, and return it.
However, that's very expensive. We don't really need to explore all the subarrays. Or, we can do that in an easier way (which is also cheaper): SLIDING WINDOW.

This is how 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.

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.

Here’s how the code looks.

```
int max_sum = 0;
int window_sum = 0;
/* calculate sum of 1st window */
for (int i = 0; i < k; i++)  {
      window_sum += arr[i];
}

/* Start the window from the left (k instead of 0)*/
for (int i = k; i < n; i++) {
    window_sum += arr[i] - arr[i-k];  // remove the left and add the right
    max_sum = max(max_sum, window_sum); // store the maximum
}
return max_sum;
```

 A must need article which covers more about this topic: Leetcode Pattern 2 | Sliding Windows for Strings | by csgator | Leetcode Patterns

Problem 2: Fruits into basket

904. Fruit Into Baskets

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)

```
while(end< tree.length){
    int a = tree[end];
    map.put(a, map.getOrDefault(a,0)+1);
    if(map.get(a)==1)counter++;
    end++;

    # something something
}
```

 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

```
while(counter>2){
    int temp = tree[start];
    map.put(temp, map.get(temp)-1); # remove elem count
    if(map.get(temp)==0)counter--; # decrease counter
    start++; # increment start
}
```

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.

Here's the combined code:

```java
public int totalFruit(int[] tree) {
    Map<Integer, Integer> map = new HashMap<>();
    int start=0, end=0, counter=0, len=0;

    while(end< tree.length){
        int a = tree[end];
        map.put(a, map.getOrDefault(a,0)+1);
        if(map.get(a)==1)counter++;
        end++;
        while(counter>2){
            int temp = tree[start];
            map.put(temp, map.get(temp)-1);
            if(map.get(temp)==0)counter--;
            start++;
        }
        len = Math.max(len, end-start);
    }
    return len;
}
```

□ □ □ A must need article which covers more about this topic: Leetcode Pattern 2 | Sliding Windows for Strings | by csgator | Leetcode Patterns

Read □

- Leetcode Pattern 2 | Sliding Windows for Strings | by csgator | Leetcode Patterns
- Sliding Window algorithm template to solve all the Leetcode substring search problems

Videos □

- Sliding Window Technique + 4 Questions - Algorithms
- Sliding Window Algorithm - Longest Substring Without Repeating Characters (LeetCode)
- Minimum Window Substring: Utilizing Two Pointers & Tracking Character Mappings With A Hashtable

Questions □

- Maximum Average Subarray I
- 219. Contains Duplicate II
- 904. Fruit Into Baskets

- 1004. Max Consecutive Ones III
- 76. Minimum Window Substring
- 239. Sliding Window Maximum

Binary Search

Introduction

 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;
// loop till the search space is exhausted
while (left <= right)
{
    // find the mid-value in the search space and
    // compares it with the target
    int mid = (left + right) / 2;

    // overflow can happen. Use:
    // int mid = left + (right - left) / 2;
    // int mid = right - (right - left) / 2;

    // key is found
    if (x == A[mid]) {
        return mid;
    }

    // discard all elements in the right search space,
    // including the middle element
    else if (x < A[mid]) {
        right = mid - 1;
    }

    // discard all elements in the left search space,
    // including the middle element
    else {
        left = mid + 1;
    }
}
```

Here's a beautiful **visualization** to understand it even more: [Branch and Bound - Binary Search]

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 cases -> 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.

```
binarySearch(arr, l, mid - 1, x)
```

```
binarySearch(arr, mid + 1, r, x)
```

Here's the whole thing:

```
def binarySearch (arr, l, r, x):
    # Check base case
    if r >= l:
        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1
```

Let's discuss some intro level questions which can be solved by just the generic template for the binary search algorithm.

Intro Problems

- Leetcode-First Bad Version

Find the first element which is bad -> simply use the binary search template.

```
def firstBadVersion(self, n) -> int:
    left, right = 1, n
    while left < right:
        mid = left + (right - left) // 2
        if isBadVersion(mid):
            right = mid
        else:
```

```
            left = mid + 1
    return left
```

- Sqrt(x)

Find the square root of the number given. We can skip iteration over all the numbers and can simply take the middle and go left or right.

```
def mySqrt(x: int):
    left, right = 0, x
    while left < right:
        mid = left + (right - left) // 2
        if mid * mid <= x:
            left = mid + 1
        else:
            right = mid
    return left - 1
```

Problem 1: Max font size (Google internship)

 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:

1. Height and width of a screen where you have to type
2. Height and width of each character you type on the screen
3. Min and max range of the 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. Shoot -> it's binary search.

Here's how the pseudo code looks like:

```
def find_max_font():
    max_font = 0
```

```
    start, end = min_font, max_font
    while start<=end:
        mid_font = start + (end-start)//2
        if font_fits(mid_font):
            max_font = max(max_font, mid_font)
        elif mid_font == 'too big':
            # move left if font is too big
            end = mid
        else:
            # move right if font is too small
            start = mid
    return max_font
```

This was an additional round (round 4), so they kept it on the easier side. Some things to keep in mind while taking a tech interview:

- Be clear with your thoughts and communicate well.
- Ask questions, look for hints, and explain before writing code.

Problem 2: Search in rotated sorted array

Leetcode #33 Search in Rotated Sorted Array

 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 we do better?

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. Let's dive right in.

We see that the array is sorted but from a different position...

Do you see 2 arrays which are sorted? -> with a number in between which separates both the arrays?

Think how we can use binary search here.

Potential solution: Let's call that a pivot point, separate out both the arrays, and find the element in both separately using binary search? Does this make sense? No? Email us 30DaysCoding@gmail.com and let's discuss it there.

```
int pivot = findPivot(array);
if (pivot > 0 && num >= array[0] && num <= array[pivot - 1]) {
    return binarySearch(array, 0, pivot - 1, num);
} else {
    return binarySearch(array, pivot, array.length - 1, num);
}
```

Now we want to find the pivot and then also write the binary search algo - which will be the cliche binary search algorithm.

 After some more digging, you'll realize that this can be done with a **single binary search** method as well. Let's discuss that ->Instead of checking the mid with target (as done in a generic binary search), we check the mid with start and end -> cause the array is distorted -> so first we want to condition on that.

Let's say the nums[start] is less than the nums[mid] -> we get our new start and end -> the start and mid. We get this condition:

```
if (nums[start] <= nums[mid]){
      if (target < nums[mid] && target >= nums[start])
            end = mid - 1;
      else
            start = mid + 1;
}
```

So we just add one more condition to the already existing binary search conditions. We shift the start and end pointers **after** we've discovered the **subarray** where we need to shift. Here's the full code:

```
public int search(int[] nums, int target) {
    int start = 0;
    int end = nums.length - 1;
    while (start <= end){
        int mid = (start + end) / 2;
        if (nums[mid] == target)
            return mid;

        if (nums[start] <= nums[mid]){
             if (target < nums[mid] && target >= nums[start])
                end = mid - 1;
```

```
            else
                start = mid + 1;
        }

        if (nums[mid] <= nums[end]){
            if (target > nums[mid] && target <= nums[end])
                start = mid + 1;
            else
                end = mid - 1;
        }
    }
    return -1;
}
```

Similar Patterns

□ □ □ There are other, advanced use cases of binary search where we want to find a minimum time or a minimum space (and more). One catch with every binary search question is the limit from low to high -> which isn't trivial for those problems.

For instance, Leetcode : Minimum Number of Days to Make m Bouquets. We can make 'm' bouquets and each one needs 'k' flowers. It doesn't look like a problem which can be solved using binary search, but it can be. We can often define a new function which does additional condition mapping for us and then helps us find the *middle*.

Here's a generic template and some awesome information to binary search questions and identify problems where there is a limit defined. Binary search template.

Read

- Lecture 5 MIT : Binary Search Trees, BST Sort | Lecture Videos
- Binary search cheat sheet for coding interviews. | by Tuan Nhu Dinh | The Startup
- Binary Search Algorithm 101 | by Tom Sanderson | The Startup

Videos □

- Introduction to Binary Search (Data Structures & Algorithms #10)

Questions □

- Leetcode #704 Binary Search

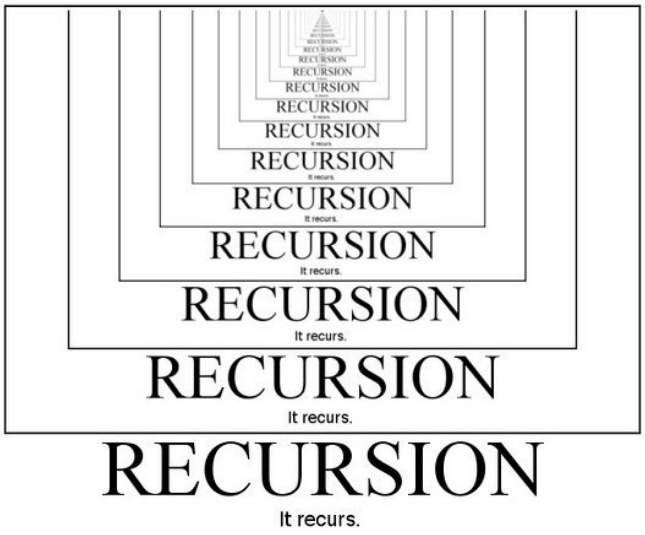
- Leetcode #349 Intersection of Two Arrays
- Leetcode-First Bad Version
- Arranging Coins
- 35. Search Insert Position
- Leetcode #33: Search in Rotated Sorted Array
- 34. Find First and Last Position of Element in Sorted Array
- Leetcode #230 Kth Smallest Element in a BST
- Find Peak Element
- Leetcode Split Array Largest Sum
- 875. Koko Eating Bananas
- Leetcode : Minimum Number of Days to Make m Bouquets

Recursion

Introduction

 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.



Solving a lot of recursive problems will help you understand 3 core concepts

- Recursion
- Backtracking
- Dynamic programming

Watch this **amazing** video: Recursion for Beginners: A Beginner's Guide to Recursion

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

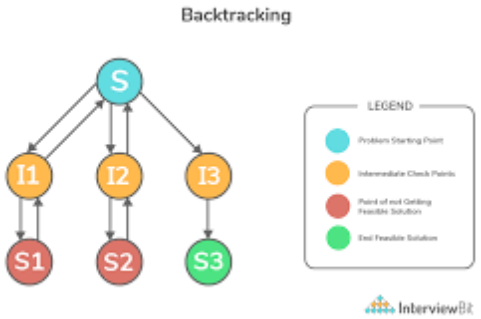
- What happens when the function is called in the **middle** of the whole recursive function?
- What happens to the stuff **below** it?
- What do we think of the base case?
- How do we figure out when to **return** ?
- How do we save the value, specially in the **true/false** questions?
- How does **backtracking**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

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



Problem 1: Generate parentheses

22. Generate Parentheses

 Generate balanced parentheses, given a number.

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.

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

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<right.

Base case? When both right and left are 0? -> cause we’re subtracting one as we go down to 0. Here’s the final thing:

```
public void dfs(List<String> list,String s, int right, int left){
    if(right==0 && left==0){
        list.add(s);
```

```
    }

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

Here are some other solutions to this: Generate Parentheses Solutions

Full code: Generate Parentheses Solution

Problem 2: Reverse linked list

 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.

```
def reverseList(self, head):
    prev = None
    while head:
        curr = head
        head = head.next
        curr.next = prev
        prev = curr
    return prev
```

Here's a nice video with the explanation: Reverse a Linked List Recursively

We can also solve this recursively and it's a great way to understand it in a better way. Here's how we do it:

- 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)

```java
public ListNode reverseList(ListNode head) {
    if(head == null || head.next==null){
        return head;
    }
    ListNode n = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return n;
}
```

It's not a super important thing to know, but a nice-to-have as a concept when you're preparing.

Pattern: Breaking down

 Tons of problems where you see a breakdown pattern can be solved using recursion. Some of them are: power of two, power of three, division, multiples,

These are problems and patterns where we see a bigger number and we want to break it down into a smaller thing to test. This is in alignment with the core of recursion, but it's easier to understand when math comes into play.

Let's discuss a question, Power of 3. We want to return true if the number given is a power of 3.

- Iterate and find the powers -> match them
- Optimized: Iterate for less numbers
- Recursively try to solve smaller problems and break it down into n/3 every time (power of 3)

```java
bool isPowerOfThree(int n)
{
   if(n<=0)return false;
   if(n%3==0)
       return isPowerOfThree(n/3);
   if(n==1)return true;
    return false;
}
```

Another question, to solidify the concept: Power of 2.
Super similar to power of 3, let's look at possible solutions and maybe a new approach for this.

- Iterate and find powers, match if possible
- Optimized: Iterate for less numbers and then match
- Recursively break it down into n/2 if it doesn't match and have base cases to check

```
def isPowerOfTwo(self, n: int):
    if n==0:
        return False
    if n==1:
        return True
    if n%2!=0:
        return False
    return isPowerOfTwo(n//2)
```

Problem 3: Letter combination of phone numbers

17. Letter Combinations of a Phone Number

 Interesting problem and can be solved both iteratively and recursively (same for any problem).

The first thing which comes to mind is to have a map of the numbers and digits, so that we can actually use it. The second thing which is trivial is that -> we would iterate over, take all the possible ways, and then store it in a list. It's basically a cliche backtracking problem where we have some arrays and we want all the possible cases in those.

A recursive function would need to have something in the arguments which we add + we update the array (using python sub-array)

```
combo(combination+letter, digits[1:])
```

We do this for every letter and add a base case for adding the combination to the result array. Here's how the complete code looks like

```
def combo(combination, digits):
      if len(digits)==0:
          a.append(combination)

      else:
          for letter in phone[digits[0]]:
              combo(combination+letter, digits[1:])
```

Here's a java solution code for it: My recursive solution using Java

Let's also look at an iterative way of solving this. We can simply take a Queue and use BFS (sort of) to iterate and then add the letters when the conditions are true. We can iterate over the digits, add the possible combinations if the size is valid.

Here's a nice solution for it: My iterative solution, very simple under 15 lines.

Backtracking goes hand in hand with recursion and we've discussed many more questions and patterns in that section, so definitely follow that after this.

Read □

- Reading 10: Recursion
- Recursion for Coding Interviews: The Ultimate Guide

Videos □

- Fibonacci Sequence - Recursion with memoization
- Introduction to Recursion (Data Structures & Algorithms #6)
- Intro to Recursion: Anatomy of a Recursive Solution

Questions □

- Explore: Leetcode Part I
- Explore: Leetcode Part II
- 150 Questions: Data structures

Extra

- Complex Recursion Explained Simply
- Recursion Concepts every programmer should know

Backtracking

Introduction

 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

46. 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

Great article on more **backtracking problems** templates: A general approach to backtracking questions in Java (Subsets, Permutations, Combination Sum, Palindrome Partitioning)

```
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: Subsets

https://leetcode.com/problems/subsets/

 We want all the possible subsets of an array [1,2,3]. Super similar to the permutations question, but we don't want to make the array shorter or anything, Just **explore** all the possible options.

We usually make a second function which is recursive in nature and call that from the first one -> it's easier, cleaner, and more understandable. There are certain ways of doing it in the same function, but this is better.

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;
}
```

Let's build the backtrack function. Let's use our template logic:

- Iterate over the array
    - Add the item
    - Backtrack - recursive call
    - Remove the item

And then think of the base case...

```
public backtrack(List<List<Integer>> list , List<Integer> tempList, int [] 
nums, int start){

    # Add the BASE CASE here

    for(int i = start; i < nums.length; i++){
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);
    }
}
```

Base case?

We want all the possible cases -> just simply add to a new list that we pass in?

```
list.add(new ArrayList<>(tempList));
```

Something to note here is that we add a new copy of the array (templist) -> and not the same templist because of recursion. Try it!

 There are other solutions to problems like these and backtracking problems in general. You can avoid the for loop and iterate over the array through the index you pass in to the function. Here are some things to consider while considering this approach

- Base case: index reaching the end of the array
- Add the item, recurse, remove the item
- Recurse without considering the item
- We recurse 2 times - with and without the element -> which is the niche of backtracking, where we have a CHOICE

So this is more on the lines of brute force when you have a CHOICE. A general approach there is to recurse when you've chosen the item and when you've not chosen it.

```
private void recur(List<List<Integer>> acc, int [] array, Stack path, int 
index){
    if(array.length == index){
        acc.add(new ArrayList<>(path));
        return;
    }
```

```
    // with array[index]
    path.push(array[index]); // add array[index]
    recur(acc, ns, path, index + 1);
    path.pop(); // remove array[index]

    // without array[index]
    recur(acc, ns, path, index + 1);
}
```

Read this carefully before moving forward. It's important to make the right CHOICES in your life haha. Make sure they're the good ones. Read more here: A general approach to backtracking questions in Java (Subsets, Permutations, Combination Sum, Palindrome Partitioning)

Problem 3: Combination Sum

39. Combination Sum

 We want to return the numbers which would add up to the target number given. We have to return all the possible combinations. So this is basically all subsets (with repeats allowed) with a target given.
From the get go, I know one thing -> we want to explore all cases, find the ones where the target matches, and then add that to a list, and return that list.
Backtracking template: Make a choice

- Iterate over the array
    - Add the item
    - Backtrack - recursive call
    - Remove the item

```
for(int i = start; i < nums.length; i++){
    tempList.add(nums[i]);
    backtrack(list, tempList, nums, target_left - nums[i], i); // not i + 1
because we can reuse same elements
    tempList.remove(tempList.size() - 1);
}
```

A good thing to note here is that we pass in the target_left - nums[i] which basically means that we're choosing that element and then subtracting that from what we have in the argument. So the base case with this would be
Target_left == 0 -> because that's when we know we can make the target.

One other thing to save some time and memory can be target_left < 0 -> to return when we reach here, because negative numbers can never become positive numbers. So once the target_left is below 0, it can never come up -> good to just return;

```
public List<List<Integer>> combinationSum(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList,
int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i); // not i
+ 1 because we can reuse same elements
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Give this a good read, watch this: AMAZON CODING INTERVIEW QUESTION - COMBINATION SUM II (LeetCode) and make sure to understand it before moving forward.

Problem 4: N-queens

51. 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

```
board[row][col] = 'O' # the whole board
for i in range(0, N):
    if isValidPosition(board[row][col]):
        board[row][col] = 'Q' # set queen
        recursive()
        board[row][col] = 'O' # remove queen
```

Make sure to think about the base cases, recursive calls, the different parameters, and validating functions. Reference: Printing all solutions in N-Queen Problem

Here's a beautiful **visualization** for this question: Backtracking - N-Queens Problem

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.

```
function(i, value, something...){
    if base_case:
        do something

    if stored_value[i]:
        return stored_value[i]
```

```
    // do something (recursive call)
    stored_value[i] = value
}
```

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.

Read

- A deep study and analysis of Recursive approach and Dynamic Programming by solving the most…
- Leetcode Pattern 3 | Backtracking | by csgator | Leetcode Patterns
- A general approach to backtracking questions in Java (Subsets, Permutations, Combination Sum, Palindrome Partitioning)
- WTF is Memoization. Okay, those who saw this term for the… | by Leo Wu | Medium

Questions

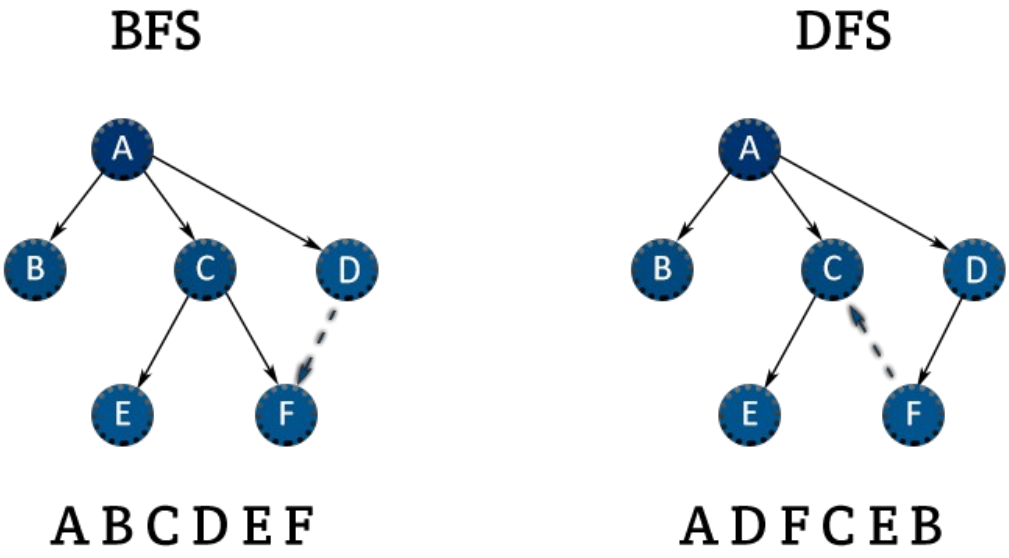
- Word Search
- Leetcode #78 Subsets
- 90. Subsets II
- Letter Case Permutation
- 17. Letter Combinations of a Phone Number
- Combinations
- 39. Combination Sum
- Leetcode : Combination Sum II
- 216. Combination Sum III
- Combination Sum IV
- 46. Permutations
- 47. Permutations II
- 31. Next Permutation
- 51. N-Queens

BFS, DFS

Introduction

 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's a great visualizer tool: Graph Traversal (Depth/Breadth First Search)



Try to understand the iterative way of solving a DFS or BFS question and how things work. There are 3 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)

Here's a beautiful visualization of a search in a tree: Branch and Bound - Depth-Limited Search
Here's a general iterative dfs pseudo-code template:

```
def dfs(root, target):
    stack = []
    stack.append(root) # add the first item

    while len(stack)>0:
        node = stack.pop() # pop the grid item

        if(node == target):
            return true

        # explore more
        # For trees -> if root.left or root.right
        if (condition):
            stack.append(new_item)

    return false;
```

The second step is that of MEMOIZATION and we want to keep a track of all the nodes visited when we're iterating over. Here's a complete version of a BFS algorithm where we keep track of the visited node using an array **discovered []**
This could be anything - array, map, set - depending on the situation. The only thing we need is to store the visited things so that we're not repeating any work.

```
public static void BFS(Graph graph, int v, boolean[] discovered)
    {
        // create a queue for doing BFS
        Queue<Integer> q = new ArrayDeque<>();

        // mark the source vertex as discovered
        discovered[v] = true;

        // enqueue source vertex
        q.add(v);

        // loop till queue is empty
        while (!q.isEmpty())
```

```
        {
            // deque front node and print it
            v = q.poll();
            System.out.print(v + " ");

            // do for every edge `v --> u`
            for (int u: graph.adjList.get(v))
            {
                if (!discovered[u])
                {
                    // mark it as discovered and enqueue it
                    discovered[u] = true;
                    q.add(u);
                }
            }
        }
    }
```

Trying to think of a recursive way to do this is also very important. We call dfs for every node after exploring the neighbors and can do that in a couple of ways -> inside the for loop or outside the for loop after adding the neighbors to a list. Here's an approach, also linking other approaches below.

```
public static void recursiveBFS(Graph graph, Queue<Integer> q,
                                    boolean[] discovered)
{
    if (q.isEmpty()) {
        return;
    }
    // deque front node and print it
    int v = q.poll();
    System.out.print(v + " ");

    // do for every edge `v --> u`
    for (int u: graph.adjList.get(v))
    {
        if (!discovered[u])
        {
            // mark it as discovered and enqueue it
            discovered[u] = true;
            q.add(u);
        }
```

```
    }

    recursiveBFS(graph, q, discovered);
}
```

Other recursive ways: Depth First Search or DFS for a Graph

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.

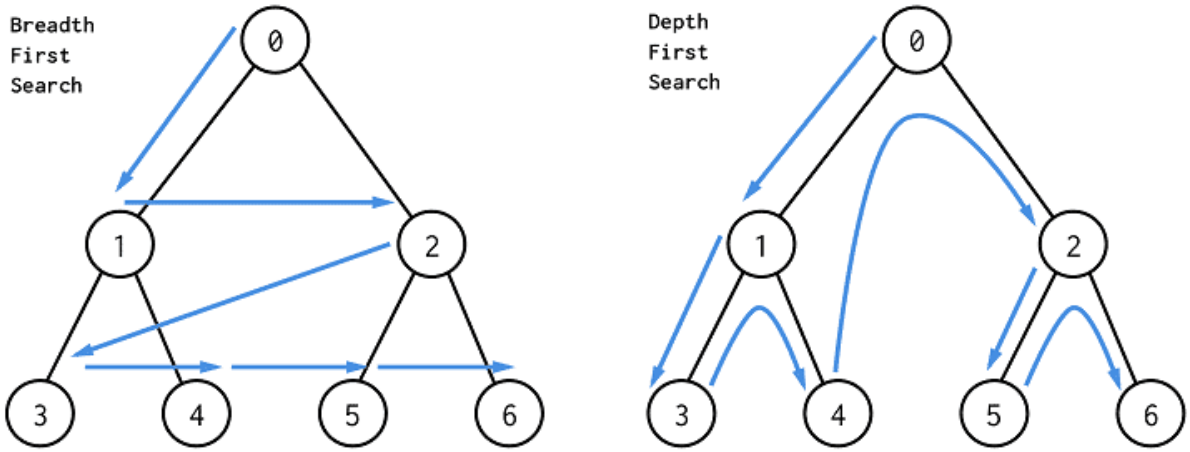
Here are some implementations and use cases for DFS, BFS:

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

Number 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
```

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 it's 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, let's solve this both iteratively and recursively.

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, 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

```
public int numIslands(char[][] grid) {
        int count=0;
        for(int i=0;i<grid.length;i++){
            for(int j=0;j<grid[0].length;j++){
                if(grid[i][j] == '1'){
                    dfs(grid, i, j);
                    count+=1;
                }
            }
        }
        return count;
    }

    public void dfs(char[][] grid, int i, int j){
        if(i<0 || i>=grid.length || j<0 || j>=grid[0].length){
            return;
        }
        if(grid[i][j]== '1'){
            grid[i][j] = '#';
            dfs(grid, i+1,j);
            dfs(grid, i,j+1);
            dfs(grid, i,j-1);
            dfs(grid, i-1,j);
        }
    }
```

Here's an iterative way to solve this:

The idea is the same, we start from the 1s, explore all the connected components, mark them visited, and then increase the count for every 1. These are the steps:

- Iterate over the find the 1's
- Call dfs for every 1 found

Iterative DFS

- Push the grid node for the 1 to the stack
- Mark the node visited (change it to something else)
- Pop the node, explore all the 4 valid neighbors
- Add those neighbor nodes to the stack

```
count=0
for i in range(len(grid)):
    for j in range(len(grid[0])):
        if grid[i][j]=='1':
            dfs(grid, i, j)
            count+=1
return count

def dfs(grid, i, j):
    s=[]
    s.append((i,j))
    while len(s)>0:
        a,b = s.pop()
        grid[a][b]='X'
        if a>0 and grid[a-1][b]=='1':
            s.append((a-1,b))
        if b>0 and grid[a][b-1]=='1':
            s.append((a,b-1))
        if a<len(grid)-1 and grid[a+1][b]=='1':
            s.append((a+1,b))
        if b<len(grid[0])-1 and grid[a][b+1]=='1':
            s.append((a,b+1))
```

Here's a nice solution video explaining the same:
GOOGLE CODING INTERVIEW QUESTION - NUMBER OF ISLANDS (LeetCode)

Pattern: 2D Matrix

 There are tons of problems where there's something to find or connect in a 2d array where the confusions just increase. This approach will help you connect the dots and approach those problems with DFS or BFS iteratively on that array.
There's nothing special here, but it's good to notice how we can take [0,0] as the root and basically convert this into a 2d matrix. This is a general way of adding the point to the queue in java, with the help of an additional class Pair => q.offer(new Pair(i, j));

```
def dfs(grid, row, col):
    stack = []
    stack.append((row,col)) # add the first item

    while len(stack)>0:
        row,col = stack.pop() # pop the grid item

        grid[row][col]='X' # mark it visited

        # conditions come here
        if (condition):
            stack.append(new_item)
```

Problem 2: Level order traversal

LeetCode 102 - Binary Tree Level Order Traversal [medium]
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's how I would do it -> Think of adding the root, take out the root, add the whole second layer or basically all the children of the previous layer's nodes. The catch here is to add the **whole level** at once. We can do that by getting the size of the queue and then iterating over it every time.

```
for(int i=0 ; i< size; i++){
    TreeNode node = queue.remove();

    // Temporary list for that level
    list.add(node.val);
    if(node.left!=null)
        queue.add(m.left);
    if(node.right!=null)
        queue.add(m.right);
}
```

At the end, the queue would have the next level and we'll repeat the whole process again for the next nodes. Here's how the code looks:

```
Set<Integer> solution = new HashSet<>();
Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);
while(!q.isEmpty()){
    List<Integer> list = new ArrayList<>();
    int children = queue.size();
    // iterate over all the children
    for(int i=0 ; i<children; i++){
        TreeNode node = queue.remove();

        // Temporary list for that level
        list.add(node.val);

        if(node.left!=null)
            queue.add(m.left);
        if(node.right!=null)
            queue.add(m.right);
    }
    solution.add(new ArrayList<>(list));
}
return solution;
```

Problem 3: Rotten oranges

994. Rotting Oranges

 Every minute a fresh orange turns rotten if it's around a rotten orange. Similar to life -> if you're around negative people, you tend to be negative. Keep a positive outlook, help everyone, and take things forward!

This is an amazing question -> let's understand the iterative way of doing this and how to solve any searching related question with a stack or queue -> iteratively. We have the minimum condition here, so using BFS is the way to go! A simple pattern, as discussed before is:

- Prepare the stack/queue -> Add the initial nodes
- Pop the node from stack, mark it visited, add the valid neighbors
- Repeat the process for the new nodes.

First step is to prepare the queue. We add the rotten oranges (represented by 2) to the queue and also count the total number of oranges. 0 -> means an empty place.

```
for (int i = 0; i < grid.length; i++) {
    for (int j = 0; j < grid[0].length; j++) {
        if (grid[i][j] != 0) total++;
        if (grid[i][j] == 2) q.offer(new Pair(i, j));
    }
}
```

We have the queue ready and now we iterate until it's empty: *while (stack.isEmpty()) {}*. We want to add all the neighbors of the current orange, which are in 4 directions and here's something to note when you have conditions like this.

When we want to traverse in all 4 directions, or maybe in 8 directions if we have a double condition, we can make a directions dictionary and iterate over it. Something like: [[0,1], [0,-1], [1,1], [1, 0]] or *int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};*

```
while (! q.isEmpty()) {
    int size = q.size();
    rotten += size;
    // if the total number of rotten oranges matches our local variable
    // then return the time it took
```

```
    if (rotten == total_rotten) return time;

    time++;
    // something
}
```

Now, we add the logic for adding the neighbors.

- Iterate in all 4 directions
- Check if it's a fresh orange -> continue the loop if it's rotten or empty cell
- Change the fresh orange into a rotten one
- Add the new position in the queue
- Increase the rotten orange for the base case -> *rotten_oragen == total*

```
for(int i = 0 ; i < size ; i++) {
      int[] point = queue.poll();
      for(int dir[] : dirs) {
          int x = point[0] + dir[0];
          int y = point[1] + dir[1];

          // check for the conditions
          // continue if it's rotten or empty
          // we're only concerned about the fresh ones here
          if(x < 0 || y < 0 || x >= rows || y >= cols || grid[x][y] == 0 ||
grid[x][y] == 2) continue;

          // turn fresh into rotten
          grid[x][y] = 2;

          queue.offer(new int[]{x , y});

          rotten_oranges++;
      }
}
```

Complete code here: [Java] Clean BFS Solution with comments

Video solution: AMAZON CODING INTERVIEW QUESTION - ROTTING ORANGES

Read □

- Leetcode patterns 1
- Leetcode Patterns 2
- Depth-First Search (DFS) vs Breadth-First Search (BFS) – Techie Delight

Videos □

- Breadth First Search Algorithm | Shortest Path | Graph Theory
- Depth First Search Algorithm | Graph Theory
- Breadth First Search grid shortest path | Graph Theory

Questions □

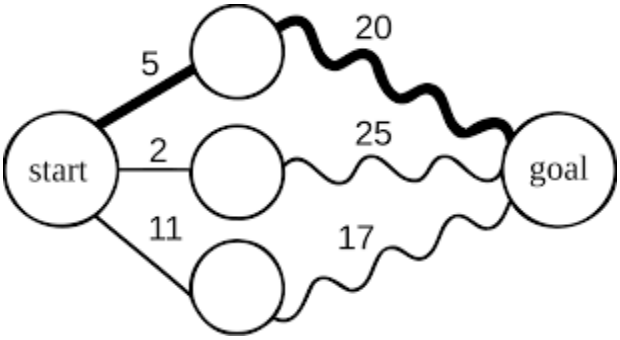
- Flood Fill
- Leetcode - Binary Tree Preorder Traversal
- Number of Islands
- Walls and Gates
- Max Area of Island
- Number of Provinces
- 279. Perfect Squares
- Course Schedule
- C/C++ Program for Detect cycle in an undirected graph
- 127. Word Ladder
- 542. 01 Matrix
- Rotting Oranges
- 279. Perfect Squares
- 797. All Paths From Source to Target
- 1254. Number of Closed Islands

Dynamic Programming

Introduction

 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.

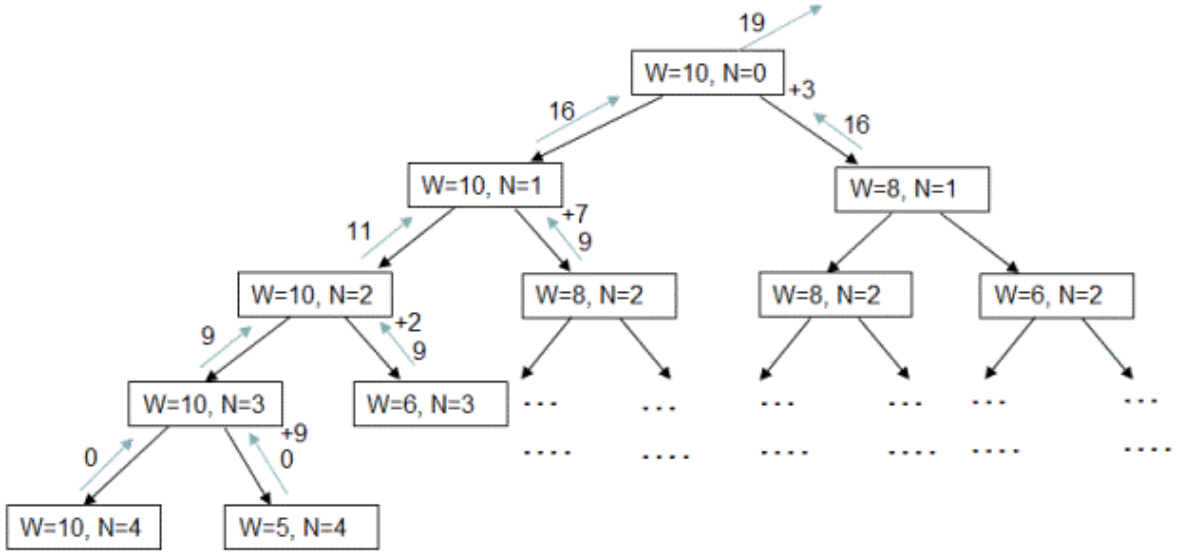
Problem 1: 01 Knapsack

 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.

Here's the code for it:

```
knapsack(weights [], values [], maxWeight = 0, index = i, memo_set = set())
{
  if(i ==  weights.length || maxWeight == 0){
      return  0;
  }
```

```
  //include the ith item
  int include = v[i] + knapsack(w, v, maxWeight - weights[i], i+1);
  // don't include
  int exclude =  knapsack(w, v, maxWeight, i+1);

  return Math.max(include, exclude);
}
// w -> weights, v -> values
```

There is a problem here, we're doing a lot of repetitive work here, do you see it? No? Go **wash** your eyes. 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.

```
knapsack(weights [], values [], maxWeight = 0, index = i, memo_set = set())
{
  if(i ==  weights.length || maxWeight == 0){
    return  0;
  }

  String key = maxWeight + "unique Key" + i;

  // check if the key is inside this set
  if(memo_set.containsKey(key)){
    return memo_set.get(key);
  }

  //include the ith item
  int include = v[i] + knapsack(w, v, maxWeight - weights[i], i+1, set);
  // don't include
  int exclude =  knapsack(w, v, maxWeight, i+1, set);

  memo_set.put(key,Math.max(op1, op2));
  return Math.max(include, exclude);
```

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.

- Watch this awesome visualization to understand it more: Dynamic Programming - Knapsack Problem
- Read more here: 0–1 Knapsack Problem – Techie Delight

Problem 2: Min path sum

64. Minimum Path Sum

 Most of the dynamic programming (and all other) questions are solved by making a choice! Let's discuss a question. Find the minimum cost path from top left to bottom right of a 2D matrix.

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

A human way to look at this is to make quick decisions and see where the biggest numbers are, and then choose them. However, humans would fail if this grid is really big.

How do we solve this using a program?

At every step, we make a CHOICE. Either we go down or we go right. And this is where recursive kicks in, making this a dynamic programming question -> where we try to solve small problems to eventually solve the big one.

At every step, we'll do these 2 things:

```
bottom_sum = current_sum + grid[row+1][col]
right_sum = current_sum + grid[row][col+1]
```

And now we convert this into recursive code,

```
current_sum = grid[row][col]
max_sum = min(
current_sum + function(grid[row+1][col]),
```

```
current_sum + function(grid[row][col+1]
))
```

Now we can make it even easier by just passing the rows and columns instead of the whole grid and bringing out some things to clean it.

```
current_sum = grid[row][col]
max_sum = current_sum + min(function(row+1, col),function(row, col+1]))
```

Instead of calculating the sum at every step, we pass it back to the recursive function who does the magic for us. We would have a BASE CASE which helps us in solving the smaller problem, which eventually solves the big one.

```
// this is the exit of the recursion
if(row == 0 && col == 0) return grid[row][col];

/** When we reached the first row, we could only move horizontally.*/
if(row == 0) return grid[row][col] + min(grid, row, col - 1);

/** When we reached the first column, we could only move vertically.*/
if(col == 0) return grid[row][col] + min(grid, row - 1, col);

return grid[row][col] + min(f(grid, row - 1, col), f(grid, row, col - 1));
```

More here: Minimum path sum solution.

Problem 3: Minimum cost of tickets

 Another interesting problem to discuss, let's do it.

We're going on a trip and we want to make it as cheap as possible (cause we're all cheap people). We want to save as much money as possible, and we're gonna write a piece of code which does it for us.

Here's the deal: We have costs for 1, 7, and 30 days, and an array of the days we're travelling, we want to optimize it such that the cost is the lowest. Solving it a humanly way, we would check all the possible ways and then make a decision -> hence making it a DP problem -> we explore all the possible cases with brute force and then memoize it.

Exploring all the cases:

```
int option_1day = costs[0] + rec(days, costs, current_day);
int option_7days = costs[1] + rec(days, costs, current_day);
int option_30days = costs[2] + rec(days, costs, current_day);
```

We need to have a way to change the 'days' such that -> if we choose option 1 (1 day), we want to move to the next POSSIBLE day. If we choose option 2 (7 days), we want to move to the next POSSIBLE day within 7 days and the same with 30 days. So there's a condition before we recurse every time -> we want to change the current_day variable

Here's the condition:

```
for(int i=0; i < days.length; i++){
  // If we go beyond the possible limit, break
  if(days[i] >= days[current_day] + 1, 7, 30){ // For all 3 cases
     break;
  }
}
```

Base case? When the index or the current_day goes beyond the days array

```
if(current_day >= days.length) return 0;
```

This can be different depending on your conditions -> maybe you're iterating over the days array through a for loop and creating some magic there. A right base case would probably be validating the current day or something in that case.

Here's the combined code solution:

```
private static int rec(int days[], int costs[], int i, int dp[]){
    if(i >= days.length) return 0;

    int option_1day = costs[0] + rec(days, costs, i+1, dp);

    int k = i;
```

```
    for(; k <days.length; k++){
        if(days[k] >= days[i] + 7){
            break;
        }
    }
    int option_7days = costs[1] + rec(days, costs, k, dp);


    for(; k <days.length; k++){
        if(days[k] >= days[i] + 30){
            break;
        }
    }
    int option_30days = costs[2] + rec(days, costs, k, dp);

    return Math.min(1Day, Math.min(7Days, 30Days));
}
```

Problem 4: Buy and sell stocks 3

Leetcode - Best Time to Buy and Sell Stock III

 Try this first: Q. 121. Best Time to Buy and Sell Stock, although they're not similar, but it's nice to get a feel of that one before coming to this one.

Let's discuss this one. We have an array, we have to buy and then sell - 2 times, and then find the maximum profit we can earn by doing this. Eg [3, 3, 5, 0, 0, 3, 1, 4], let's solve this in a human way. Buy at 3, sell at 5. Then buy at 0, sell at 4. Total is 6. Easy? I just thought of the difference as I was going, what could be the maximum difference. However, this approach can only work for very simple examples or the first question (Q. 121. Best Time to Buy and Sell Stock).

Let's solve it through code.

At every step when we iterate from left to right, we have a CHOICE. It's a little complex, think a little. The CHOICE is to either buy or not buy OR sell or not sell when you're at that step. We do this because we're buying or selling only 2 times. Here's how the choices look:

```
// if we're buying right now
lets_buy = function() - array[i]
lets_not_buy = function()
```

```
// if we're selling right now
lets_sell = function() + array[i]
lets_not_sell = function()
```

Here's the code for this: Buy and Sell 3 solution

We're coming up with a dynamic programming guide with 25 questions discussed in complete detail, stay tuned for that. Subscribe to our newsletter here for more.

Problem 5: Paint house

leetcode 256. Paint House (Python)

 There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

At every step, we have a CHOICE to choose a color and then see what would be the maximum at the very end. So we explore all the possible cases, remove the repetitive cases using memoization, and eventually solve the question by 'DP'. At every step,

- If you choose red, then choose the min of blue or green from previous row
- If you choose blue, then choose the min of red or green from previous row
- If you choose green, then choose the min of red or blue from previous row

```
public int minCost(int[][] costs) {
    if(costs == null || costs.length == 0) return 0;
    int n = costs.length;
    for(int i = 1; i < n; i++){
        costs[i][0] += Math.min(costs[i-1][1], costs[i-1][2]);
        costs[i][1] += Math.min(costs[i-1][0], costs[i-1][2]);
        costs[i][2] += Math.min(costs[i-1][0], costs[i-1][1]);
    }

    return Math.min(costs[n-1][0], Math.min(costs[n-1][1], costs[n-1][2]));
}
```

Solution Video: LINKEDIN - PAINT HOUSE (LeetCode)

Problem 6: Edit Distance

LeetCode – Edit Distance

 Here's the question: We have 2 strings and we want to transform the first one into the second one using minimum operations. Every time, you can either insert, delete, or change the letter from any of the strings.

WOW, I have no clue how to solve this to be honest :P. Let's think together.

We have 3 CHOICES, edit/delete/or inserting a new character. This gives me a hint that at every step, I can do 3 things and eventually explore all possible ways to find the answer. We can then use memoization for repetitive work and we'll have our answer. Sounds easy? … no it's not. Come on, when did DP become easy?

Just kidding, let's make it easy. 3 choices? 3 recursive options -> insert, delete, and update. But there's a catch, deletion doesn't mean we're deleting -> we'll just call the string[1:] or string.substring(1) in the recursive function to create the deletion identity. Same for inserting -> adding a letter in one string, means deleting something from the other (in a way), so we can mix and match the deleting/insertion operations. Coming to update -> that just means we're changing that letter and moving forward, so the recursive call will be first[1:] and second[1:]. Here's how the recursive calls look like:

```
int delete =  rec(s, t.substring(1));
int insert =  rec(s.substring(1), t);
int update =  rec(s.substring(1), t.substring(1));
```

Base case? You forgot right? Well, forget getting that internship then. Just kidding, let's think of the base case -> if we have both the strings inside our function -> if one of them finishes (because we're taking substrings) -> we should handle those cases. Here's how that will look:

```
if(first_string.length() == 0)
  return second_string.length();

if(second_string.length() == 0)
  return first_string.length();
```

Matching case? We also want to recurse with substring(1:) when both the characters match. This is the same as the update operation but without adding 1 to the final result.

Watch this awesome visualization: Dynamic Programming - Levenshtein's Edit Distance

Here's the combined result:

```
public static int rec(String s, String t){
    if(first.length() == 0)
         return second.length();

    if(first.length() == 0)
         return second.length();

    // if characters are same
    if(s.charAt(0)  == t.charAt(0))
         // don't add 1 here as the characters match
         return rec(s.substring(1), t.substring(1));
    else{
        int op1 =  rec(first.substring(1), second.substring(1));
        int op2 =  rec(firs, second.substring(1));
        int op3 =  rec(first.substring(1), second);

        return 1 + Math.min(op1, Math.min(op2, op3));
    }
}
```

Here's the full solution: luckykumardev/leetcode-solution

Read □

- My experience and notes for learning DP
- Dynamic Programming (Theory - MIT)
- Dynamic Programming (Theory MIT)

Videos □

- MIT Playlist: 19. Dynamic Programming I: Fibonacci, Shortest Paths
- Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges

Questions □

Easy

- 53. Maximum Subarray

- 509. Fibonacci Number
- 70. Climbing Stairs
- Min Cost Climbing Stairs
- N-th Tribonacci Number

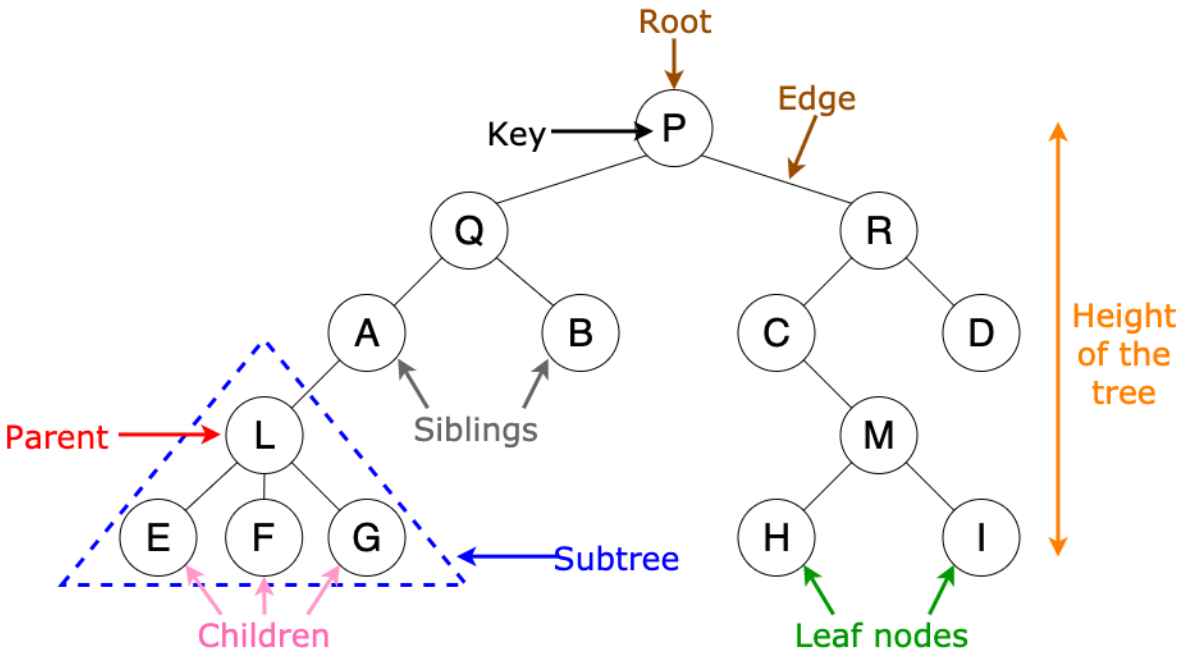
Medium

- 322. Coin Change
- 931. Minimum Falling Path Sum
- Minimum Cost For Tickets
- 650. 2 Keys Keyboard
- Leetcode #152 Maximum Product Subarray
- Triangle
- 474. Ones and Zeroes
- Longest Arithmetic Subsequence
- 416 Partition Equal Subset Sum
- 198. House Robber
- Leetcode - Decode Ways
- 139. Word Break
- LeetCode – Edit Distance
- 300. Longest Increasing Subsequence
- 787. Cheapest Flights Within K Stops

Trees

Introduction

 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

Let's start with inorder traversal: We define a stack and will traverse the tree iteratively. Recursive solutions to these 3 basic ones are pretty straightforward, so we'll try to understand them a little more with iterative ones.

We start with the root, move until it's null or the stack is empty. We move to the left if we can, if not -> we pop, add the popped value and then move right.

```
List<Integer> res = new ArrayList<>();
```

```
if(root==null) return res;

Stack<TreeNode> stack = new Stack<>();
TreeNode curr = root;
while(curr!=null || !stack.isEmpty()){
    if(curr!=null){
        stack.push(curr);
        curr = curr.left;
    }else{
        curr = stack.pop();
        res.add(curr.val);
        curr = curr.right;
    }
}
return res;
```

Pre order traversal

 We add the root, then the left subtree, and then the right subtree. It's a stack so things work in the opposite direction -> first in last out, so make sure to check that carefully.

```
Stack<Node> stack = new Stack();
stack.push(root);
result = [];

while (!stack.empty())
{
    Node curr = stack.pop();
    result.push(curr.data);
    // print node

    if (curr.right != null) {
        stack.push(curr.right);
    }

    if (curr.left != null) {
        stack.push(curr.left);
    }
}
```

Post order traversal

 We visit the left subtree, then the right subtree, and then the root. So we simply add the left item first, then the right item, and the root.

```
Stack<Node> stack = new Stack();
stack.push(root);
result = []

while (!stack.empty())
{
    Node curr = stack.pop();
    result.push(curr.data);

    if (curr.left != null) {
        stack.push(curr.left);
    }

    if (curr.right != null) {
        stack.push(curr.right);
    }
}
// Print the REVERSE of the result.
// Or store it in a stack
```

Additional questions

- LeetCode 102 - Binary Tree Level Order Traversal [medium]
- Kth Smallest Element in a BST
- Leetcode #98 Validate Binary Search Tree
- Binary Tree Zigzag Level Order Traversal
- Binary Tree Right Side View

Applications

- Number of nodes
- Height of tree or subtree
- Heap sorting

Problem 1: Min depth of a tree

 The question is -> what's the minimum depth or where is the lowest child for the tree.

From the get go, I'm thinking of finding a node which doesn't have any child?

It's about the height, so I'm thinking of going level by level and then seeing when we hit a node with no children? Oooooooo.. Sounds like a good plan, let's do that.

Let's go level by level and see where the node with no children is -> we return it as soon as we find that.

Here's the code for it:

```
public static int depthOfTree(TreeNode root) {
   Queue<TreeNode> queue = new LinkedList<>();
   queue.add(root);
   int minimumTreeDepth = 0;

   while (!queue.isEmpty()) {
     minimumTreeDepth++;
     int levelSize = queue.size();
     for (int i = 0; i < levelSize; i++) {
       TreeNode currentNode = queue.poll();

       // leaf node condition
       if (currentNode.left == null && currentNode.right == null)
         return minimumTreeDepth;

       // explore the children and add those
       if (currentNode.left != null)
         queue.add(currentNode.left);
       if (currentNode.right != null)
         queue.add(currentNode.right);
       // add other neighbors if this is a n-ary tree
     }
   }
   return minimumTreeDepth;
 }
```

This is the same as level order tree traversal -> we push the node initially, pop it -> add the children on the next level and then repeat the process.

Problem 2: LCA of binary tree

236 - Lowest Common Ancestor of a Binary Tree

 Problem: Find the lowest common parent of 2 given nodes in a tree. They could be any 2 nodes. Damn, what a nice question. We were traversing down for this whole time, but it sounds like we want to go up from both the nodes and then find the parent which comes first. How do we do that? (no clue, BYE) Just kidding, let's do it. The first thing which comes to my mind is that we can start from the node, maintain a separate list, add parent-node relation there and then maybe look at that list to find that first parent? It is very important to understand this (so writing code iteratively) and then thinking more towards writing a recursive solution.

Having a parent node relation is important here, so here's the first thing I think: We make a map and store {parent: node} inside that map as we go down.

```
 // store parent and node relation if valid
parent_map[node.left] = node
parent_map[node.right] = node
```

So we do a simple iterative DFS, store the parent node relation, and then come back to see that relation to find the common node.

```
while node_1 not in parent_map or node_2 not in parent_map:
        node = stack.pop()
        if node.left != None:
            parent_map[node_1.left] = node_1
            stack.append(node.left)
        if node.right != None:
            parent_map[node.right] = node
            stack.append(node.right)
```

Here we fill in the parent_map and try to build the parent node relation for all nodes. Once we have the map ready, we can use that map to go back and find the common ancestor. We make a set, add the first node in the set while iterating over the parent_map, then we check for the node_2 in the set and when we find that -> we break the while loop and return node_2 at that point.

```
node_set = set()
while node_1:
    node_set.add(node_1)
    node_1 = parent_map[p]
while node_2 not in node_set:
    node_2 = parent_map[node_2]
```

```
return node_2
```

It's often hard to come up with recursive solutions instantly, but over time - you'll be more comfortable (I'm not :P) to bring them up.
More solutions here: Lowest Common Ancestor of a Binary Tree

Problem 3: Binary tree to BST

 We have a binary tree and we want to convert that into a binary search tree, where the left subtree is smaller than the root, and the right subtree is greater than the root.
Doesn't this look similar to the inorder traversal ??? Inorder traversal gives us the binary search tree in a sorted order, so we can use that to bring it back up as well.
Wait… whaaaat? Haha yeah.

We just need an iterator to traverse through the next nodes from an array, list, set, or something else.
Here's how it'll look:

```
convertToBST(Node root, Iterator<Integer> it)
{
    if (root == null) {
        return;
    }

    convertToBST(root.left, it);
    root.data = it.next();
    convertToBST(root.right, it);
}
```

A solution video explaining the same: Converting Binary Tree to Binary Search Tree without changing spatial structure

Read

- Leetcode Pattern 0 | Iterative traversals on Trees | by csgator | Leetcode Patterns
- Inorder Tree Traversal – Iterative and Recursive – Techie Delight

Videos □

- Data structures: Introduction to Trees
- Binary Tree Bootcamp: Full, Complete, & Perfect Trees. Preorder, Inorder, & Postorder Traversal.
- 5. Binary Search Trees, BST Sort

Questions □

- Leetcode - Binary Tree Preorder Traversal
- Leetcode #94 Binary Tree Inorder Traversal
- Leetcode - Binary Tree Postorder Traversal
- Leetcode #98 Validate Binary Search Tree
- 783. Minimum Distance Between BST Nodes
- Symmetric Tree
- Same Tree
- Leetcode #112 Path Sum
- Leetcode #104 Maximum Depth of Binary Tree
- Leetcode #108 Convert Sorted Array to Binary Search Tree
- Leetcode #98 Validate Binary Search Tree
- Binary Search Tree Iterator
- 96. Unique Binary Search Trees
- Serialize and Deserialize BST
- Binary Tree Right Side View
- 96. Unique Binary Search Trees
- Binary Search Tree Iterator

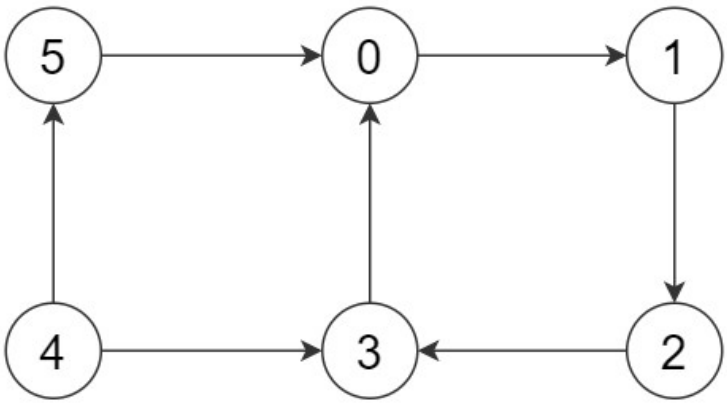
Graphs

Introduction

 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: Finding the root vertex



 A human way of finding the root will be to look at 4 and say that there are no incoming edges at 4, so it's the root. Think of it in a tree like format, where the root is at the top and we have children below it.

How do we code this?
We want to find a node which doesn't have any incoming nodes. So we start from the first node, go to the neighbors, mark all the neighbors visited (and not vertex -> because they have an incoming edge). We keep doing this until we reach the end and have a node which is not in the visited set.

An important part of graph algorithms is also to transform the given input into an adjacency list. We iterate over the edges and make a mapping from source to destination, something like this:

```
function(List<Edge> edges, int N)
    {
        adjList = new ArrayList<>();
        for (int i = 0; i < N; i++) {
```

```
            adjList.add(new ArrayList<>());
        }
        // add edges to the directed graph
        for (Edge edge: edges) {
            adjList.get(edge.source).add(edge.dest);
        }
    }
```

Then we use this list to do our searching!

So the solution here seems to be trivial -> we iterate over, find the new nodes, mark the neighbors visited, and then finally return the vertex. DFS/BFS anything works -> let's try to do it recursively. We have the theory of strongly connected components here, which is used to find different sets of nodes in a graph which are connected with each other -> which can be modified to return a node with no vertex.

Here are the steps

- We start exploring from 0 to n, we call DFS() if the node isn't visited, and then mark it and it's neighbors visited during DFS
- We also store the value of the iterating node till the very end -> this is the last node which was discovered and is our best bet
- We then call DFS again from this node and see if we find any unvisited nodes. If we do find any unvisited nodes -> it means that there are more than 2 root vertices -> return -1. If we don't find any unvisited, meaning that all nodes are visited, then we return the last element that we stored.

Let's analyse through code. We explore and call dfs on the nodes and keep a track of the last node:

```
boolean[] visited = new boolean[N];

int last_node = 0;
for (int node = 0; node < N; node ++)
{
    if (!visited[node])
    {
        DFS(graph, node, visited);
        last_node = node;
    }
}
```

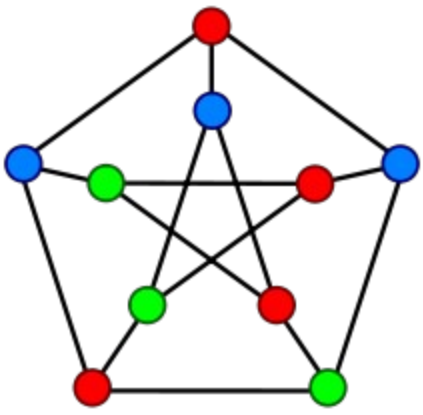
Once we're out of this loop, we have the last_node stored ->> which is our best bet of being the vertex.
Now we can reset the visited array and check if the visited array has any more nodes or not.
- Why are we resetting the value of the visited array? Because we want to do a fresh search.
- Why are we checking for visited [i] -> because the vertex should be the only one which was not visited.
- Why are we returning -1 => because we didn't find the vertex if there are more than 1 nodes not explored -> meaning more than 1 vertex.

```
// reset visited = [false] for every node
DFS(graph, last_node, visited);

for (int i = 0; i < N; i++)
{
    if (!visited[i]) {
      // return -1 if we find an unvisited node
        return -1;
    }
}

// return the last node if we've visited all nodes.
return last_node;
```

Here's an article, explaining more about this: Find root vertex of a graph – Techie Delight

Problem 2: Graph coloring

 This is a very interesting problem covering the core of graph iteration, so let's take a look here.



We want to color the nodes in a way that no 2 consecutive nodes have the same color. There are a lot of implementations attached to this concept, some of them are:

- Scheduling: Problems where we have a list of jobs or times or rooms, and we want to find an optimal way to find the schedules of different things.
- Other seating related problems can also be solved using this approach - where we don't want any 2 people sitting next to each other.

A general approach to problems like these:

- Start from a node
- Check the colors of the neighbors, store it in a set, and then use that set for the next color.

```
// check colors of adjacent vertices of `node` and store them in a set
for (int i: graph.adjList.get(u))
{
    if (result.containsKey(i)) {
        assigned.add(result.get(i));
    }
}
```

Then we try to find the color for the node. This will be a simple search algorithm, where we iterate over and find the color:

```
// check for the first free color
int node_color = 1;
for (Int color: colors)
{
    if (node_color != color) {
        break;
    }
    node_color++;
}
```

Once we find the color -> *node_color* -> we add it to the node {node: color} and store it in a list. Other questions and concepts in this range can be solved with a similar pattern:

- Explore the nodes
- Find the neighboring colors/conditions

- Add those and store to find the next best thing

Similar questions

- [849] Maximize Distance to Closest Person
- Leetcode : Exam Room

Try to use Interesting random things to find the closest or the minimum distance things. Think of 2 pointers, multiple iterations, traversing from the back, priority queue, etc.

 We don't have to think of every question in a graph or tree as a graph or tree question. Try to find the conditions, see how you can restrict them and find the next optimal thing. Questions like these help you with problem solving. Once you start thinking of different random ways to solve a problem in your head, Woohoo. You're a better problem solver than yesterday! It's about your own journey of learning and growing, keep it up!

Credits

- Graph coloring - Wikipedia
- Scheduling (computing)

Problem 3: Detect cycle in graph

 If you have an undirected graph and want to find the cycle, what do you do? I would leave the interview and go home.

Just kidding, let's try to solve this.

We want to search through the graph, see if we find the visited node again and then return true. There's a catch here -> it's an undirected graph, so we don't know if the parent has an outgoing arrow or not -> there's no way to tell which node is the parent and parent-child nodes are always connected.

```
boolean visited[]=new boolean[V];
for(int i=0;i<V;i++){
    if(!visited[i]){
        if(DFS(adj,i,visited,-1))
        return true;
    }
}
return false;
```

So we need an additional condition here: To check for the parent node when writing Dfs code. So how do we check for the parent?

We can pass in the parent value every time we're calling the recursive function. So every time you explore a new node, you pass in that node as the 'parent' variable. The next time that parent variable gets changed to the new node.

```
static boolean DFS(adj,int head, boolean visited[], int parent){
    visited[head]=true;
    // iterate over the adjacency list
    for(int node:adj.get(head)){
        if(!visited[node]){
            // pass head to the DFS function
            if(DFS(adj,node,visited, head))
            return true;
        }
        // we check the node and parent relation here
        else if(node != parent)
        return true;
    }
    // if we don't find the cycle
    return false;
}
```

For iterative solution, we follow the simple DFS path with stack

- Add the node to the stack
- Add a condition -> In this case: check for the cycle
- Pop the node and explore the neighbors
- Add the valid neighbors to the stack

We can pass in a {root, parent} and then check once we pop it off the stack.

```
stack.push({node, parent});
while (stack.isEmpty()):
      node, parent = stack.pop();

      # we found the cycle
      if node!=parent and visited[node]:
            return true

      for neighbor in node.neighbors:
            if neighbor:
                  visited[neig]
                  stack.append(neighbor)
```

```
    # some other things
```

Read more here: Detecting Cycles in a Directed Graph

 There's a second part to this, where we find a cycle in a directed graph. Here's a nice visualizer to see that in action: Simple Recursive - Cycle Detection

Problem 4: Friend Circles

 We have a 2D matrix and we want to find friends from that matrix. All the friends who have the same number row & column wise.
Think of it as nodes connecting with each other, where we're trying to find the connected ones to form a **friend circle**. So we can iterate over the nodes, do a simply DFS, and count the number of friend circles we have. Super similar to number of islands and other DFS questions - the only trick is to **identify** this as a DFS and graph problem.

DFS would look something like this:

```
private void dfs(int[][] M, int i, boolean[] visited, int n) {
    for (int j = 0; j < n; j++) {
      if (M[i][j] == 1 && !visited[j]) {
        visited[j] = true;
        dfs(M, j, visited, n);
      }
    }
  }
```

We call this DFS function for every unvisited node, mark it visited in the DFS function, and then increase the circles.

```
for (int i = 0; i < n; i++) {
  if (!visited[i]) {
    dfs(M, i, visited, n);
    numCircles++;
  }
}
```

Complete code here: Friend circles

Problem 5: Connected components

323. Number of Connected Components in an Undirected Graph

 Super similar to flood fill, number of islands, and other questions where we have to find a **connected network of nodes** and return something from that at the end.

Let's follow our DFS pattern:

- Add the initial node to stack
- Pop from the stack, mark the node visited
- Explore the valid neighbors through some condition
- Repeat the process

Here's the catch -> we want to call dfs() for every node in the list that we have, so either we can make a separate function or just do it inside the loop. Here's how the dfs would look:

```
while(!dfs.empty()){
    int current = dfs.top(); dfs.pop();
    visited[current] = true;

    for(int neighbour : adjList[current]){
        if(!visited[neighbour]) dfs.push(neighbour);
    }
}
```

We iterate over the list of nodes and call this for every node -> marking all the connected components visited and increasing the counter once for every **new** node. Here's how the complete code looks like.

```
for(int i = 0; i < n; i++){
    if(!visited[i]){
        ans++;
        dfs.push(i);

        DFS(node);
    }
}
```

 Understand the underlying principles, visualize it in your head, and explain it to your mind before moving forward. A lot of times, you won't have to code the whole thing in an interview, but explain,

explain, explain! They want to know your approach and understanding before knowing how you write code.

Algorithms

 There are other, advanced graph algorithms which are good to know and often overlap with some shortest path questions. So here are some links you can refer to, when studying about these:

- Kruskal's Algorithm
- Detect Cycle In Graph
- Union Find Algorithm In Graph
- Prim's Algorithm

 Missing something? Email us at 30dayscoding@gmail.com and let us know if you need additional help! We're happy to help you with more resources.

Read

- A Gentle Introduction To Graph Theory | by Vaidehi Joshi | basecs
- Advanced Graph Algorithms: Dijkstra's and Prim's | by Mikyla Zhang | Medium
- 10 Graph Algorithms Visually Explained | by Vijini Mallawaarachchi

Videos

- Intro: Graph Theory Introduction
- Intro: Lecture 6: Graph Theory and Coloring | Video Lectures | Mathematics for Computer Science | Electrical Engineering and Computer Science
- Intro: Lecture 12: Graphs and Networks | Video Lectures | Computational Science and Engineering I | Mathematics
- Dijkstra's Shortest Path Algorithm | Graph Theory

Questions

- Employee Importance
- Redundant Connection
- 130 Surrounded Regions
- 721. Accounts Merge
- Leetcode-Clone Graph

- Word Search
- Network Delay Time
- Is Graph Bipartite?
- 802. Find Eventual Safe States
- 841. Keys and Rooms
- Leetcode : Possible Bipartition
- [947] Most Stones Removed with Same Row or Column
- 994. Rotting Oranges
- 787. Cheapest Flights Within K Stops
- 1319. Number of Operations to Make Network Connected

 Here are some famous topics and algorithms under graph theory, which are exciting to know about but aren't necessarily used directly in coding interviews:

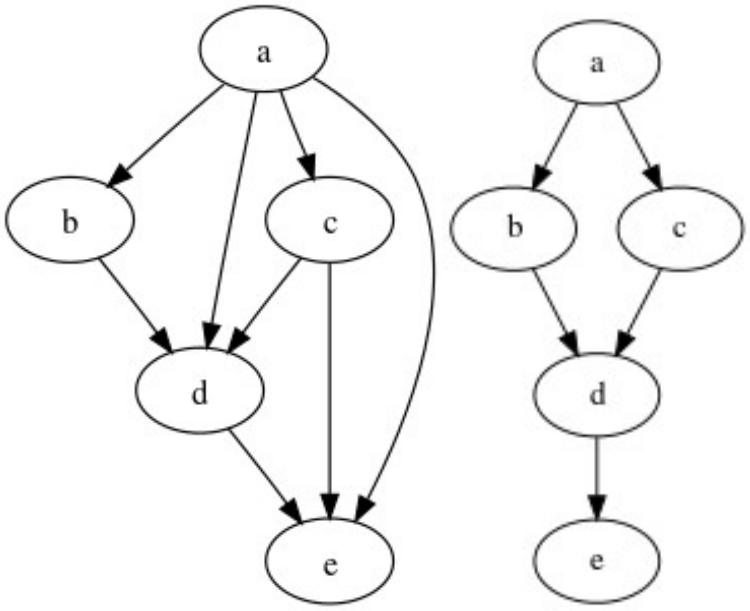
- Prim's algorithm
- Kosaraju's algorithm
- Bellman ford
- Floyd Warshall

There are also other algorithms which are discussed in the section below here.

Topological Sorting

Introduction

 The name suggests sorting, so it probably should be :P. Here's the definition: "topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering"
In simple words, we need to sort then in such a way that that the 'prerequisite' comes before all the others and we have a directed structure from one node to another.



Let's understand this with CLASSES at your school/college/university. You have to take calculus before taking advanced mathematics and you have to take basic programming before moving forward -> that's topological sorting. You can make your class schedule using this algorithm.

Here's a beautiful way to see topological sorting in action: Branch and Bound - Topological Sort

Let's convert this to code, step by step.
Firstly, we want to cover all the nodes so we can use a stack or queue. It's a DAG and we're concerned about the depth, so let's use a stack. Queue is also an option here.
We have an array given to us, let's iterate over that -> go as deep as possible and add that to our set. We want to go to the last node and begin from there. Here's the algorithm from Wikipedia.

```
L ← Empty list that will contain the sorted
elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
```

```
if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Removing edge means marking it visited and never coming back to it again. So here's what the first thing looks like.

```
for(int i=0; i < nodes.length; i++){
      if(visited[i] == false){
            visited[i] = true;
            toposort(i, visited, adj, s);
      }
}
// Print the stack here -> it's sorted!
```

We have to define toposort() which does the same thing -> takes the pointer to the very last node, adds it to the stack, marks them visited along the way, and then eventually fills up the array. Here's the toposort function:

```
toposort(int i, boolean  visited[], adj[][], Stack<> stack){
    for(int x : adj.get(i)){
        if(visited[x] == false){
            visited[x] = true;
            toposort(x, visited, adj, stack);
        }
    }
    stack.push(i);
}
```

After filling up the stack, we print out all the items from that in a sorted form. Here's a nice explanation video by WIlliam Fiset on topo sort: Topological Sort Algorithm | Graph Theory

Here's another version of the code for topological sorting: Python Topological Sorting, [Topological Sort Algorithm]

Now that we know the basics of topological sorting, let's understand it more through a question -> the most popular one: course schedule.

Problem 1: Course Schedule

207. Course Schedule

 Course schedule is an amazing problem and it resonates with every student, although no one likes to solve it. It's far from reality, because we all want to sort courses by difficulty :P (everyone loves easy courses)

Onto the question -> We have the number of courses and an array of prerequisites -> the prerequisites can be multiple for some classes. Like you might have to take CS101 for 120 as well as 130. So we need to take that into consideration as well.

Firstly, let's transform the prerequisites so that we can use them. We transform the 2D matrix into a graph-like thing where we have a key -> value thing for prerequisite -> course.

```
ArrayList[] graph = new ArrayList[numCourses];
for(int i=0;i<numCourses;i++)
    graph[i] = new ArrayList();

boolean[] visited = new boolean[numCourses];
for(int i=0; i<prerequisites.length;i++){
    graph[prerequisites[i][1]].add(prerequisites[i][0]);
}
```

We want to call DFS on all the nodes as we go and mark them visited once we cover them -> basics of topological sorting.

```
for(int i=0; i<numCourses; i++){
      if(!dfs(graph,visited,i))
          return false;
      }
return true;
```

Onto writing the dfs() function where we visit every node from that one node, mark the neighbors visited and keep a track of the eventual course structure -> whether we can take the courses or not.

The return type is a little different as we're returning true or false based on that particular node. So we visited the courses, store them in a visited array and return true if we're able to take the courses from there. We do this for all other nodes until we find a negative result. If we don't, we return true at the end.

```
private boolean dfs(ArrayList[] graph, boolean[] visited, int course){
    # if we've already taken the course, return false
    if(visited[course])
        return false;
    else
        visited[course] = true;;

    for(int i=0; i<graph[course].size();i++){
        # call DFS for the course now -> .get(i)
        if(!dfs(graph,visited,(int)graph[course].get(i)))
            return false;
    }
    # mark it visited or taken
    visited[course] = false;
    return true;
}
```

Resources

- Topological Sort Graph | Leetcode 207 | Course Schedule
- [LeetCode]Course Schedule. May 30-Day Challenge | by Yinfang
- Course Schedule | Deadlock detection | Graph coloring | Leetcode #207
- Leetcode - Course Schedule (Python)

Read □

- Definition: Wikipedia
- Visualizer: Branch and Bound - Topological Sort

Videos □

- Topological Sort Algorithm | Graph Theory
- Topological Sort | Kahn's Algorithm | Graph Theory

Questions

- Topological Sort
- Leetcode : Find the Town Judge
- LeetCode 210. Course Schedule II

Greedy Algorithms

Introduction

 Algorithms where we make choices at every step because of a reason (optimal choice) are called greedy algorithms. Like returning the max everytime in an array, or maybe returning the cheapest food near you from a list of restaurants with multiple menu items. Greedy answers can definitely work, but it might not be the most optimal thing to do wrt time and space complexity.

For instance, you have a tree and you want to find the maximum path sum of that tree. The correct solution to that would be to explore all different cases, add memoization to the logic, and finally return the max path sum from that. However, if you try to use a greedy approach right from the top, you would end up making the wrong mistake of choosing the maximum element at every level - which would be wrong. So we have to be smart about using it at the right time. Here are some sub topics which will help you understand things in a better way.

A lot of questions can be solved by sorting the input and then adding some logic to that. Let's discuss a question: meeting rooms. We have the starting and ending times for a room throughout the day. And we want to check how many people can be there at the maximum time or something -> so we sort the times, arrange the people in terms of the time and then find the maximum while iterating through the instances. Let's discuss some problems on similar concepts.

Problem 1: Merge intervals

56 Merge Intervals

 Another problem which can be solved fairly quickly after sorting is this one. Instead of comparing all the possible cases, if we just sort the inputs and then compare the last and first elements and then combine them -> it'll be much easier.

Solution: A simple Java solution

Problem 2: Meeting rooms

 We have a 2D array of the incoming and outgoing times for a person inside a room. We want to return the number of meeting rooms we would need to accommodate them.

So for eg: [[2,7], [5,7], [3,4]]. We need 2 rooms here, one for the 2,7 one and the other one for the next 2 people who come and go.

Brute force looks annoying here, we iterate over, find all possible cases, memoize something, and then finally return the optimal answer. We want the minimum rooms, so we condition something on that, and return that.

What if we change the game a little here, what if we track every time someone comes in and goes (after sorting). So if we sort this array -> we would see someone comes at 2,3,5 and someone leaves at 4,7,7.

What if -> we turn the people anonymous and every time someone comes in, we +1 the counter, and everytime someone leaves the room, we -1 the counter? Try it. While doing this, we store the max value of the counter and eventually return that max value. That's the maximum number of rooms we need.

```
def meetingRooms(start, ends):
    rearrange_rooms = [(s,1) for s in start] + [(s,-1) for s in ends]

    rearrange_rooms.sort()

    rooms=0
    max_rooms=0
    # iterate and add value to room
    for pos, value in rearrange_rooms:
        rooms += value
        # store the max rooms
        max_rooms = max(rooms, max_rooms)
    return max_rooms
```

You can also manually +1 and -1 for the incoming and outgoing people, here we just transform that into a big array with these elements: **(incoming_time, +1) or (outgoing_time, -1)**

 See how sorting + greedy helps solve some amazing problems with ease. Always think of sorting arrays if they can simplify the problems. The time complexity is NlogN for sorting which often helps in optimization.

Problem 3: Largest number

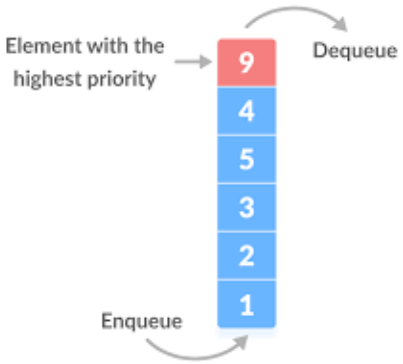
Leetcode-Largest Number

We have an array of nums, we want to make the largest number from those elements.

```
public String largestNumber(int[] nums) {
    // Get input integers as strings.
    String[] asStrs = new String[nums.length];
    for (int i = 0; i < nums.length; i++) {
        asStrs[i] = String.valueOf(nums[i]);
    }

    // Sort strings according to a custom comparator.
    Arrays.sort(asStrs, new LargerNumberComparator());

    // If, after being sorted, the largest number is `0`, the entire number
    // is zero.
    if (asStrs[0].equals("0")) {
        return "0";
    }

    // Build the largest number from a sorted array.
    String largestNumberStr = new String();
    for (String numAsStr : asStrs) {
        largestNumberStr += numAsStr;
    }

    return largestNumberStr;
}
```

Priority Queue

 Priority Queue is a big part of greedy algorithms -> tons of questions revolve around priority queues and it's important to understand how we can use them. They support insert, delete, getMax(), and other operations in logN time -> so instead of doing extra work with getting the max or min, we can use heaps and make it faster.



Here's the formal **definition**:
It's a heap based structure where we can sort and store elements in a min/max fashion so that every time we need a new element -> we just pop it off from the top instead of sorting and computing the whole thing again.
What is a heap? It's a tree like structure with these **conditions**:

- Complete binary tree
- Min heap: Every node should be smaller than the ones below it. So the element at the top (root node) will be the min one.
- Max heap: Every node should be bigger than the ones below it. So the element at the top (root node) will be the max one.

 Priority queues are also heavily used in graph theory -> where we want optimal paths or cheaper things. For eg: Cheap tickets from point A to B. We can store the edges in a priority queue as we iterate and then return the top path (with some other conditions).
Djikstra's algorithm is a common one, where priority queue is used as the main data structure. It's used to find shortest paths between nodes in a graph. Imagine an airplane flight network where we want the cheapest flight path. There are multiple short path algorithms which come under the banner of graph theory, where most of them have to do something with priority queues. So it boils down to the

fundamental knowledge of BFS/DFS and how to add some tweaks for short paths, priority queue and other things.

Problem 4: Top k elements

https://leetcode.com/problems/top-k-frequent-elements/

 We have an array full of repeating elements and we want to return the maximum frequency ones - in a row. So if we have [1,1,1,3,3,5] and k=3 -> then we return 1,3,5.

One way to do this is to count the numbers using a hashmap. Use the map something like this: map[element] = count. Sort the map using the keys, iterate over the array, and then iterate over it again -> and then return the top K ones.

Python does it short hand format but it basically means this -> **sorting(map[values])** and then we keep a counter thing for it.

```
counter = Counter(nums) # count the elements
sortedcounter = sorted(counter, key=lambda key: (counter[key], key))
# sort the array
count = 1
res = []
for i in range(len(sortedcounter)-1, -1,-1): # iterate from the back
    res.append(sortedcounter[i])
    # return when the counter reaches k
    if count == k:
        return res
    count += 1
# return res if less than k
return res
```

Bucket sort: "**Bucket sort**, or bin **sort**, is a **sorting** algorithm that works by distributing the elements of an array into a number of **buckets**."

We just iterate over the frequency map and add our items to the buckets for every frequency. Notice here, how we have buckets for frequency and every bucket is an array list where we add the key. So it's the other way round. Eventually, the **bucket** would look like this:

3 -> 1...more elements with frequency 1

2 -> 2… more elements with frequency 2

```
for (int key : frequencyMap.keySet()) {
```

```
      int frequency = frequencyMap.get(key);
      if (bucket[frequency] == null) {
            bucket[frequency] = new ArrayList<>();
      }
      bucket[frequency].add(key);
}
```

Once you've added the items, we iterate from the back and return the k elements.

```
for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {
      if (bucket[pos] != null) {
            res.addAll(bucket[pos]);
      }
}
```

We can also use a priority queue or a min heap and add/store elements in there, which does the sorting for us. Remember, whenever there's something to do with min/max or return a list of elements in some sort of order -> priority queue can be very useful. Here's how the code would look like for a heap

Make sure to understand the solution, make a small document for yourself, and your notes there. If you have any additional questions, email us at 30dayscoding@gmail.com.

```
public List<Integer> topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> map = new HashMap<>();
    for(int n: nums){
        map.put(n, map.getOrDefault(n,0)+1);
    }

    PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
        new PriorityQueue<>((a, b) -> Integer.compare(a.getValue(),
b.getValue()));
    for(Map.Entry<Integer,Integer> entry: map.entrySet()){
        minHeap.add(entry);
        if (minHeap.size() > k) minHeap.poll();
    }

    List<Integer> res = new ArrayList<>();
    while(res.size()<k){
        Map.Entry<Integer, Integer> entry = minHeap.poll();
        res.add(entry.getKey());
```

```
    }
    return res;
}
```

Problem 5: Coin calculator

Part I

 Imagine you have to pay some amount for your food, let's say $75 and you have a set of denominations with you -> {1,2,10,25}. What's the minimum number of coins you would use to fulfill that $75 order? Brute force would probably lead us to trying every possible way and then returning the minimum number of coins. We could memorize repetitive things, and hence use dynamic programming to solve the question. Can we do something better?
We can use a greedy approach here, to simply use the maximum denomination first, so starting off with $25 notes and using those until we can't and then using the others.

Part II

Here's the catch, for example you have {$13, $25} bills and the total bill is $26. What do you do? If you use a greedy approach, you would end up using the $25 and then leave the $1 behind. You gotta pay that, or wash the dishes. Here's where we would need to explore other options and the backtracking hits us. Make sure you see both the parts here, and not just one.

Read

- Non Overlapping Intervals. This week I encountered many interval… | by Osgood Gunawan | The Startup
- When to use Greedy Algorithms in Problem Solving

Videos

- Interval Scheduling Maximization (Proof w/ Exchange Argument)
- 3. Greedy Method - Introduction

Questions

- Leetcode-Largest Number
- Graph Coloring Problem – Techie Delight
- 435 Non-overlapping Intervals

- 787. Cheapest Flights Within K Stops
- Greedy

Tries

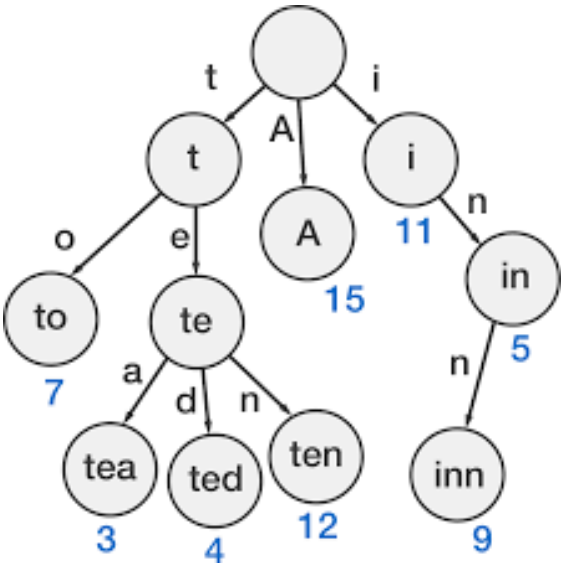
Introduction

 Tries are also a type of **prefix** trees which are tree-like structures to store strings.

Let's start with a question: You have 2 strings and we want to find the common letters in it.

The first brute force way is to iterate over the first string, add the letters to a set -> then iterate over the next string and see all the elements that are in the set. You could also do things like string2.contains(char) -> but it's the same thing wrt time complexity.

We can insert and find strings in O(L) time, where L is the length of the string. Another use case can be to print the characters in order.



Problem 1: Implement Trie

Leetcode 208. Implement Trie (Prefix Tree)

 We're going to implement the trie here and understand how it works. A lot of times -> you would have to implement this on the side and then use it in a question, so we're going to discuss a question as well. At the same time, the question could also be "search for a letter" -> where we can just use the search function.

First, we want to decide how the Node class looks like. Every node needs to hold a map of the children and a boolean which tells if it is the last node (leaf node / last character):

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.isLast = False
```

We need the Trie class now. The major functions are insert, search, startsWith -> where we can also add more -> delete, findChar, etc. Let's begin the insert function.
Here's a great article before moving forward: [Trying to Understand Tries. In every installment of this series… | by Vaidehi Joshi | basecs]

Insert

 We want to insert a character at the very end of the trie. The first part of that is iterating down and finding the last character (through the isLast field of TrieNode) and then add the character to the map. The letter which we'll add will be a TrieNode() and not just a character. Every node is a TrieNode -> which has those 2 things.
Here's how we do it

- Iterate over the word - every letter
- Iterating forward -> node = node.children[letter]
- We add the letter there -> node.children[letter] = TrieNode()

```
def insert(self, word):
        node = self.root
        for letter in word:
            if letter not in node.children:
                node.children[letter] = TrieNode()

            node = node.children[letter]

        node.isLast = True
```

Searching

 We want to search for a character or stream of characters in a string.

Here are the steps:

- Iterate over the letters
- If the letter is not in node.children -> return false. Remember, node.children is a dictionary of the letter mappings for the children, -> so it should be there.
- Iterating forward -> node = node.children[letter]
- If we reach the end without returning false, we return if it's the last element or not -> using the isLast class field.

```
def search(self, word):
    node = self.root
    for letter in word:
        if letter not in node.children:
            return False

        node = node.children[letter]

    return node.isLast
```

Starts With

 We want to return true if the string (prefix) is at the start of a word. We can simply use the class field to our advantage and find the right answer here.
Here are the steps

- Iterate over the letters
- If the letter is not in node.children -> return false. Remember, node.children is a dictionary of the letter mappings for the children, -> so it should be there.
- Iterating forward -> node = node.children[letter]

```
def startsWith(self, prefix):
    node = self.root
    for letter in prefix:
        if letter not in node.children:
            return False
        node = node.children[letter]

    return True
```

Resources

- Trie Data Structure - Beau teaches JavaScript

- Trie Data Structure Implementation (LeetCode)

Questions

- Leetcode 208. Implement Trie (Prefix Tree)
- Leetcode 139. Word Break
- Leetcode Word Break II
- Leetcode 212. Word Search II
- Leetcode 1032 Stream of Characters
- Leetcode 421 Maximum Xor of Two Numbers in an Array

Additional Topics

 These are some random mixed questions, which will teach you something new to learn. We should never solve a question expecting it to come in our interview (even something similar), but to learn something new from it!

Remember, we're not trying to solve hundreds or thousands of questions, but to

- Understand the concepts
- Build problem solving skills
- Enjoy our time with questions
- Become a better developer

Kadane's algorithm

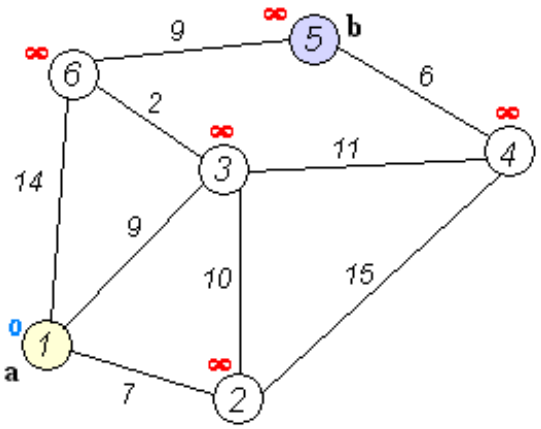
Wikipedia: Maximum subarray problem

 It's used to solve the maximum subarray problem and the concept is to keep a track of the sum as you go -> and change it to 0 when it's negative. (so you're positive at the very least). An edge case is all negative numbers -> where you return the min of those.

-2	1	-3	4	-1	2	1	-5	4

Djikstra's algorithm

 Djikstra's algorithm is a shortest path algorithm, where priority queue is used as the main data structure. Imagine an airplane flight network where we want the cheapest flight path from point A to B. There's also the shortest-path-tree which basically returns a tree with lowest cost from one node to another. So instead of just a short path from A to B, we do it for all the nodes in the graph.
Here's a nice video about this algorithm: [Dijkstra's Algorithm - Computerphile]



Let's understand the algorithm:
The basic understanding is that we want to visit every node, mark them visited, calculate the cost until then, and finally return the shortest path. We can simply do a BFS (shortest path!) and use a Queue for that.
Here's the catch -> if we use a simple queue, then it would be hard to get the minimum element when we pop it off. So instead, we can use a priority queue -> where every time we pop something off, we get the minimum element.
Following the BFS principles, we add the node to the queue, pop it off, explore it's neighbors + do some calculations for the route + mark them visited, and then repeat. Here's a nice **visualization** of the dijkstra's algorithm: [Greedy - Dijkstra's Shortest Path]

```
function Dijkstra(Graph, source):
```

```
   dist[source] ← 0                           // Initialization

   create vertex priority queue Q

   for each vertex v in Graph:
       if v ≠ source
           dist[v] ← INFINITY                 // Unknown distance from source to v
           prev[v] ← UNDEFINED                // Predecessor of v

       Q.add_with_priority(v, dist[v])


   while Q is not empty:                      // The main loop
       u ← Q.extract_min()                    // Remove and return best
vertex
       for each neighbor v of u:              // only v that are still in Q
           alt ← dist[u] + length(u, v)
           if alt < dist[v]
               dist[v] ← alt
               prev[v] ← u
               Q.decrease_priority(v, alt)

   return dist, prev
```

Credits: Wiki

AVL Trees

In a normal BST, the elements in the left tree are smaller than the root and the right ones are bigger than the root. It's very useful for sorting and we can find the element in O(logN) time. There's a catch -> for the given nodes in an array -> there's a format that we have to follow which generates multiple binary trees with different structures.

[1,2,3] can generate a binary search tree with the root 3, left child 2, with left child 1 -> this is not what we wanted and hence we need something better.

AVL trees have a condition, the balance factor has to be in the range {-1,0,1}. So it's a self balancing binary search tree.

Resources:

- 10.1 AVL Tree - Insertion and Rotations
- AVL tree - Wikipedia
- AVL Tree Visualization

Sorting

Sorting is super important as a concept but not super important in terms of knowing everything about them. For questions, you can use .sort() to sort whatever you're using, and rarely you'll be asked to actually implement the underlying algorithms. Read more here: Sorting algorithm

Here's a great visualizer for all sorting algorithms: Sorting Algorithms Animations
Another one more: Brute Force - Bubble

More

If you think we should add a section or anything in general, please write to us at 30dayscoding@gmail.com

---

Additional Awesomeness

Questions

- 150 Questions: Data structures
- Striver SDE Sheet

Blogs

- How to make a computer science resume
- How to apply for Internships and Jobs
- How to take a technical phone interview
- How to find coding projects + people
- How to learn a language/framework from scratch
- How to revise for Coding Interviews in 15/30/45 days
- Everything about a technical internship
- How to choose the right university (USA)
- How to Get an Internship in MNC | Board Infinity

Youtubers

DSA

- WilliamFiset (English)
- IDeserve (English)
- Kevin Naughton Jr. (English)
- Back To Back SWE (English)
- Tech Dose (English)
- Codebix (Hindi)

Competitive coding

- SecondThread
- Errichto's Youtube channel
- William Lin

Websites

- 30DaysCoding
- Geeks for geeks
- Leetcode Patterns – Medium
- Interview Question