**Algo Expert Sum:**

**4 Number Sum**:

Function that takes in a non-empty array of distinct integers and an integer representing a target sum.

Find all quadruplets in the array that sum up to the target sum and return a 2d array of all these quadruplets in no particular order.

If no 4 numbers sum up to the target sum, the function should return an empty array.

To create 2D array with variable number of rows and columns, we can use List<Integer[ ]> or List<List<Integer>>

Naïve solution: 4 for loops.

O(n^4)

A quadruplet can be expressed as a pair of numbers. We can turn a quadruplet into a pair of numbers.

Can reduce to 2 number sum.

x, y, z, k

P: x, y

Q: z, k

In the hash table, we can store the number P and can map it to (x, y).

x and y are the numbers that generate P.

The number P can be obtained by summing up a bunch of pairs of numbers.

{ 6: [ [4, 2], [7, -1] ], }

We should store an array of pair of numbers for a particular number P.

When we are going to be generating our pairs of numbers and then our quadruplets. We have to prevent double counting the quadruplets or pairs.

Double counting can happen like this:

13: [7, 6], 3: [4, -1]

And 10: [6, 4], 6: [7, -1]

In 2 number sum solution, we iterated through the array in one single pass.

Here, we can have 1 for-loop and then 2 inner for loops. These will be used strategically to not double count quadruplets.

We first iterate one by one through all the numbers in the array.

The first inner for loop will simply iterate through all the numbers that come after our current number.

Second for loop will iterate through numbers before the current number.

In the first for loop,

For these numbers we will generate the sum of this number and our current number.

Suppose the current number is 4. In the 1st for loop, we go through the numbers {-1, 1, 2}

P = 4 + -1

4 + 1

4 + 2

We check whether (targetSum – P) is in the hash table or not.

If it is, then we have found the quadruplet but if it is not, then we will not add that sum to the hash table yet.

Using the 2nd for loop, we iterate through the numbers that comes before our current number.

The numbers that come before 4 are {7, 6}

Here, we try to generate Q

Q = 4 + 6

4 + 7

These sums are the ones that we store in our hash table and the ones that we eventually use. That will allow us to not duplicate our quadruplets.

**IMP NOTE**:

In the collections we always use object types and not primitive types.

If we have List<Integer[ ]>

Arrays are of fixed size. So, to iterate over arrays we should use for-each loop.

Elements have to added in the array at the time of creation.

Adding elements into a list can be done anytime.

**SubArray Sort**:

Return an array of starting and ending indices of the smallest subarray in the input array that needs to be sorted in place in order for the entire array to be sorted.

How to determine which array to be sorted?

In a case, left part and right part of the array could be sorted. In the middle, we could have unsorted numbers.

If we get number in array which is not in sorted order, we will have 2 numbers which will not be in sorted order.

IMPORTANT operation:

Create a subarray from an existing array.   
Arrays.copyOfRange()

There are always gonna be atleast 2 numbers which are not sorted if one of them is not sorted.

If one number is out of order, it could mean that a huge subarray needs to be sorted.

Suppose we have [ 1, 2, 3, 4, 5, -1 ]

-1 is unsorted with respect to 5. So 5 is also unsorted.

If we just sort 5 and -1, then our 5 is good but -1 still is not good.

Here we will have to sort the entire array.

The subarray we have to sort will depend on the final position of the unsorted number.

There could be multiple unsorted elements.

Starting index of the subarray to sort will depend on the minimum unsorted number.

Find all unsorted numbers and remind ourselves that if there is 1 unsorted number, then there are atleast 2 unsorted numbers.

Out of these atleast 2 unsorted numbers, we need to find greatest and smallest one.

Then we have to find their final positions in the sorted array.

Let’s take the example,

[ 1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19 ]

Compare 1 with 2.

Compare non-edge numbers with previous and next number. Check if that number is sorted or not.

Compare 2 with 1 and 4.

If we are talking of sorted, either take increasing or decreasing, not both. (Ask the interviewer)

In the group of unsorted numbers, find the maximum and minimum numbers.

Find the final position of these minimum and maximum numbers in our final sorted array.

Just save these indices.

O(n): There could be multiple array traversals.

Do not keep just 1 array traversal in mind.

**Largest Range**:

Write a function that takes in an array of integers and returns an array of length 2 representing the largest range of integers contained in the array.

Assume only 1 largest range.

A range of numbers is defined as a set of numbers that come right after each other in the set of real integers.

Numbers don’t need to be sorted or adjacent in the input array in order to form a range.

Range [2, 5] means {2, 3, 4, 5}

The range does not need to have numbers right next to each other in the array.

[ 1, 11, 3, 0, 15, 5, 2, 4, 10, 7, 12, 6 ]

The largest range of integers in this array is [0, 7].

This array has all the numbers in this set { 0,1, 2, 3, 4, 5, 6, 7 }

and hence in the range [0, 7]

There is another range [10, 11, 12] but it is smaller than the range [0, 7]

Obvious way: Sort the array.

Hash table is useful when we are not expecting to iterate it in sorted order, or compare it quickly to another hash table.

Store all of our numbers in a hash table.

We can iterate through the numbers in our array.

When we are at a current number, then we can check whether numbers smaller than this current number and also numbers greater than this current number are present in the hash table.

We can check for an element in the Hash table in constant time (O(1)).

Map the numbers to a boolean or anything that makes sense.

First we use linear time to put the elements into the hash table.

Then we again iterate over the array.

O(n) time and O(n) space.

We don’t have to store the entire range. We can just store the starting and ending number.

map.replace(..)

map.get() returns null if the element is not present in the map.

Boolean.***TRUE***.equals(map.get(num))

Or can use

if( map.get(num) != null ) )

better is to use containsKey()

**Min Rewards**:

Can assume that the scores are unique.

Given an array of scores.

All students must receive atleast 1 reward.

Any given student must receive strictly more rewards than an adjacent student with a lower score and must receive strictly fewer rewards than an adjacent student with a higher score.

Find minimum number of rewards.

We cannot sort else the order will change.

O(n^2) time, O(n) space

O(n) time, O(n) space.

Questions to ask/clarify:

Can we duplicate numbers? No

Are we only given positive integers? Yes

Order matters

Minimum number of rewards

**Naïve approach**: O(n^2) time, O(n) space.

We can iterate through the array.

Check if the current number is greater than the previous number. Then increment by 1, else backtrack to previous number and increment that by 1.

Do not check like this: current number greater than next element. (We do not know the value of the next element, only know about previous values).

Starting at index 0, assign it a reward of 1. And then move on.

We have to backtrack only when the previous number’s cost is 1 as the current number’s cost cannot be less than 1.

If the current number has a cost is say 5 and if we find a smaller number next to it, then we can happily set to it minimum value of 1. If we try to set it to 2 then it could violate other numbers.

**Note**: If we try to find the minimum number’s index in the array and then try to iterate left and right, then still we will have a case which will require backtracking.

[ 1, 7, 8, 3, 2 ]

[ 1, 7, 3, 2 ]

**Another approach**:

Useful technique with arrays:

Peaks and values, high and low points.

Local maxes and local mins

In the array of scores,

There are trends in the scores.

Scores can be depicted in sort of linear graph where we start .

We have got local minimums and local maximums.

Every local min will get 1 reward. They are smallest in their neighbourhood.

Local maximums will have most rewards in their neighbourhood.

Start from the local mins and expand on both sides till the peaks. Increment the rewards till we reach the peaks.

To assign value for a peak, we can take the

1 + max(left, right)

Or if we are only storing minimas, then if we reach same number twice (peak), then can take max(current, 1+neighbour)

Peak is greater than its adjacent elements.

We will be going to every number exactly once.

**Method 3**: Cleanest

Expanding to the left and right of the local mins does not have to start at the local mins.

We can actually do that by iterating through the array once from left to right and once from right to left and we will end up doing the exact same thing when we expand to the left and right of the local mins.

Left to right iteration:

Compare current number with the number before it. This sort of mimics expansion to the right of local mins.

Right to left:

Compare current number with the number after it.

This mimics left expansion from local mins.

We can mimic this behaviour without having to start from local mins.

When going from left to right, we only care if the current element is greater than the previous element, else we continue.

When going from right to left, we only care if the current element is greater than the previous element, else we continue.

**IMP NOTE**:

Again note that linear time does not mean 1 for loop. We necessarily do not to write the algorithm in 1 for loop.

Initialize the rewards array be 1 by default.

In Method 3, we need to do Math.max() in the 2nd for-loop. It does not matter whether the iteration order of this 2nd for loop is from left to right or right to left.

**In Method 2**:

**IMP NOTE**:

If we start exploring from the left most minima and go rightwards, then we need to do Math.max() when traversing from left to right.

This will be the most likely case.

If we start exploring from the right most minima and go leftwards, then we need to do Math.max() when traversing from right to left.

**ZigZag Traverse:**

Given 2d array.

Zigzag order starts at the top left corner of the 2d array, goes down by 1 element, and proceeds in a zigzag pattern all the way to the bottom right corner.

O(n) time and O(n) space.

There are only 2 directions that we will be travelling in the 2d array.

When we are doing zigzag, we are either going down or either going up.

Diagonally Up: Move 1 row up and 1 column to the right

Diagonally Down: Move 1 row down and 1 column to the left.

Keep track of direction.

We continue to go in diagonal direction till we reach a boundary number.

When we reach a boundary point, we either go down or to the right to the next number and then go diagonally from that number.

How to identify whether to go right or down?

After going diagonally, when we reach a boundary number, then we go right or down. We go right or down towards a number which is itself a boundary number.

And then we go diagonally again from that number.

Another thing we can do is store the direction of traversal. If we go diagonally up, then when we reach a boundary point, we go right.

If we go diagonally down, then when we reach a boundary point, we go down.

Going right or down only happens on the perimeter.

Important technique to convert 2d array into 2d array:

Integer[][] array = { {1, 3, 4, 10}, {2, 5, 9, 11} , {6, 8, 12, 15} , {7, 13, 14, 16 } } ;

List<List<Integer>> list = **new** ArrayList<>();

**for**(Integer[] item: array) {

list.add(Arrays.*asList*(item)) ;

}

If we are at last row and going diagonally down, then we switch direction and go to the right.

If we are going down and are at column 0, then we go down.

If we are going up and are at last column, we go down.

If we are going up and are at first row, we go right.

We have to start at the top left corner.

When going down,

2 interesting spots for our current element: First column or last row.

We update our direction. (This update applies on next element)

When going up,

2 interesting spots for our current element: First row or last column. We update our direction.

The change of direction applies to the next element.

When we move from 1 to 2.

1 has direction of down and it is in column 0, so we update the direction and move one element down. This change of direction applies to 2.

Now 2 has a direction of Up. 2 is neither in row 0 nor in last column. We move up and do not update the direction.

If the current element is in row 0 and last column and is going up, only then we update the direction.

If the current element is in column 0 and last row and is going down, only then we update the direction.

Direction of the current element is decided by the position of the previous element and not the direction in which we go from this element.

Element 13 has direction “down” even from it we go right (up) towards 14.

**IMP Edge Case**:

When going down,

When we are at last row and first column, then we have to go right. We cannot go down. So add a check for last row before the first column.

I may not have caught this if I had not tried to solve it on my own.

**Same BSTs**:

An array of integers is said to represent the Binary Search Tree (BST) obtained by inserting each integer in the array, from left to right, into the BST.

Write a function that takes in 2 arrays of integers and determines whether these arrays represent the same BST. We are not allowed to construct any BSTs in our code.

We will be allowed to construct the BSTs on the whiteboard during the interview.

First value of the two arrays is going to be the root value of the BST.

Look at the lengths of the array.

Different length arrays could not represent same bst.

Remove the root node. We want to find the array which represents the right subtree (right bst) and which represents the left subtree (left bst).

We can try to find all the values smaller than the first value (root value) and values greater or equal to the root.

Create 2 subarrays which represent left and right subtrees respectively.

Then we perform the same logic on these left and right subarrays.

Compare the lengths of 2 left and 2 right subarrays.

Interesting observation: For 2 equal bsts the order in which numbers greater than root and smaller than root occur is same in both the arrays.

But this won’t cover all cases.

We cannot naively compare these 2 arrays.

We separate the subtrees into left and right subarrays.

If the first element in both left and both right subarrays are equal, we continue with our recursion.

With Recursion, think of base case.

2 recursive calls.

Repetition: Use a function.

When the 2 arrays will be empty at the same time, we return true.

Time: O(n^2)

Both the arrays are of length n.

On both arrays, we are doing O(n) + O(n) loops.

We also take O(n) for the subtrees, that is keep doing O(n) n times.

We will be doing validation of sub bsts n times.

Left and right subarrays represent sub BSTs rooted at index 0.

Space: O(n^2)

Creating new subarrays. Creating in total O(n^2) auxiliary memory.

In 1 single recursive call, it is O(n). 4n

Can reduce the space to equal to depth of the bst being represented. O(d)

For balanced bst: O(logn)

Can do better in terms of space.

We always pass in the 2 same arrays. We also pass in the pointer of the root we are at.

**Max Path Sum in Binary Tree**:

Return max path sum of a binary tree.

Path is a collection of connected nodes in a tree where no node is connected to more than two other nodes.

The source node is not fixed. It could be any node.

Can the nodes value be negative? Yes.

All positive

Mixture of positive and negative

All negative.

Have to go all the way down to leaf nodes.

We can only go down.

Cannot add everything blindly as there could be negatives.

If max path sum has a triangle shape, we cannot add it to the current node.

One way we can deal with this is by computing at every node, not just the naïve max path sum at that node but rather computing a few different things, one of which is going to be the max path sum as if we were a branch, as if we were just a line and not in the form of triangle that goes both sides.

However, it might be the case that the max path sum is a triangle.

We also want to keep track of this sum somehow.

We will compute maximum path sum for a node and only using branch, not using triangles.

We will also compute the maximum path sum as if we were a root node and as if we could be a triangle.

Compute running max path sum.

We are saying that our max path sum will return 2 things.

First thing it will return is maximum sum at our given tree that includes our tree’s current value or node’s current value but that is only a branch and not a triangle.

Call stack is reused.

Never add triangle sums to values of the current node.

(Max path sum as a branch, Running Max path sum)

Total 6 cases.

**Note**: A branch starting from root node does not need to go till the leaf node.

The parent needs MSB value to add it to it’s value and compare with RMPS.

Only thing a parent needs to do is find the max path sum in the path that includes itself. And then it compares this with existing RMPS.

When a data structure is associated in a recursion, it is usually returned or passed in between the recursive calls.

Head Recursion.

Builds the base cases first.

Why we store the 2 values?

We pass them to the parent node and this helps the parent to create these 2 values for itself.

The code that we wrote was also valid for the base cases. Base case values were initialized.

**Max Sum Increasing Subsequence**:

Write a function that takes in a non-empty array of integers and returns the greatest sum that can be generated from a strictly-increasing subsequence in the array as well as the array of the numbers in that subsequence.

Ask this to the interviewer:

Can assume that there will be only one increasing subsequence with the greatest sum.

Not easy question

Complicated when the array is mixed.

Can miss a bunch of just smaller numbers which are close to a bigger number.

There smaller numbers could make a global sum.

A subsequence can be obtained by removing some of the elements from a sequence and preserving the order of the elements.

We can use dynamic programming.

We can build an array of the same size as our input array.

At each index, we can store the value: max sum increasing subsequence generated till that index and including that element.

Building solutions by combining smaller subproblems.

We also need to build the optimal solution rather than just finding the optimal value.

This problem is similar to Longest Increasing Subsequence.

Re-iterate through previous numbers and check whether they are less than our current number.

Build another array which will keep track of our sequences.

At each index in this array, we will store the index of the previous number in the increasing subsequence that ends at that index.

For index 0 there is no previous index as the max increasing subsequence is the number at index 0 itself.

We can store null at index 0.

To build a solution for a particular index, we check sequences[] array for this index.

Then we check the index equal to the value of this index.

At any given point when we are iterating through the array, we will declare our current number.

The maximum sum is not necessarily at the last index of sums[] array.

We can find the maxSumIndex inside the double for loop itself instead of creating a new for loop.

**Longest Common Subsequence:**

Write a function that takes in 2 strings and returns their longest common subsequence.

Single character in a string and the string itself are both valid subsequences of the string.

Can assume that there will only be one longest common subsequence.

When there is a mismatch, then the longest common subsequence may have one of the characters but not both.

lcs of a string with another empty string is an empty string.

So many ways to solve the lcs problem.

Can take the cache as int[][] or String[][]

To fill each entry of the 2d array, we are checking previous values. These are not constant time operations.

When we do have letters equal to each other, we look diagonally.

For example, for the last entry we do ‘XYZ’ + ‘W’

This is not a constant time operation.

Time complexity for this would be whatever the length of the resulting string is, O(4).

That is, time taken to form ‘XYZ’ + Time taken to form ‘W’

The worst case is when we have strings that look like this:

AAAAA

AAAAAA

At each entry of the 2d array, we would be doing concatenations. At each of them we would be doing not constant time operation.

These operations converge to whatever the time complexity is of the final operation, which is going to be the length of the lcs.

Time = O(mn\*min(m, n)) = Space

Can we do better? Yes, we can.

We can improve our space complexity by noticing that at any given point in our array,

We only look at the three values. Current place which we are filling, the value to the left of it and value above it.

When we are in a row, we are only looking at values in that row and values in the previous row.

Instead of storing entire 2d array, we can store the 2 rows.

In another approach, we can completely eliminate

**“min(n, m)”** part of space time analysis.

We can reduce our space-time analysis to O(nm).

The way we can do this is by not storing lcs’s in each of these entries.

**IMP**:

Instead of storing the whole lcs’, we can just store pointers, booleans or something which tells us whether or not we will be using the letter at that index.

The pointer will point to the previous location from which the previous letter of the lcs comes from.

Can create a function to backtrack through our 2d array and find the lcs.

In code,

We can use a 3d list or array

Each entry of the 2d cache is independent and a list of characters is stored at each entry.

Multi-dimensional array or list is like another list inside a list index.

We have to initialize the rows as well as the individual lists as they are object.

List<List<List<Character>>>

In general case of LCS problem, we do not know beforehand about whether the bigger string is on row side or on column side. That is why we do Math.max() when there is a mismatch.

If we know about the bigger string, we can avoid the Math.max() operation.

**Imp**:

Complexity of String concatenation ?

For every call to String.concat(), a new string instance is created.

To create that instance, the contents of the previous string needs to be copied to the new one plus the contents of the concatenated string.

This is done for every string in the collection. So if you look at in the terms of how many strings are concatenated (and not the characters copied), you're copying the contents of the n strings by the time you reach the nth iteration and all that matters is the worst case (the last iteration). Therefore it has O(n^2) performance.

Basic Concept: String is immutable.

StringBuilder is mutable.

**Min Number Of Jumps**:

Famous Interview Question

Non-empty array of +ve integers where each integer represents the maximum number of steps we can take forward in the array.

Write a function that returns the minimum number of jumps needs to reach the final index.

Jumping from index i to i+x always constitutes 1 jump, no matter how large x is.

[3, 4, 2, 1, 2, 3, 7, 1, 1, 1, 3]

From index 0, we can take a maximum of 3 steps forward.

If the value at current index is k, then

the minimum value of current index will depend on next k indices.

Using bottom up approach, we fill the cache from right to left. Start filling from array.length-2

In top down, we call the recursive call on index 0 and then it goes all the way till it hits the base case, last index.

**1d cache**

cache[i]: minimum number of jumps required to go from i to last index of the array.

The base case value will be at last index of cache[].

In top down approach, we may need to initialize the cache to some default value, say -1 in some cases.

O(n^2) time, O(n) space

Can do better in O(n) time and O(1) time.

Here the array has positive numbers which is very important condition.

We cannot jump backwards.

**Another interesting approach**:

jumps[]

At any index we can store the minimum number of jumps required to go from starting index(0) to that index.

From 0 to 0: 0 jumps.

For an index, iterate before all previous indices and see if we can update minimum number of jumps.

In this, we can fill the cache from left to right.

**Another approach**:

When we iterate over indices, we know what the maximum reach in the array is.

[ 3, 4, 2, 1, 2, 3, 7, 1, 1, 1, 3 ]

When we are at the first index, the max reach (farthest we can get) is index 3.

From index 1, the max reach is 5.

At each index, we know what our max reach is.

When we are at first index, we know that we can take 3 steps until we have to take a jump.

When we are at 3, until we reach 1, then only we know that we have to take a jump.

At index 0, maxReach = 3, steps = 3

When we iterate through the array, we update the values of maxReach and steps.

When we iterate, we take one step forward. We reduce the number of steps that we have.

What happens if steps reach 0.

We reach steps = 0 at index 3.

When we run out of jumps, then we have to make a jump. Increment ‘jumps’ variable.

When we are at index 3, our maxReach is index 5. There is no way we can go farther than index 5.

At index 3 we know that we have run out of steps and have increased our jumps.

At index 3 how many steps do we have until we have to make another jump ?

We have 2 more steps to reach our maxReach.

**IMP**:

The code for this is smartly written to handle all cases for maxReach and steps.

In code, we iterate from 1 to array.length-2 as we return ‘1+jumps’ in the end and this covers all cases for possible end values of maxRange.

For i >= 1, maxRange > steps as minimum value array elements can be 1.

If we want to build solution, then there could be multiple solutions. We can build in O(n).

**Water Area**:

Given an array of non-negative integers where each non-zero integer represents the height of a pillar of width 1.

Imagine water poured over all of the pillars, write a function that returns the surface area of the water trapped between the pillars viewed from the front.

Can use DP here. There is a pattern or formula here to calculate the area.

Calculate how much water will be contained right above an index.

The water at an index will depend on the height at that index and will also depend on the position of that index. Is this index trapped between 2 large pillars or is it not trapped between 2 large pillars.

Excess water which spills over the pillars will not be counted in our final solution.

For index 2, there is a pillar to the left of it and also pillar to the right of it.

For any given index in our array, whether that index be a pillar or not a pillar, we need to see what is the tallest pillar to the left and right of this index.

The water will be trapped between those 2 pillars.

We compare the height of the 2 pillars and then subtract it with the height of the current pillar to get the height of the water for this index.

We can iterate from this index till the index of the biggest pillar on the right if the height of all indices in between these 2 pillars is smaller than the minimum of the 2 pillars.

Build an array of the same length as our input array.

Build an array for leftMax and rightMax.

leftMax: Iterate forwards, rightMax: Iterate backwards.

To calculate water contained at each index, can create another array.

Take the smallest between leftMax and rightMax and then subtract with current height.

We do not find leftMax and rightMax in a single iteration because finding rightMax along with leftMax would require additional O(n) for each index leading to O(n^2).

Finding rightMax by iterating backwards only leads to O(n)

Another example where work is split instead of doing it in one go. And also a case where both forward and backward iteration of an array is helpful.

We cannot ignore the first and last indices of the array.

Water poured over first and last indices will spill but still they can affect the leftMax and rightMax values and thus affecting the total water area.

Can store the individual sums for each index and also calculate the running sum.

We can do this using only a single array. First store the leftMax in this array. And then in another single iteration, iterate backwards, find rightMax and apply the logic to find the running sum.

Here an answer to a subproblem does not depend on previous problems but on leftMax and rightMax. We find leftMax and rightMax using the given array heights.

Interesting version of DP.

**Knapsack Problem**:

We are given array of arrays, where each subarray holds two integer values and represents an item; the first integer is the item’s value and the second integer is the item’s weight.

We are also given an integer representing the maximum capacity of a knapsack that we have. Our goal is to fit items in our knapsack without having the sum of their weights exceed the knapsack’s capacity, all the while maximizing their combined value.

Write a function that returns the maximized combined value of the items that we should pick as well as an array of the indices of each item picked.

We can assume that there will only be one combination of items that maximizes the total value in the knapsack.

All possible combinations having

total capacity<= capacity

Then find the one with the maximum value.

Backtracking

Famous problem in the area of Algorithm Analysis.

First solve this using recursion.

Also see this:

<https://medium.com/swlh/solving-the-target-sum-problem-with-dynamic-programming-and-more-b76bd2a661f9>

Can store a 2d array.

See the solution using a HashMap.

We can pick items in any order.

First Row: What would be the maximum value that we can store at each of the bags of capacities 0…10 if we had no item.

2nd row: Taking first item.

3rd row: Taking first two items. Not necessarily have to take both the items. Can also take either or none.

Empty item is not really an item. It is just a base case.

**Questions to ask to the interviewer**:

Can the item values be negative?

Can the weight of an item be 0.

What to return when we have no item? 0

If there is exactly one solution, then we may not have to worry about equality.

**NOTE**:

We cannot build the solution alongside with building the cache.

Because the currently added item may not be present in the optimal solution as we have the freedom to pick items randomly.

In Knapsack problem, memoization (Top Down) does better than bottom up approach.

Well I believe theoretically you should be able to solve a DP problem with either approach. However, there are instances when bottom up approach can become too expensive.

Consider a knapsack problem with the knapsack\_size = 200,000 and the num\_items = 2000. To fill in a two dimensional DP table with just ints is not going to be possible.

You'll exhaust the main memory of an ordinary computer. We do not require to fill in all the entries in a table to achieve the desired final computation. A recursive top-down approach is far superior in a case like this.

Also see TheoryNotes\_Programming.docx

**Disk Stacking**:

Given an array of non-empty array of arrays where each subarray holds three integers and represents a disk.

These integers denote each disk’s width, depth and height respectively. Our goal is to stack up the disks and to maximize the total height of the stack. A disk must have a strictly smaller width, depth and height than any other disk below it.

Write a function that returns an array of the disks in the final stack, starting with the top disk and ending with the bottom disk. Note that we cannot rotate the disks, the integers in each subarray must represent [width, depth, height] at all times.

We can assume that there will only be one stack with the greatest total height.

**IMP NOTE**: Also measure the time of my function.

This is a variation of LIS (Longest increasing subsequence) problem.

The disks can be given in any order in terms or dimensions.

To make it easier to solve and make our algorithm a lot more efficient, the first step we can do is to sort our input array in ascending order by one dimension, lets pick height.

Brute Force: There would be many solution.

Build a solution array, heights[] of same length as our input array.

This solution will hold at each index, maximum height of a tower that we can build by putting the disc at index i at the bottom of the tower.

A particular disc can serve as the bottom of many towers. We store the maximum height possible for this index when this disc is at the bottom.

We can initialise the heights[] array by initializing the heights by mapping them with the heights of ours discs.

This initialization means that we only take this disc and we have a tower with this disc only.

We will progressively update the heights[] array to see if we can get taller heights.

Let’s iterate through our input array of discs.

When we are at a particular disc, the discs that come after this will have height greater than or equal to this disc. (Since the input array is sorted by heights).

We cannot stack a disc that comes after the current disc.

Constraint:

A disk must have a strictly smaller width, depth and height than any other disk below it.

heights[i]: Maximum height of a tower that we can build by putting disc at ith index in array[] at the bottom of the tower.

When we are at an index, we iterate backwards because we assume that this current disc is at the bottom.

This is where sorting the array with respect to one dimension helps us.

List<Integer[]> is useful when size of each list item is fixed.

O(n^2) time and O(n) space.

To build the solution, we can create a sequences[] array and store the previous pointers.

The solution of current subproblem is dependent on previous subproblems, so we can build the solution easily.

Sorting can be done in nlogn which can be eliminated by n^2.

**Numbers in Pi**:

Given a string representation of the first n digits of Pi and a list of positive integers (all in string format), write a function that returns the smallest number of spaces that can be added to the n digits of Pi such that all resulting numbers are found in the given list.

If no number of spaces to be added exists such that all resulting numbers are found in the list of integers, the function should return -1.

Choice: Choose, Explore, UnChoose

Constraints

Goal

This problem is a canonical example of a DP problem.

We try testing out combination. We start adding spaces after certain numbers.

When we add a space after a certain number, we compute the minimum number of spaces that we will need in another number, after this number.

Suppose we add a space after 31, the natural next question is, What are the minimum number of spaces needed in 41592.

“3141592”

[ 3141, 5, 31, 2, 4159, 9, 42 ]

The first space can be at any position.

Repeating subproblems

Iterate through all the prefixes. Put a space after each prefix. Check whether the prefix exists in the list.

If this prefix is found in the list, then apply the same logic again for the suffix.

Top Down: Left To Right

Can also build from right to left: Bottom Up approach.

In this problem, we have to solve all the subproblems to obtain a global solution.

In top down approach, try to return -2 on some edge cases and see if it works.

**Space**: O(n+m)

N is the length of the pi and m is the length of the given input array.

“3141592”

For 92, we only compute 92 by iterating through all of the prefixes of 92 only once.

We only compute 41592 by iterating through all of the prefixes of 41592 only once.

**Time**: O(n^3 + m): n iterations and for each iteration there is a recursive which also has n iterations.

And, there is also an operation of calculating substrings which takes linear time making the total time to cubic.

And we also check whether a substring is present in a list or not, which further takes O(m) time.

Strings are immutable and slicing through strings takes linear time.

Can optimise on this slicing.

Can pre-compute in a hash table all the prefixes once.

Using Map<Integer, Integer> over

Map<String, Integer> can save space.

Cannot understand the AlgoExpert solution currently. Try again later.

**Topological Sort**:

We are give a list of arbitrary jobs that need to be completed, these jobs are represented by distinct integers.

We are also given a list of dependencies. A dependency is represented as a pair of jobs where the first job is a pre-requisite of the second one. In other words, the second job depends on the first one; it can only be completed once the first job is completed.

Write a function that takes in a list of jobs and a list of dependencies and returns a list containing a valid order in which the jobs can be completed.

If no such order exists, the function should return an empty array.

Topological Sort has various real life applications.

The jobs could be anything, let’s say we are working at a restaurant. The jobs could be washing the dishes, cooking the food, serving food to people, paying staff, etc.

Find possible order of completion that follows the dependencies.

Topological sort is done using a Directed Graph.

Topological ordering is a linear ordering of the vertices of the graph such that for every directed edge from X 🡪 Y, X comes before Y in the final ordering. …..X…..Y…….

Represent the dependencies as a graph.

Jobs will be the nodes/vertices in our graph.

[1, 2, 3, 4]

[ [1, 2], [1, 3], [3, 2], [4, 2], [4, 3] ]

Answer: [1, 4, 3, 2] or [4, 1, 3, 2]

The direction of the edge will be from prerequisite to dependency.

[1, 2]

1 is the pre-requisite and 2 is the dependency of 1.

First finish the tasks which are not dependencies of anyone (which are not dependent on completion of any other task).

That is, vertices which do not have any edge coming into them.

1 and 4 do not have any pre-requisites.

**In a node, store its pre-requisites along with other stuff**.

If we have a cycle then its topological ordering does not exist.

Suppose we start with node 2. We see that the node 2 has pre-requisites.

So we have to finish the pre-requisites first.

2 has 1, 3 and 4 as pre-requisites. Traverse through the pre-requisites and then add these pre-requisites and then add 2.

We can start from any random node and go in any order.

Suppose we go to 3 first. 3 also has pre-requisites.

We have to go through all pre-requisites of 3 before we actually add 3 to our final ordering.

3 has pre-requisites of 1 and 4.

1 does not have a pre-requisite.

Go back to 3 and then go to 4.

Add 4 as it does not have any pre-requisite.

Go to 3 and add it. Go back to 2 and add it.

Keep track of nodes that we have already visited.

We are applying Depth First Search.

Going as deep as we can through the pre-requisites.

Once we reach the end, we backtrack and mark that node as visited.

If we have a node that is not a part of the dependency graph, then we will still have it in our topological ordering since we go through all of the vertices.

Cycles: invalid dependency chain.

Topological ordering is only valid for a graph which does not have any directed cycles.

We will deal with directed acyclic graph. (DAG).

To deal with cycle,

We will not only keep track of nodes that we have already visited but also we will keep track of nodes that we are in the process of visiting, nodes which part of our Depth First Search chain.

Now there is also an edge from 2 to 4.

Suppose we start at 2, mark 2 as in progress. Then go to 3, mark 3 as in progress. Go to 1 and 4.

1 has no pre-reqs and mark it as visited.

Go to 4 and it has 2 as pre-requisite. But 2 is in progress. If 2 is in progress, then we have got a cycle.

If we visit a node that is in the process of being visited, then we have got a cycle.

Then we return an empty list for it.

This is the depth first search algorithm solution of this problem.

O(j+d) time and space where j is the number of jobs and d is the number of dependencies.

j is the number of vertices and d is the number of edges in the graph.

O(V+E)

We will traverse once through all of the vertices and can run into visited nodes. We also go through all of the edges.

We are also building this graph.

Space complexity of the graph outweighs the complexity that we get from a recursive DFS.

We store the nodes in a map. The key is the name or number of the node.

The drawback of only using an ArrayList to store the nodes is that we cannot access the nodes according to the same name as the name of the jobs.

We are using a map for a proper connection. We also need the list.

Creating this list became easy by using a map.

In the code we have to use some order to traverse through the nodes.

**2nd solution**: 2nd algorithm

Instead of going arbitrarily through the nodes and doing DFS to find nodes that have no pre-requisites, we are going to keep track of nodes that have no pre-requisites.

Keep track of how many pre-requisites every node has.

If we have some nodes (>0) with no pre-requisites then we can apply the logic.

List of vertices which are not pre-requisites, noPrereqs.

noPrereqs = [1, 4]

Grab the last node and pop it from the list and add it to our final array.

After adding 4 in the final topological ordering, we can remove edges going out from 4.

Now 2 and 3 no longer depend on 4 since task represented 4 has been completed.

We can now update the values for the number of pre-requisites for node 2 and 3.

And so on.

If we have a cycle in the graph, noPrereqs list will become empty and final topological ordering will not have all the vertices.

O( j+d ) time and space.

We are building the same graph. We are keeping track of numbers for each vertex, but that is constant space.

For time complexity, we will be going through all of the nodes one by one and will go through all of the edges.

This algorithm has various applications.

**Boggle Board**: Teaches us a lot.

Given a two dimensional array (a matrix) of potentially unequal height and width containing letters.

This matrix represents a boggle board. We are also given a list of words.

Write a function that returns an array of all of the words contained in the boggle board.

A word is constructed in the boggle board by connecting adjacent (horizontally, vertically, or diagonally) letters, without using any single letter at a given position more than once.

While a word can of course have repeated letters, those repeated letters must come from different positions in the boggle board in order for a word to be contained in the board.

Note that 2 or more words are allowed to overlap and use the same letters in the boggle board.

words[] array can have any word.

Think of a data structure which can make this a lot simpler.

Divide our problem into 2 separate problems.

One of the problems that we have is to actually traverse the boggle board to construct words.

This is likely going to be some graph traversal problem.

The other problem we have is to match these characters to the given list of words.

We have to do it in such a way that we have to keep track of the words that we are constructing through the boggle board.

How to do this fast?

This is where we can make use of a tree data structure.

Trie

We can build a tree which will hold all of our words. When we are dealing with bunch of strings and we have to do some sort of string matching with regards to those strings, trees can come in handy.

Trees make accessing various characters from multiple strings really easy.

When using a trie, here we are not directly using string matching, instead we are making use of \* which represents end of a word in a trie.

If we dump all of the strings in a tree, at the root node it will have the letter ‘t’.

**\***: end symbol denoting end of string.

As we traverse through the boggle board, we can do string matching easily in constant time (O(1)) by using this tree.

If we start with letter ‘t’ in the top left corner, instead of iterating through all of the words and checking which words start with ‘t’, we check if the letter ‘t’ is in the root node of our tree.

We can create a separate/single root for all of our strings and connect their head with this root.

If we get a match for ‘t’, then we dive into this ‘t’ node.

We have to traverse through all of the characters in the boggle board atleast once.

Every character in the boggle board is a potential contender for the first character of a word in our list of words.

Check whether the character is present in the current level of our tree.

Treat the 2d array as a graph where each node is the element in this matrix.

Do not explore a neighbour if we just came from it. (Not allowed to repeat characters).

We have to keep track of the nodes that we have already visited during the current iteration.

Create a new 2d array of the exact same dimensions as our boggle board.

This array will keep track of the nodes that we have visited so far in one possible exploration branch of the algorithm and not so far in all of the explored branches in the algorithm.

Exploring in a depth first search approach.

As we finish exploring branches, we have to unmark a vertex as unvisited instead of visited.

We are also checking whether or not we are constructing a valid string.

Checks for preventing going outside of the matrix.

We can store the neighbours of a vertex in a list.

Keep track of the tree depth.

When we explore all of the neighbours of a vertex, we backtrack and mark it as unvisited.

By keeping track of the visited nodes, we can avoid counting the same letter more than once.

**Space**:

Building a tree which will contain all of the words in our given list of words and all of the words have atmost s characters, where s is the length of the longest word.

w: number of words that we are given.

ws auxiliary space for the tree.

We are building an auxiliary matrix which will contain booleans of whether or not the node has been visited in our current branch.

2ws + nm

n: width of input matrix, m: height of input matrix.

Returning a list of words: ws

**Recursive function calls**: never gonna have more than s recursive function calls more than once because the longest word we are given has length s.

**Time**: ws time to build the tree. Iterate through w words and they have atmost s length or s characters in them.

We are iterating through the matrix of width n and height m.

For all of the nm nodes that we explore, we go through all of their neighbours to try to construct a string.

For every neighbour we have got more and more neighbours.

1 node has atmost 8 neighbours. How far an exploration can go? s (length of longest word)

In the worst case, we explore all of the 8 neighbours. For those 8 neighbours, we have another 8 neighbours to explore.

8.8.8.8…..s times

**Time**: ws + nm + mn\*8^s (Worst Case)

Many nodes have less than 8 neighbours and the exploration won’t go s levels deep.

Thinking about the length of the words helped us to think about the overall time complexity.

Map:

We create a Trie using a map because keys serve as the indices of the children nodes and the values serve as the actual child nodes.

For a node we not saving its name in the object, but storing the name of the children as keys in the map.

In a node we are also storing the word that can be formed by reaching that particular node but we do so only for the node which has ‘\*’ as one of its children values. For other nodes, we store empty word.

We use a map because key values of a map are unique and we want the unique structure.

In a trie node, we are not storing the name of the node but storing the name of the children nodes as key values in the hash map.

In a map, we can check whether a key or a value exists in it separately but for an ArrayList, we have to check for a particular node value separately (cannot using contains() method).

If in case we did not want to store the words ‘this’ and ‘that’ under the same branch, then we can use an ArrayList (Remember B trees?)

The basic idea behind using Map and ArrayList to create a tree is because a node can have any number of children rather than having exactly 2 children as in a binary tree.

If we use an array list to create a tree, then the keys values here are the indices as a list is always ordered.

And when add a child of a node in the array list then we can do so according to the index or according to some other logic (Like defining the size of the list already and then adding a child to some particular index according to logic).

We can add a Set<String> to a List<String>

First we add all the possible Strings to a trie.

Then we are given a board. We do all possible explorations of the characters in the board and also checking whether the individual characters are present as key in the current node’s map and then add a word in our final list only when reach a node that has ‘**\***’ as its hash map key.

The exploration only becomes deeper if the current character matches the key value in the map of the current node.

**Continuous Median**:

Write a ContinuousMedianHandler class that supports:

* The continuous insertion of numbers with the insert method.
* The instant O(1) time retrieval of the median of the numbers that have been inserted thus far with the getMedian method.

The getMedian method has already been written for us. We have to write the insert method.

5, 10, 100, 200, 6, 13, 4

5, 7.5, 10, 55, 10, 11.5, 13

Uses heap data structure.

Median: number in the middle of a set of numbers (Numbers must be sorted).

1, 2, 3: 2

1, 2, 3, 4: 2.5

Every time a new element, we can sort the elements by finding the position of the new element in the currently sorted elements.

That is the previous are already sorted and then we add a new element.

This is similar to Insertion Sort.

Our insertion method will run in **O(n) time**.

Trivial answer.

There is a better way to do this using heaps.

We have to keep track of middle numbers. We don’t care about all of the numbers being in sorted order but what we do care about is keeping track of those middle numbers.

We could have a set/bucket of numbers on the left that represent the lower half of the numbers.

We could also have a set/bucket to store the greater half of the numbers.

We would like to have maximum value in the lower half and minimum value in the greater half.

If we have both of these values, then we can always compute the median.

We can either take their average or we can either take the number which comes from a greater set.

If we have odd count of numbers, then if the lower half has more numbers than greater half then maximum value in the lower half is the median and vice versa.

Heap data structure can help us to do this.

Heap data structure does not have a sorted order but it helps us to keep track of the minimum (Min Heap) or maximum (Max Heap) value of the set.

In a heap, we store the min or max values at the top.

When we insert a number, we can keep track of lower half of numbers in a max heap and greater half of numbers in the min heap.

When we get a new number to insert, we either insert it in the lower half, that is in the max heap if the new number is smaller than the maximum of the lower half or in the greater half.

Insertion in a heap is a lot faster (logn time) than insertion in a sorted array.

We may have to rebalance the heaps.

For the first element, we can add it to any heap.

After adding an element to one of the heaps, check their lengths.

If both of their lengths are equal, that is even number of elements in the set, we take the maximum value in the lower half and minimum value in the greater half and we compute their average.

But if they are of different lengths, then we take the top value of the one that has the greatest length.

5, 10, 100, 200, 6, 13, 4

First we have 5. Let’s place in the lower half.

Lower half has greater length than the greater half.

Median = 5

Next number is 10.

10 is greater than the maximum number in the lower half, so it belongs to the greater half.

Median = 7.5

5 10, 100

Median = 10

5 10, 100, 200

Median

When we add 200, 200 > 5.

Now we have got imbalanced heaps. (difference in the number of elements between the heaps >= 2)

We can no longer compute median instantly by applying the previous 2 logics.

Total count of elements is even.

After we have inserted a value in the appropriate heap, we have to check whether we need to rebalance the heaps.

We remove the top value from one of the halves (which create an imbalance) and insert it in the other half.

Rebalancing takes the same time as the insertion (O(logn)).

Insertion and Deletion in Heap: O(logn)

5, 10 100, 200

Median = 55

When we add 10 in the max heap, we will sift it up.

Now both of the half have equals lengths and compute the median.

Insertion: O(logn)

**NOTE**:

Rather than writing the code for heap construction, we can use a Priority Queue.

Another heap problem could be that **we could be given an array and we may be asked to convert it into a min and max heap**.

**Min Heap**: Lowest value has the highest priority.

To convert a given array of elements into a min or max heap, we take the first parent index and iterate down to 0. And for each of these indices, we siftDown.

We convert the subtrees into a heap.

Before deletion, the heap properties were already there.

After deletion, the reason we only have to do one siftDown is deleting a node only violates a part of the tree. Most of the subtrees are already heaps.

Rather than having 2 separate heaps for Min and Max, we create a BiFunction and define > and < operations.

**Find Loop**:

Write a function that takes in the head of a **singly** linked list that contains a loop (the list’s tail node points to some node in the list instead of None/null).

The function should return the node (the actual node, not just its value) from which the loop originates in constant space.

We can only go forwards in a singly linked list.

Traverse through the linked list, add every node to a hash table.

Since hash table can only have unique keys, If we come to a node which has already been added in the hash table, we are done.

But we can do this in constant space.

We can traverse the list using 2 pointers.

First pointer, second pointer.

First pointer traverses through each and every node.

Second pointer traverses through every second element/node. (Move the second pointer by 2 positions).

First time, the first pointer and second pointer overlap at node 7.

Relation between distances both pointers have travelled.

Suppose the first pointer has visited ‘x’ node and second pointer has travelled ‘2x’ distance and not ‘2x’ nodes.

Suppose the distance between the start of the linked list and start of the loop is ‘d’

Let the distance between the start of the loop and point of overlap be ‘p’

First pointer has travelled ‘d+p’.

Second pointer: ‘2d + 2p’

We have to find the remaining distance ‘R’, remaining distance between the point of overlap and point of origin of the loop.

We can move our two pointers by ‘R’ distance.

Our 2nd pointer has travelled the distance of entire linked list and also some additional distance.

Let’s call the distance of the entire linked list as ‘T’.

Second pointer has travelled

T + extra arc(P) = 2D + 2P

**T = 2D + P**

**D = T – (D+P)**

We can find D+P.

R = D

We move the first pointer to the head of the linked list.

The distance from the first pointer to the loop is now equal to the distance from the second pointer to the loop.

Now move F and S at the same pace. When they overlap, we will be done.

Time: O(n)

Space: O(1) We only have 2 pointers.

First pointer moves slowly as compared to second pointer. So first pointer will determine the time complexity.

**VIMP**:

Ask the interviewer about whether we are dealing with test cases that always have a loop in a linked list.

Picking the increment value of the second pointer as 2 minimizes the overall running time. Taking a value greater than 2 will lead to increase in running time.

For greatest efficiency, the slow pointer should not have travelled the loop more than once.

See the maths behind it.

<https://stackoverflow.com/questions/5130246/why-increase-pointer-by-two-while-finding-loop-in-linked-list-why-not-3-4-5#:~:text=If%20the%20linked%20list%20has,there%20will%20be%20no%20overtaking.>

**Reverse Linked List**:

Reverse a **singly** linked list in place (do not create a brand new list) and return its new head.

Assume that the input linked list will always have atleast one node.

O(n) time and O(1) space.

Put aside the edge cases. Ignore the head and the tail.

With Linked List, using multiple pointers can be extremely helpful.

We have to start from one end, preferably from the head. We cannot start from the middle as we will need reference to the previous nodes. And in a singly linked list, the ‘next’ pointers point to the next node.

Also need to store the ‘next’ reference to the 2nd pointer as we can lose it if we change ‘next’ pointer of 2nd node to point to previous node before storing it. 3 pointers.

**Merge Linked Lists**:

2 **singly** linked lists that are in sorted order, respectively.

Merge the lists in place (should not create a brand new list) and return the head of the merged list; the merged list should be in sorted order.

Assume that the lists have atleast one node.

Do not try to keep track of tail as well separately.

Keep track of the pointers.

We return the head of the linked list which has smaller first element.

Recursive and iterative solutions.

Select a base linked list, that is the list (here 1st) to which we will add in the elements of the second list at the right place.

As we iterate over both the linked lists, whenever we find ourselves at a value in the second linked list that is smaller than the value in the first linked list then we have to do some sort of mutation.

We will need 3 different pointers that point to 3 different nodes at any given time.

**NOTE**:

When we need to store previous and current pointers in a linked list, then we should start by pointing the previous pointer to null and current pointer to the head.

**Iterative solution**: O(n+m) time and O(1) space

We never stay at a certain node more than once unless we are further traversing nodes of the other linked list.

**Recursive solution**: O(n+m) time and O(n+m) space taken by call stacks.

We have 3 varying pointers at every recursive call.

**Edge Cases**:

The head of the linked list will be decided by the smallest head element in the 2 lists.

* P1 becomes null first. Append remaining 2nd list to the end of 1st list.

* P2 becomes null first. We are done.

We add the element of 2nd list into 1st list when current element of 2nd list is smaller than the current element of the 1st element.

We add P2 in between prev and P1.

Both P1 and P2 cannot be null at the same time.

(Assuming lists have atleast 1 element).

**IMP NOTE**: Ask the interviewer about what to do when P1 and P2 have equal nodes.

**Shift Linked List**:

Write a function that takes in the head of a singly linked list and an integer k, shifts the list in place (doesn’t create a brand new list) by k positions, and returns it new head.

Shifting a linked list means moving its nodes forward or backward and wrapping them around the list where appropriate.

Shifting a linked list forward by one position would make its tails become the new head of the linked list.

Whether nodes are moved forward or backward is determined by whether k is positive or negative.

Assume input linked list always has atleast 1 node.

**Approach 1**:

Find the size of the linked list.

Apply the logic for k.

Find the newTail. Using this new tail, set the newHead.

Point the ‘next’ pointer of previous tail to the previous head.

**Lowest Common Manager**:

We are given 3 inputs, all of which are instances of an OrgChart class that have a directReports property pointing to their direct reports.

The first input is the top manager in an organization chart (i.e., the only instance that isn’t anybody else’s direct report), and the other two inputs are reports in the organizational chart.

Write a function that returns the lowest common manager to the two reports.

Similar to Youngest Common Ancestor.

Main difference is in the data model that we are given.

The inputs have a different interface.

Report is basically another the object of the class.

Tree like data structure.

Draw the tree on the white board.

This problem can be solved cleanly using recursion.

To find the lowest common manager, we have to find the lowest subtree in our entire organization tree that has both of our reports in it.

Depth first search.

Iterate through all of the reports and go into the subtrees for these reports till we find both the reports.

Leaf node: the array list is empty for this node.

One of the reports can be the child of another report.

Questions to ask the interviewer:

What to do for duplicate reports ?

Time: O(n) where n is the number of the nodes, that is number of people in the chain.

Space: O(d) where d is the depth of the recursion. (Recursion).

**IMP NOTE**:

In a tree, if number of vertices is n, then number of edges is n-1.

In my first approach, I created a recursive function which return the count of number of nodes found at the subtree starting at particular ‘topManager’.

But when the count reached 2, I had to somehow return the lowest common manager.

But this function return int.

To make the function also return the lowest common manager, I passed in a map whose key is the count 2 on which we had to return. When the count reached 2, I set the value for this key(2) in the map and done.

Using a map we can return second thing from a function.

We are dealing with 2 pieces of data in the Recursion.

**Interweaving Strings**:

Write a function that takes in three strings and returns a boolean representing whether the third string can be formed by interweaving the first 2 strings.

To interweave strings means to merge them by alternating their letters without any specific pattern.

**Clarifying questions to ask**:

Should the order of characters be same for a string?

Elegant recursive solution.

We have to keep the order of the characters in the way that they appear in the string.

Merge the 2 strings so that the order of characters that was in original strings is same.

Concatenating the strings also means to interweave them.

Put the 2 strings on top of one another.

We will be declaring a few pointers to specific letters in the three strings and we will traverse all three of them at the same time.

We will start at first letters in the 2 strings.

Also look at the first character of the 3rd string.

Is one of the first letters of the 2 strings equal to the first letter of the 3rd string?

If not, we are done and the strings won’t be interwoven.

**NOTE**:

The way clement explains the question is the same way we have to explain the question on the white board in an interview.

8 cases.

When all the 3 strings are not empty, we can have 2 recursive calls and take the OR operation of these 2 results. If the first recursive call, we return true without needing to go to the next recursive call.

Cache has 1 less row and column in top down as compared to in bottom up DP.

**Interesting optimization**:

If the third string is interweaved version of first 2 strings, then the length of the 3rd string must be equal to sum of lengths of first 2 string, else return false.

If the sum of lengths is equal then we apply the further logic.

Backtracking

We are incrementing i and j just by 1, so k increments by only 1 when we do i + j.

We do further recursive calls only when there is some match.

**IMP NOTE**:

Rather than breaking the strings using substrings, use two integer pointers to iterate over the strings and pass them in the recursive function calls.

Keep a check on the pointers about whether they cross the end of the string.

Since we are only incrementing the indices by 1 and when there is a match, we do k = i+j then k only increments by 1.

Avoid doing substring() in these problems as it takes O(n) time.

When k becomes equal to the length of the third string, then we can return true because only reach in this function call when the characters match in the previous call. And also we know that the sum of the lengths of the 2 strings is equal to the 3rd string.

That is why it works. (Proof of Correctness).

With Dynamic Programming,

We need to create a cache of size

(one.length()+1) x (two.length()+1)

We can have the case when the pointer in first string has reached the end and pointer of second string, j hasn’t.

cache[i][j] has the answer for:

Can the substring of 1 starting at index i and substring of 2 starting at index j be woven into substring of 3 starting at index k(i+j) ?

In this example, the cache size is

(one.length()+1) x (two.length()+1)

Rather than (one.length()) x (two.length())

As we are returning boolean in subsequent function calls rather than some integer as we used to do in other dynamic programming questions.

Our final answer is cache[0][0]

When using bottom up, to fill in the base cases in the cache we have to compare the characters in order to fill true or false.

Here the cache can have 2 values in the base case as well.

In other dp problems covered so far, the cache used to have a single value in base case.

**Another very important point**:

Here if we had created 2d array of boolean[][], there was no way of knowing about whether the cache has been filled or not.

If we had int[][], we may have initialized it to -1.

But with boolean[][] we can only initialize it to either false or true. And they both are part of our final answer to all subproblems.

This will by default be false.

Here null comes to the rescue. null is not a part of our final answer or answer to subproblems.

So we use Object type Boolean instead of primitive type boolean.

We can even use 2d array of integers and use 0 to denote false and 1 for true.

Initialize it with -1 initially.

By checking sum of lengths, we cover 5 cases in top down and 4/5 cases in bottom up depending on initial input.

Recursion: Time: O(2^(n+m)) Space: O(n+m)

DP: Time and Space: O(mn)

In bottom up we fill the table from down to up.

**Very Very Important Concept**:

With Strings,

In Top Down approach we are starting with 0th index of both the strings which means strings starting at index 0.

In the recursion, the base case appears towards the end of the string.

In bottom up approach, we can either start from the end or from the start.

If we start from end, then if we take the last characters of both the strings, then to fill in this index, we check whether one of these characters is equal to the last character of the ….

Here a particular index means: String starting at that index.

If we start from 0, then a particular index of string means: String ending at that index.

If we go up: Final answer will be cache[0][0]

Answer to current subproblem comes from subproblems on the right and bottom.

If we go down: Final answer will be cache[str1.length()][str2.length]

Answer to current subproblem comes from subproblems on the left and top.

Both will be equal.

Repeating Subproblems.

**Shifted Binary Search**:

Write a function that takes in a sorted array of integers as well as a target integer. The caveat is that the integers in the array have been shifted by some amount; in other words, they have been moved to the left or to the right by one or more positions.

For example, [1, 2, 3, 4] might have turned into

[3, 4, 1, 2]

The function should use a variation of the Binary Search algorithm to determine if the target integer is contained in the array and should its index if it is, otherwise -1.

**Method 1**:

Use of modulus.

Clarifying questions to ask:

Duplicates?

Range of numbers?

**Method 2**:

Important observation: When we shift the array and if we stand at any index then with respect to that index either the left part or the right part of the array will be sorted.

To convert iterative solution into recursive, think about the variables which were changing in the iterative solution and pass these variables in the recursive function calls.

**Search For Range**:

Sorted array of integers. Find a range of indices in between which the target number is contained in the array and should return this range in the form of an array.

The first number in the output array should represent the first index at which the target number is located, while the second number should represent the last index at which the target number is located.

Return [-1, -1] if the integer is not contained in the array.

Duplicate targets can be present.

Use variation of Binary Search.

**Important Observation**: If the array is sorted, then if there are multiple occurrences of the target element then they have to be present together.

We will apply binary search twice.

Once to find the left extremity and once to find the right extremity.

Trying to find both by single binary search attempt may be difficult.

For left extremity,

If middle equals target, then keep looking on left.

For right extremity,

If middle equals target, then keep looking on right.

Compare target number with the number at the previous index (middle-1).

(middle should not be 0).

If they are unequal, then we know that we have reached the left extremity for the target number.

If middle is unequal to the target, then we go to left or right subarray.

Re-apply the exact same algorithm to find the right extremity.

We will check if middle index is array.length-1 and we check whether element at index ‘middle+1’ is equal to target.

If element at index ‘middle+1’ is equal to target then we go further go right else we stop.

Binary search principle of eliminating half of the array or subarray everytime.

Logic for doing so is different.

2logn

Time: O(logn),

Space: Recursive: O(logn)

Iterative: O(1)

We can write separate while loops for finding left and right extremes.

We can also use a boolean and merge both logic into one while and can call 2 functions.

(To make the code cleaner).

**QuickSort**:

Pick a pivot. Iterate through the rest of the array and we sort every other number in the array with respect to the pivot.

Every number smaller than pivot moves to the left of the pivot and every number greater than pivot moves to the right.

Pivot will reach its final sorted position.

We can pick the pivot at random or choose the first or last number as the pivot.

We have left and right pointers. Our aim to move the numbers greater than pivot to the right side of the array and numbers smaller than the pivot on the left side.

If the left pointer has a number greater than pivot and right pointer has number smaller than pivot then we swap left and right.

If left number is smaller than or equal to the pivot then it is correctly sorted with respect to the pivot.

We increment left pointer.

If right number is greater than the pivot then it is correctly sorted with respect to the pivot.

We decrement right pointer.

**Checking for equality is very very important**.

QuickSort is not stable. It exchanges non adjacent elements.

On left side of the pivot, there can be numbers less than or equal to the pivot and on the right side there can be numbers greater than or equal to the pivot.

**Another IMP NOTE**:

Swap pivot with the right pointer if we have a check for it in the third else-if, else swap with left pointer. We are guaranteed to swap the pivot with a number less than or equal to it.

When we have a single element in the subarray, then it is already sorted.

**IMP NOTE to improve space complexity**:

Call the recursion on the smaller subarray first.

The array starts to become sorted from the right hand side.

If we pick a random pivot at a random index, then after picking the random pivot, we will want to swap the pivot element with the element at the first number in the array in order to apply the logic.

**Time**:

Worst Case: Happens when the pivot divides the array into subarrays of considerable size difference. O(n^2), O(n) operation for each pivot.

Best Case: Pivot divides the array exactly in half everytime. When pivot divides the array in exactly half, then we are going to make logn calls of quicksort until we reach subarrays of length 1.

O(nlogn), n/2^k = 1, k = logn base 2

Finding the right index for a pivot takes O(n) time.

Average Case: O(nlogn)

See its mathematical proof.

Space: O(logn)

In order to implement quick sort, we need to use recursion. Have to sort the array in-place.

Frames on the Call stack. In order to limit the amount of space used on the call stack, here the technique of applying quick sort on smaller subarray first comes into play.

If we only ever apply quick sort on the smaller of the subarrays then we will make at most O(logn) calls and have O(logn) space used on the call stack at once.

Tail recursion: Removes the memory usage on the call stack at least in a lot of languages. Depends on the compiler.

O(logn) space complexity requires us to make the recursive on the smaller subarray.

If we make calls on a smaller subarray first, then we know that we will make atmost O(logn) calls and use up atmost logn space on the call stack at once.

If we do calls on the big subarray and then on the smaller subarray, then we are risking to fall into O(n) space complexity algorithm because we might make in the worst case O(n) calls on the call stack before calling the recursive function on a single element.

**QuickSelect**:

Write a function that takes in an array of distinct integers as well as an integer k and returns the kth smallest integer in that array.

Should take linear time, on average.

Searching algorithm. Uses a technique similar to quick sort.

The array may be not necessarily be in sorted order.

Another variation:

Find the kth largest element in the array

Slight variation of quick sort.

**Time**:

Best: O(n): GP

Average: O(n)

Worst: O(n^2)

**Space**: O(1) in all 3 cases.

See the proof for average case.

On average, the pivot end up in between the first and last quarter of the array.

Maximum size of the subarray everytime will be

3/4th the size of the subarray.

**Heap Sort**:

We divide the array into 2 subarrays.

One of the subarray is gonna be unsorted and other subarray will be sorted throughout the algorithm.

We will not be physically dividing this array. That is, we won’t be creating 2 different subarrays.

We will intuitively allocate space to the sorted and unsorted part of the array.

In the beginning, the sorted subarray will be empty and unsorted subarray will be the entire array.

**Goal**:

Reduce the size of unsorted subarray to 0 and increase the size of sorted subarray to the size of the array.

The unsorted subarray will not be entirely unsorted. It is going to be a Max Heap.

To sort the array in increasing order: Use Max Heap

To sort the array in decreasing order: Use Min Heap

Heaps can be represented in form of array by the way they work.

Convert the given array into max heap.

Then swap the first element with the last element.

Then do siftDown to restore the Heap properties.

Now the heap size is reduced by 1. And we have 1 element in the sorted subarray. And so on.

**Note**: Cannot convert int[] to List<Integer>

**Space**: O(1), we do everything in-place. All we do is swaps.

**Time**: O(nlogn), n is the length of the array.

**Shorten Path:**

Write a function that takes in a non-empty string representing a valid Unix-shell path and returns a shortened version of that path.

A path is a notation that represents the location of a file or directory in a file system.

A path can be an absolute path, meaning that it starts at the root directory in a file system, or a relative path, meaning that it starts at the current directory in a file system.

In a Unix-like operating system, a path is bound by the following rules:

* The root directory is represented by a /. This means that if a path starts with /, it is an absolute path; if it does not, it is a relative path.
* The symbol / otherwise represents the directory separator. This means that the path /foo/bar is the location of the directory bar inside the directory foo, which is itself located inside the root directory.
* The symbol .. represents the parent directory. This means that accessing files or directories in /foo/bar/.. is equivalent to accessing files or directories in /foo.
* The symbol . represents the current directory. This means that accessing files or directories in /foo/bar/. In equivalent to accessing files or directories in /foo/bar.
* The symbols / and . can be repeated sequentially without consequence; the symbol .. cannot, however, because repeating it sequentially means going further up in parent directories. For example, /foo/bar/baz/././. and /foo/bar/baz are equivalent paths, but foo/bar/baz/../../.. and /foo/bar/baz definitely are not. The only exception is with the root directory: /../../.. and / are equivalent, because the root directory has no parent directory, which means that repeatedly accessing parent directories does nothing.

Clarifying questions to ask: Can the given path be an empty string? No

Is the given path always valid (For an invalid path, we may have 3 dots)? Yes

Note that the shortened version of the path must be equivalent to the original path. In other words, it must point to the same file or directory as the original path.

A lot of different edge cases. Which data structure to use ?

Absolute path: starts at root directory, /

Relative path

If a path starts with ‘/’ then it is an absolute path.

If a path does not start with ‘/’, then it is a relative path.

‘/’ used anywhere else acts as a directory separator.

foo/bar/.. is equivalent to foo.

/foo/.. is same as /

We can have multiple forward slashes back to back up and that doesn’t do anything.

foo////bar is same as foo/bar

If we had single dots in between the /’s then that would have been the same thing.

/a/./././././b is same as /a/b

Double dots are meaningless in this case:

/../../../.. is same as /

We cannot go past the root directory.

Ask clarifying questions to the interviewer.

Is the path given always valid?

We want to remove all of the unnecessary information/repetition from the given path name.

We will have to interpret different symbols in our path. We will split our string into various symbols.

Split on ‘/’

Splitting “/foo/bar” on ‘/’ splits it into 3 strings, empty string, “foo” and “bar”.

Splitting functions run in O(n) time.

**IMP**:

If we are deleting an item from a list, then do not forget to add the check for non-empty list.

Empty string “” and ‘.’ are totally useless.

Interesting case: Given path starts with “..”

Filter the array.

*Keep in mind whether the path is relative or absolute path*.

When we have .. then we have to remove the previous directory (token).

We can use a stack data structure.

After filtering and applying the logic, we would

re-join the elements using ‘/’ as the joining element.

If the given path was an absolute path, then we would add an empty string to our stack which will represent the starting ‘/’

Handle the case when we have multiple double dots in the beginning of an absolute path.

**Another edge case**:

We have a relative path.

The relative path can also start with double dots.

In a relative path that starts with double dots (..), we cannot remove the double dots as we will lose information if we do so.

**Space**: O(n), n is the length of input string.

We are creating a list of tokens which will have the same length as the input string.

**Time**: O(n), Splitting the string is an O(n) time operation.

Filtering the tokens is also an O(n) time operation.

We are iterating through n tokens and the operation or set of operations that we are going to do for each token is/are going to take O(1) time.

All we are doing is checking whether the token is an empty string or a single dot or a double dot.

(O(1) time operation)

Popping and pushing on the stack: O(1) time

Final Join: O(n) time operation.

If we re-create the final shortened path by iterating through all of the tokens and appending them to a string, then that will be O(n^2) time operation in the majority of languages where strings are immutable.

When the path is an absolute path, we treat the empty string as a token and add it in the stack. Using join, we append ‘/’ in between all of the tokens and not at the edges.

Difference between .equals and == with Strings.

Generally .equals is used for Object comparison, where you want to verify if two Objects have an identical value.

== for reference comparison (are the two Objects the same Object on the heap) & to check if the Object is null. It is also used to compare the values of primitive types.

Better to keep the logic for absolute and relative paths separate to improve code readability.

**Longest Substring Without Duplication:**

Write a function that takes in a string and returns its longest substring without duplication.

We can assume that there will only be 1 longest substring without duplication.

We are going to traverse the string and will keep track of last seen index of each letter.

Store the corresponding of the letter in a hash table.

Also, mark the starting index which means that starting at this index, we have no duplicated characters.

c l e m e n t i s a c a p

When we reach ‘l’ we check whether l is present in our hash table or not.

c : 0

l : 1

e : 2

m : 3

Update the longest substring to “cl”

When we reach second ‘e’, we see that ‘e’ is present in the hash table.

Now we update our start index.

We want to check which of the 2 substrings is bigger:

Current substring that ends at the current index (this index is not included in the substring ‘clem’) vs longest substring so far.

Find the start index of our new substring.

Check which is bigger: ‘clem’ or ‘me’

We update ‘e’ in our lastSeen.

We see ‘e’ as the first duplicate character. We have to start from index 3 as before this we have a duplicate character ‘e’.

If the 2 substrings are equal, then talk to the interviewer about which one to take. Already existing one or one that comes later.

c l e m e n t i s a c a p

when we reach ‘c’ at index 10, then we check whether we can start right after the previous duplication of ‘c’, that is at letter ‘l’, but we already have a greater start index of 3 as there was a duplicate ‘e’.

max(3, 0+1) = 3

**Time**: O(n),

**Space**: O(min(n, A)), where n is the length of our string and A represents the length of the alphabet that is represented in our string. We are storing letters in a hash table and we are storing a substring of a certain length.

How many letters at most can we store in the hash table ?

Number of letters in the string: No, this is incorrect as it will only be the case when all the letters in the given string will be unique.

Set of unique letters that are present in our given string.

In an iteration, startIdx represents the current substring.

The duplicate character can be there in the current substring or it can also be there at an index before the ‘startIdx’.

We have to make sure that value of ‘startIdx’ is correct all the time.

**Underscorify Substring**:

Write a function that takes in 2 strings: a main string and a potential substring of the main string.

The function should return a version of the main string with every instance of the substring in it wrapped between underscores.

If 2 or more instances of the substring in the main string overlap each other or sit by side by side, the underscores relevant to these substrings should only appear on the far left of the leftmost substring and on the far right of the rightmost substring.

If the main string does not contain the other string at all, the function should return the main string intact.

Time complexity analysis is bit difficult as compared to other questions.

Suppose we were underlining substrings in a word document or a Google doc or we were italicising substrings, we would underline or italicise the entire group, not some separate words.

**Think**

**Divide the problem into different stages.**

Steps to approach the solution:

Find every location of the substring in the main string.

We will find the indices that represent the location of the given substring.

Pair: start and end index.

First get the locations.

If we have the locations, then we cannot add underscores straight away.

Before we add underscores, we have to get final locations.

The next step will be to collapse locations.

We will have a bunch of arrays with indices where underscores need to be and we will collapse these arrays into just 1 array that represent the entire group where substring overlaps or sits side by side.

And finally we will have our “underScorify()” function which is going to take our final locations and actually add the underscores.

The best way to get the location of substring “test” is to traverse the main string and call a built in function to find all the locations of “test”.

We first call the built in function at index 0. Next, we call it on the next index, 1.

Can’t we call it after the length of the substring? That is from index 4 ?

No, our substring might overlap itself.

“tttttttt”, “ttt”

Here the substring “ttt” overlaps the main string several times.

The first substring match when we call the find() function was at index 10. So we know that in between there is no match. The only potential match that we can get now is after index 10, index 11.

When we call find() on index 11, we get a match at index 14. And so on.

Now we have the locations and now we would like to collapse them.

We will iterate through locations array. We will keep track of previous location and build a new locations array.

In the locations array,

14 is repeated

We reach 10 and check whether this index 10 overlap or sit next to the previous index that we had.

Previous instance of “test” substring ended at index 4.

Next we go to 14, 18.

Current substring index matches with the end index of previous substring match.

We collapse them. We take the rightmost index of current array and replace the rightmost index of previous array with the rightmost index of current array and add update it to the newer locations array.

In the end, we iterate through our main string and new locations and create a new char array.

And then join all the characters to form the final String.

Use of built in String function, indexOf()

**Time**: lot more complicated,

Look on all 3 functions and see which one is complicated. O(mn)

3rd underscorify() function: Runs in O(n) time.

In collapseLocations() function: O(n), will be biggest when we have underscores after every character.

In getLocations() function, we traverse the string and call the built in function to find indices.

Traversing the string: O(n)

The fastest runtime that matching function can have is O(n+m) where n is the length of the main string and m is the length of the smaller substring that we are given. We call this .find() at almost every point when we traverse the main string, then naturally the runtime should be O(n^2 + nm).

But this is not correct.

From a simple algorithmic analysis point of view, this makes sense. It is not as specific as it could be. We can apply some Amortized analysis techniques to narrow this down.

We look at the different types of input that we can get. The different ramifications that these different inputs can have on the subfunctions like traverse() and .find().

And then from there we can narrow down the actual time complexity.

Lets imagine that we have a string “abcabcabctest” and smaller substring is “test”.

We first call .find() function at index 0.

And then we call .find() function at the ‘e’ of test. And we don’t find anything and we are done.

O(n+m)

Lets say we have ‘aaaaatestaaaaatest’ and ‘test’

This function will run in roughly O(2(n+m))

Suppose we have ‘test’ a bunch of times.

‘testestestestestest’, ‘test’

We have the substring ‘test’ roughly n/m times. Here it does not run in O(n(n+m)) times.

We are calling .find() method roughly n/m times.

The runtime of .find() method here is not O(n+m).

In the case where substring is located back to back to itself, we find it really quickly.

All we look at it ‘esttest’, that is roughly 2m characters.

When we call it the second time, we iterate through the characters esttest.

The runtime actually ends up being n/m \* 2m

= O(n)

The worst case is when we have a very few instances of the second substring which led to the runtime of O(n+m).

The time complexity varies depending on the input.

How different inputs can affect different parts of an algorithm.

Different sub algorithms.

**Space**: O(n), n is the length of the main input string. There are 2 things that we are storing.

Our new string will be atmost of length ‘2n’ if we had underscore after every character.

The getLocations() and collapseLocations() functions where we store new arrays, the biggest these can be is when we have underscores almost every other character.

These will have an upper bound of O(n).

We cannot have more locations than the characters in the main input string.

**Pattern Matcher**:

We are given 2 non empty strings. The first one is a pattern consisting of only “x”s and/or “y”s; the other one is a normal string of alphanumeric characters. Write a function that checks whether the normal string matches the pattern.

A string S0 is said to match a pattern if replacing all “x”s in the pattern with some substring S1 of S0 and replacing all “y”s in the pattern with some substring S2 of S0 yields the same string S0.

If the input string does not match the input pattern, the function should return an empty array; otherwise, it should return an array holding the strings S0 and S1 that represent “x” and “y” in the normal string, in that order. If the pattern does not contain any “x”s or “y”s , the respective letter should be represented by an empty string in the final array that we return.

We can assume that there will never be more than 1 pair of strings S1 and S2 that appropriately represent “x” and “y” in the normal string.

“xxyxxy” (pattern)

“gogopowerrangergogopowerranger” (main string)

How do we actually check whether or not the main string solves the pattern and how do we find the actual values that may get matched?

First we can do is, find how many instances of the letter ‘x’ and how many instances of the letter ‘y’ appear in our pattern?

Gather the counts of ‘x’ and ‘y’ in our pattern.

What we will do is we will generate a new pattern which will simplify our algorithm a lot.

Generating a new pattern means that we are going to make sure that our pattern starts with an ‘x’ instead of ‘y’.

We could have been given the pattern ‘yyxyyx’ which would have been same, just values will be interchanged.

Turn the pattern that starts with ‘y’ into the one that starts with ‘x’ and track whether we actually did that switch.

We will create a function, getNewPattern().

It will return an array of our new pattern.

It will check whether the first letter is ‘x’ or not. If it is, then just return an array of all of the letters in our current pattern.

If the first letter is ‘y’, then return an array where we turn every instance of the letter ‘y’ into letter ‘x’ and every instance of the letter ‘x’ into ‘y’.

And keep track whether or not we actually had to do that switch.

Just an array version of the given string with the switch applied if required.

Now we get the counts of ‘x’ and ‘y’. Get the first position of ‘y’.

We know in our new pattern, the first letter is ‘x’, we know that all the time we start with an ‘x’, we want to say, ‘where does the first y appear’?

We create a new function, getCountsAndFirstYPos().

It counts the number of ‘x’s and ‘y’s and gives us the first ‘y’ position.

Check whether if there is a ‘y’ by checking the count of ‘y’s.

If count of ‘y’s is 0, then we have to deal with another way of solving the problem as we will only be dealing with pattern of ‘x’s.

It will be an edge case.

If the count of ‘y’s is >0, then we will iterate through our main string and just try a bunch of combinations.

Try a substring by having it as our ‘x’ variable.

If this is our ‘x’ variable, could we even have a ‘y’ variable based on the number of ‘x’s that we have and the number of ‘y’s that we have.

Does the string actually divide itself properly?

We start by saying, let’s try a substring X of length 1.

We iterate through our main string and we start at length of 1 with the first letter ‘g’.

Let ‘x’ be represented by ‘g’.

Let the ‘lenOfX’ be equal to 1. If the len of ‘x’ is equal to 1, then we can actually get what the length of ‘y’ should be.

If the length of ‘x’ is 1 and we know that we have 4 x’s and 2 y’s.

And we know that we can calculate the length of our main string, we can just call the .length() function on it. Then we can find the length of corresponding ‘y’.

The length of our main string is 30, then length of ‘y’ will be = ( length of the string – lenOfX \* number of x’s that we have in our pattern ) / number of y’s that we have in our pattern

If lenOfX =1, then lenOfY = (30-1\*4)/2 = 13

Then we check whether it is possible or not.

Yes, we can have a substring of length 1 and substring of length 13.

We may not get a round number as the length of potential ‘y’ depending on the length of the string.

+ve integer: valid

Where would ‘y’ start in our main string?

What would be the first ‘y’ index of ‘y’ in our main string ?

We found the first position of ‘y’ in the pattern and it was at index 2.

There were 2 ‘x’s before the first ‘y’.

We can calculate the first ‘y’ index in the main string to be equal to,

yIdx = firstYPos \* lenOfX

The substring that represents ‘y’ starts at the index 2 in the main string.

What are the actual representations of ‘x’ and ‘y’?

x = “g”

y = “gopowerranger”

All that we have done so far is we have gone through a bunch of calculations to lead to the point where we have a potential ‘x’ as ‘g’ and a potential ‘y’ as “gopowerranger”.

And now we want to check, do these potential ‘x’s and ‘y’s actually match the main string.

The pattern that we get with getNewPattern() function, we replace every ‘x’ with “g” and every ‘y’ with “gopowerranger”.

The string that we will get is, “gggopowerrangergggopowerranger”.

Compare this string with our main string.

Now, we move on and iterate through our entire substring this way.

Now we update lenOfX as 2. Calculate new value of lenOfY and so on.

And we stop when we get a match as we have assumed that there is just 1 pair.

If in the beginning, we had replaced ‘x’s with ‘y’s then in the end we will swap x and y.

**Time**: O(n^2 + m)

The getNewPattern() function runs in O(m) time. Because we are generating a new pattern.

And getCountsAndFirstYPos() function also runs in O(m) time.

All we are doing is traversing this pattern and counting the number of x’s and number of y’s and keeping track of the ‘y’ position.

The first 2 functions run in O(m) time.

The main function is going to run in O(n^2) time.

Not only we are iterating through the main string, which has length n entirely, that is an O(n) operation.

At each point that we are iterating through this, we are generating strings X and Y, and then we are transforming our getNewPattern() array into a string where we replace all the x’s with the tentative x and all the y’s with the tentative y.

And then we compare this string of length n to the main string of length n.

All these things, mapping the new pattern to a new string, using these substrings, and then comparing this new built string to our main string. All of these are going to be O(n) time operations.

Calculations of the length, etc are O(1) time.

We are doing all of the operations a total of n number of times as we iterate through the main string, the time complexity becomes O(n^2).

Can discuss with the interviewer whether to neglect m in comparison to n^2.

The time complexity does depend on the input pattern, which has length m.

**Space**: O(n+m) where n is the length of our main string and m is the length of the pattern.

We generate the pattern in the beginning and this has a length of ‘m’.

And then we generate ‘counts’ hash table which does not take much space. It is constant space.

In our main function, everytime we generate a potential ‘x’ and potential ‘y’, we end up generating a potential string when we replace all the x’s and all the y’s with the potential ‘x’ and potential ‘y’.

During each main iteration, we are generating a string of length n, where ‘n’ is the length of our input string.

This means that our algorithm at any given time is using up O(n+m) additional space.

**Multi String Search**:

Write a function that takes in a big string and an array of small strings, all of which are smaller in length than the big string. The function should return an array of booleans, where each boolean represents whether the small string at that index in the array of small strings is contained in the big string.

Note: Cannot use language built in string matching methods.

**Naïve solution**:

Iterate through every string in our list of small strings. And for each of them we will iterate through the big string, compare every character in the big string to the first character in our small string.

When we find two characters that are equal to each other, then we start comparing every character thereafter to see if our small string is there in the big string.

We are iterating through all the small strings. For all the small strings, we are saying, iterate through all the characters in the big string.

Anytime we see a character that is equal to the first character of our current small string, start moving forward and seeing if all the characters after that match our small string.

If they do, then we found a small string in our big one. Otherwise, we keep redoing that process.

**Space**: O(n) where n is gonna be the number of small strings, the length of the list of small strings.

It is purely determined by the output.

We have to output an array of length n.

Otherwise, we never really hold any unnecessary or auxiliary space as we compare our characters.

**Time**: O(bns)

b: length of the big string.

s: length of the biggest small string.

We are iterating through every small string.

n iterations.

For every small string, we iterate through entire big string.

For every character in the big string, we compare atmost ‘s’ characters.

For each iteration of character in the big string, we are doing atmost s (length of largest small string) other iterations to match characters.

This is where the worst case scenario time complexity of bns comes from.

Can we do better?

The space complexity is the best that it is gonna be.

No matter what our algorithm is, we are always going to return an array of length n.

If anything, the space complexity can only become worse at this point.

What is the data structure that can help us? Trie

Tries are really helpful when it comes to string matching or verifying whether or not certain strings are contained in others.

We are going to build a suffix trie of the big string.

We are going to have a root node that will point to the first letter of every suffix in the big string.

And it will store every suffix that starts at that letter.

So what are all the suffixes in our string.

We are going to have every suffix.

The suffix tree will hold every suffix in the big string.

Building the suffix trie will take O(b^2) time and space.

We are iterating through all of the suffixes: 2 for loops.

Now we can iterate on small strings. And for every small string we can say, Is this string contained in our suffix trie?

To verify whether or not a suffix is contained in a suffix trie or not just takes O(n) time where n is the length of the string.

When we iterating through our small strings to see if they are contained in our suffix trie, we are not actually saying, Is the string “**this**” a suffix ?

If we were looking for “this” to be a suffix, then we would not have found it in the trie.

We are not looking for whether or not “this” is a suffix.

We are trying to check that “this” is simply contained in the big string.

And by having built a suffix trie, we can very easily check if “this” is contained in it.

If we can traverse the trie to the final letter of the current string that we are at. In this case, “this” and the final letter of this is ‘s’.

If we can get to a point where we can find ‘s’, by going through ‘t’, ‘h’ and ‘i’ in the suffix trie, then we have found the word, ‘this’, it is contained in the big string.

This is going to take O(n) time, where n is the length of the current string we are looking for which is, “this”.

What is the biggest string in our small strings, it is gonna be “bigger” or “string”.

Both of these have a length of ‘s’ (length of largest small string).

So for all of our small strings, when we check to see if they are contained in our suffix trie, we are going to take atmost O(s) time.

Because again, to find if the string is contained in this suffix trie, it is just gonna take O(n) time where n is the length of that string.

If the largest length of our small strings is ‘s’, then we are going to take O(s) time.

Finding a word like “string” in our suffix trie.

At the root node of our suffix tree, we would have the letter ‘s’.

This letter ‘s’ will be located at the root node. This ‘s’ is going to point to ‘t’ which will point to ‘r’, then ‘i’, ‘n’, ‘g’ and then to asterix ‘\*’.

A letter can point to multiple nodes. The letter ‘s’ will also point to a space.

In this suffix trie, we don’t really care about the fact that there is an asterix, ‘\*’ at the end of the suffix because we are just going to traverse it until we find that a small string is contained in it, not necessarily that it ends in an asterix, which normally denotes the end of an actual string.

We are iterating through all of the small strings and for each of them we are saying, check if it is contained in the suffix trie.

Check if the word is contained in the suffix trie. It takes O(n) time, where n is the length of the string.

The largest string in the small strings has the length ‘s’.

There are n small strings. We will be doing O(ns) operations, that is O(ns) time to find if all of the n strings are contained in the suffix trie.

We have a total time complexity of O(b^2 + ns).

Space complexity will be O(b^2+ n).

Is it better than O(bns) ?

It depends on the length of the big string versus the length of the small strings.

Can we even do better than this? Yes

It also uses trie.

We will build trie based on small strings.

What if we didn’t build a suffix trie but just build a normal trie containing all of our small strings.

We build this trie.

And then we iterate through the big string, through every character in the big string.

And we check, is there a small string contained in our trie starting at this character in our big string.

We start at the first character in our big string, which is ‘t’ and we say, Is ‘t’ located in our trie ?

Here we check whether or not if there is an end symbol or not after we find a match for a small string.

We are looking for an asterix as it denotes the end of the small string.

“big” does not point to an asterix.

To build the trie is going to take O(ns) time, where n is the number of small strings.

We are iterating through all the small strings and inserting them into our trie.

We are doing atleast n operations and to insert a word or a string in a trie is going to take O(n) time, where n is the length of that string.

And because we are dealing with small strings that have atmost length ‘s’ (largest length of the small strings), then we are going to have a time complexity of O(ns) to dump all of these in the trie.

The space complexity is going to be O(ns) because we are building a trie that has ‘ns’ characters.

The time it is going to take to iterate through the big string and find whether or not we have all of the small strings, find whether or not they are contained in the trie is going to be O(bs) time.

Why is it O(bs) time?

We are iterating through the entire big string.

We are iterating through all of the characters and then we are going to check in the trie whether or not we have a string.

We are doing ‘b’ iterations through all the characters.

Then why ‘s’?

As we iterate through all of the characters in the big string, for each of them, we say, check if that character is in the trie that we have built.

If it is, check if the next character is in the trie that we have built.

Then check if the next character is in the trie that we have built.

And if we find an asterix, we found a string.

If we ever get to a point where the current character is not in the trie that we have built, for instance, when we are at big, b, i, g, if we get here and if the next character which is a space is not in the trie, then we just stop.

We can imagine that we will never iterate through more than ‘s’ characters.

Because the largest amount of characters that we can find in our trie is going to be the length of the largest of the small strings.

So the largest amount of characters that we can iterate through until we reach the end of our trie is going to be the length of the largest of our small strings.

For instance, “bigger”.

In the large string, at each character that we iterate through it, we will do atmost ‘s’ traversal deeper and deeper into our trie because our trie is never going to reach past ‘s’ characters down.

**Time**: O(ns+bs)

**Space**: O(ns+n) = O(ns)

The naïve solution had the best space complexity.

We tend to prefer optimizing time complexity.

Iterating through all the small strings and checking whether or not they are contained in a suffix trie is exactly the same thing as iterating through all of the same strings and inserting them in a trie.

We know that the small string must be able to be contained in the big string since small strings are smaller than big strings.

Since ‘s’ is always going to be smaller than b, then bs will be better than b^2.

This problem is solvable in many different ways.