**Apartment Hunting**:

You are looking to move into a new apartment, and you are given a list of blocks where each block contains an apartment that you could move into. In order to pick your apartment, you want to optimize its location. You also have a list of requirements: a list of buildings that are important to you.

For instance, you might value having a school and a gym near your apartment.

The list of blocks that you have contains information at every block about all of the buildings that are present and absent at the block in question. For instance, for every block, you might know whether a school, a pool, an office, and a gym are present.

In order to optimize your life, you want to minimize the farthest distance you would have to walk from your apartment to reach any of your required buildings.

Write a function that takes in a list of blocks and a list of your required buildings and that returns the location(the index) of the block that is most optimal for you.

If there are multiple most optimal blocks, your function can return index of any one of them.

It has got a canonical less optimal and more optimal solution.

We are given a lot of information in the prompt, some of which is actually not at all useful for the problem.

This is often done in interviews because the interviewee is tested on their ability to be able to parse the prompt and identify what information is actually going to be useful and what information is not going to be useful.

List of hash tables. List represents blocks in a city.

Each index represents different block of the city and every hash table in that list represents the buildings that are either present or absent at a given block.

All of those buildings that map to false, we can almost ignore them entirely.

Answer: Index 3

At index 2, gym and school are super close. But we will have to walk 2 blocks to get to the store.

We have to find a block that has an apartment that is going to have to walk the least amount of distance to reach all of the important buildings.

Gym, school and store matter to us.

Maximum distance to reach all of the important buildings has to be minimized.

The first way to approach this is to iterate through all the blocks and calculate distances.

At every block, we go through all of our required buildings.

For gym, we will see how much minimum distance it takes to get to the gym from all indices in the list.

How far is the closest gym from index 0 ?

How far is the closest school from index 0, our current index ?

It’s right there, 0 away.

For block 0, we iterate through all of our 3 requirements.

We have all the distances of the closest of each building that is important to us.

How far away is the farthest one of these buildings?

For index 0,

G, Sc, St

1 0 4

We take the maximum of these 3.

Our final decision for choosing a particular block as our apartment will depend on the maximum distance of a particular required place.

We have to take the minimum of these maximum values for each index.

For index 1,

G, Sc, St

0 1 3

This means that, if we take our apartment at index 1, we have to walk atleast 3 blocks to reach all of the 3 required buildings.

We can see here that index 1 is better than index 0.

What we are doing is, we are iterating through the entire array of blocks, then at each block we are iterating through the entire array of required buildings and **for each of those**, we are re-iterating the entire array of blocks to find the closest distances.

3 nested for loops.

Time Complexity: O(B^2\*R)

B represents the number of blocks that we have in our list and R represents the number of requirements that we have in our list.

Constant time operations (Comparisons) for each iteration.

Space complexity: O(B)

Creating an extra array or list to keep track of maximum values.

**Steps to follow for writing code**:

* We need the first loop over the blocks in order to use that index for calculating the minimum distance of each required building from a particular block.
* Once we have minimum distance of all required buildings from all blocks, we take the maximum of these minimum distances for each individual blocks and store in another array.
* Then we return the index of the element which has the minimum value in this new max array.

We have store the maximum of minimums.

And then take the index of the minimum element.

Clarifying questions to ask:

What if one of the required buildings does not exist in any of the blocks ?

We can avoid creating a list. Can keep track of a variable that says ‘smallest distance’ and keep track of index at which it will be overwritten.

Important optimal space technique to find the maximum of minimum values or minimum of maximum values.

We do not need to create minDistances[] array.

**Can we do better from a time point of view? Yes**

It seems like we will have to iterate through all of the blocks and requirements once.

Thing that bothers is B^2 term and we can eliminate the B^2 term.

In the 3rd for loop, where we re-iterate through all of the blocks, so that for each requirement we can find the nearest building of that requirement and find that closest distance.

Is there some way in which we can pre compute that such that our iterations are then fewer? Yes

Precomputing technique.

This technique could be applicable in a lot of algorithm problems.

Let’s say we have a list and we have a bunch of for loops to compute values, ask yourself, maybe there is a way to precompute values and then optimize like that.

For all of our requirements, imagine that we already knew all of the closest distances of the buildings from each index.

Imagine for the gym, we knew that at index 0, the closest gym was one block away. At index 1, the closest gym was 0 blocks away and so on.

G: [ 1, 0, 0, 1, 2 ]

Similarly imagine that we knew this for schools and store as well.

School: [ 0, 1, 0, 0, 0 ]

Store: [ 4, 3, 2, 1, 0 ]

We were calculating these values in the 3rd for loop in previous solution.

If we have these values, then we could literally just iterate through all of the blocks and for index 0, take the maximum of the 3 values at all the respective 0 indices and so on.

Store these maximum values and find the index of the minimum value.

Imagine we are given these 3 arrays, the solution will then just be O(BR) time complexity.

We just do one iteration at each of the blocks and for each of the blocks, we calculate the maximum values for each of the r distances (r rows, number of requirements).

We can compute these values in a time efficient manner in O(BR) or less.

If we can compute the distances row for any of the requirement (required building) in O(B), then we can do all of these computations ‘r’ times.

For every requirement, we can do 1 pass from left to right.

Iterating for gym:

For each block, Keep track of the **closest/nearest** gym that came before me.

At index 0, there is just school. No gym is before it.

At index 1, there is 1 gym and it is 0 blocks away.

For gym,

[ X, 0, 0, 1, 2 ]

Now, we do another iteration from right to left.

For index 4, we check whether there is a gym, no there is not and the closest gym is 2 blocks away.

For index 1, there is a gym to the right of it which is 1 block away.

But at index 1 there is already a value less than 1, we have to take the minimum value.

At index 0, we update the minimum distance from the gym to be equal to 1.

We do this in O(B) time. We do ‘r’ computations in O(B) time. So, that is O(BR) time.

Now the total time complexity is O(BR) time.

Space complexity will be a bit worse.

**Space**: O(B+BR) = O(BR)

We are storing ‘r’ arrays and each array has ‘B’ length.

We got rid of the squared term.

This improved the time complexity but made the space complexity worse.

**IMPORTANT POINT TO NOTE**:

The buildings given as part of the required buildings array have to be present at one of the blocks.

So we don’t need to check whether a building won’t be present in any of the blocks or not.

**Note**:

In a 2D array, we can iterate row by row or also column by column.

Pre computing techinque: left to right and right to left iteration of the array.

**Calendar Matching**:

Imagine that you want to schedule a meeting of a certain duration with a co-worker. You have access to your calendar and your co-worker’s calendar (both of which contain your respective meetings for the day, in form of

[startTime, endTime], as well as both of your daily bounds (i.e., the earliest and latest times at which you are available for meetings every day, in the form of [earliestTime, latestTime].

Write a function that takes in your calendar, your daily bounds, your co-worker’s calendar, you co-worker’s daily bounds, and the duration of the meeting that you want to schedule, and returns a list of all the time blocks (in the form of [startTime, endTime]) during which you could schedule the meeting, ordered from earliest time block to latest.

Note that times will be given and should be returned in military time. For example: 8:30, 9:01, 23:56.

It has got a real world application.

Coding part is difficult in comparison to algorithm part.

Algorithm part is logical.

We are given the duration of the meeting and 2 calendars, our and our co-worker’s.

We are also given our and co-worker’s daily bounds, that is in-time and out-time.

What does it mean for a time range to be valid?

We both don’t have to be busy. There should not be any other meeting in that range and the range should fall in the in-time, out-time range.

Find the availabilities in between the given blocks of time.

Common availabilities.

We are given the calendar of meetings, that is the times we are unavailable.

First we could convert the given 4 inputs, the 2 calendars and 2 daily bounds into one calendar of all of the **unavailabilities** where atleast one of the 2 co-workers in unavailable.

First thing we should do is update both the calendars to take into account the daily bounds.

We can create 2 additional meetings for each person from 2 bounds.

0:00-9:00 (will go to the far left of the 1st calendar),

20:00-23:59 (will go to the far right of the 1st calendar)

Updated calendars that tell about when both the individuals are busy.

Now, we merge these calendars.

We longer want to deal with 2 separate calendars.

Lets just deal with 1 merged calendar that basically tells us what the unavailabilities of these 2 individuals are.

If we are trying to find a meeting that matches for both individuals, then we need the availabilities or the blocks of time where not just 1, but both the individuals are available.

Opposite of this is a calendar with all the unavailabilities where atleast one person is unavailable. It’s basically merging the 2 calendars.

The way we are going to merge these 2 calendars is we are going to merge them in a sort of merge-sort fashion.

We will have pointers at the beginning of both of the calendars.

When the meeting time in calendar 1 and calendar 2 have the same starting time, then pick either of them first and add it to the merged calendar.

We want the calendar to be ordered from beginning of the day all the way to the end of the day because that is going to make things a lot easier for us when we finally have to find the availabilities.

The question gives us the calendars in a sorted order. Good question to ask to the interviewer.

If we want to merge both of them in a sorted order, then we can do so in a merge-sort fashion where we iterate through both calendars looking at the first value in the calendar at any given time and seeing which one is smaller than the other.

We will compare the starting time of a given meeting.

We will put pointers at the beginning of both calendars.

Pick 2 times and compare which range has smaller starting time.

If they both start at the time, we can pick either.

Move the pointers.

We merged the 2 calendars in ascending order based on the start time of every meeting.

These represent all of the times where atleast one of the co-worker is unavailable or busy.

Now we can look in-between blocks of time. These in-between times will be where both the co-workers will be available.

If one block of time is bigger or atleast equal to the meeting duration then it would be one of the valid blocks of time.

Our merged calendar is crazy looking in terms of end time.

We can flatten or merge some of the times into one.

Flattening is a common techinque in algorithms.

Flattening the ranges.

Flattening the array or ranges into fewer ranges that can capture multiple ranges at once.

Flatten these so that they no longer overlap.

Calendar of unavailabilities: Where atleast one person in unavailable.

Check the gaps.

**Time**: O(c1+c2), merge step

Initial merge: O(c1+c2), iterating through the main merged calendar will also be the same thing. Merged calendar’s length will be atmost c1+c2.

If the merged calendar never had overlapping time slots, then will iterate through c1+c2 ranges.

**Space**: O(c1+c2), will be creating a merged, flattended and updated calendar.

Will convert String to number.

Total **5 steps**.

**Iterative In-Order Traversal**:

Write a function that takes in a Binary Tree (where nodes have additional pointer to their parent node) and traverses it iteratively using the in-order tree-traversal technique;

The traversal should specifically not use recursion. As the tree is being traversed, a callback function passed in as an argument to the main function should be called on each node (i.e., callback(currentNode) )

Each BinaryTree node has an integer value, a parent node, a left child node, and a right child node. Children nodes can either be BinaryTree nodes themselves or None/null.

Recursive calls take space on the call stack.

Iteratively, constant space is used.

This question can be asked in the following way:

Traverse a binary tree in order using constant space.

The binary tree example we will see covers all of the edge cases.

If we are at a node, say 2, we can either call the callback that we were given on it or we do not call it and decide to call it at a later time.

What is gonna make us decide whether or not to call it is basically, what the previous node we traversed was.

If we are coming from the node with value 1 to the node with value 2, then we know that we should not call the callback on value 2.

In in-order traversal, we first call the callback, or we first explore all the nodes to the left, then the top level node, then all the nodes to the right.

We explore the left subtree first.

Imagine that we have explored all of the left subtree and were coming up, from let’s say the 4 node, then we would like to call the callback on 2 because we have explored all of the left subtree of 2 and we have presumably called the callback in all the right places there.

And now its time to call the callback on 2.

In the given example, we first call the callback on 4, 9, 2, 1, 6, 3, 7.

The key thing to decide is to whether to call the callback on a given node, which depends on what the previous node we just traversed was.

We are going to keep track at every node what the current node is, what the previous node was and what the next node is gonna be.

We are going to do all of this iteratively to avoid using any extra space, just a constant amount of space.

When we first traverse the tree, the current node is going to be 1.

The previous node is going to be null because there is nothing above our current node.

The next node is going to be 2.

We will go left first because this is in-order traversal.

How do we determine that 2 is the next node ?

If previous node is null, then it means that we are at the root node.

If there is a left, if we do have a left node, we must go left.

In this case, we do have a left node and so we are going to do just that.

We are going to grab our current node pointer and put it next to the 2.

And then we are going to grab our previous pointer and point it to 1.

We are at 2 now. How do we know that we should not call the callback on 2 and instead, we should further explore down to the left.

Here every binary tree node has a pointer to its parent node.

If the previous node is equal to our current node’s parent node, then that means that we just came from the top and we have to further explore down.

So we go further down.

Because we have a left child here further down, we can just move our current and previous nodes.

Now we are at node ‘4’

At 4, there is no left child node.

Now it is the time to explore to the right of the node 4.

Before we go to explore on right, we have to call the callback on 4 and that is when we call it.

We move our current node to the 9 and previous node to 4.

Since the previous node is the parent node of 9, we further explore down for node 9.

We see that we have nothing on the left. We call callback on 9 and try to go on right.

We don’t have right node either, so we go back to our parent node.

Now, the current node becomes 4 and previous node becomes 9.

Now we are at 4. And now, the parent node of 4 is not equal to the previous node.

Here, the previous node is a right child, 9, we know that we must have explored all of the right subtree, which means that we must have also already called the callback on ourselves because we always call the callback on ourselves before we go down to the right.

So, the next thing that we have to explore is our parent node, we have to go back up.

We move our current node to 2 and previous node to 4.

Now the previous node is the left child of 2, which means that we are coming back up from the left.

So we are gonna call the callback on ourselves.

Once we call the callback of 2, we set our new current node, or next node to be either our right child, or our parent node.

When we call the callback on ourselves, we always want to go right.

If we can’t go right because we don’t have a right child node, then we go further up.

Now, our previous node is our left child node, so we call the callback on 1.

Move our current node either to the right or to the top if we don’t have a right node.

Here, we do have a right node.

Our current node is 3 and previous node is 1.

Now we again have the case where the previous node is equal to the parent of the current node.

When previous node is equal to the right child, we go back up.

When we go back up and reach the point where, current node is 1, and previous node is 3.

Right child of current node is the previous node.

We are going to make our current node be null and move the previous node to our current node.

Now we are done.

**Time**: O(n)

We explore single node in the tree and that it pretty much it, and at every node, all we are doing is checking, what was the previous node ?, what was our current node’s parent ?

We are doing these constant time operations.

n is the number of nodes in the tree.

We will probably visit a node, at most three(3) times.

A node like the number 3, we visited 3 times.

We visited it once when we went down, then we went to 6 then back up to the 3 and we called the callback, then down to 7, and then, back up to 3.

Since we will visit a node, atmost, roughly 3 times, and some of them, fewer times, the time complexity is definitely going to be O(N).

**Space**: O(1)

No need to use frames on the call stack.

We will just use few variables and updating them throughout the algorithm.

**IMP**:

Backtracking iterators paper

https://www.lri.fr/~filliatr/publis/enum2.pdf

Backtrack the iterator to a previous step.

2 lists implementation of functional queues arising from a Zipper based breadth first traversal.

https://cs.lmu.edu/~ray/notes/backtracking/

Start with the simplest tree.

**Situations**:

Going down

* Current node’s parent is the previous node. We don’t care about whether current node is left or right child of the previous node.

Going up:

* Previous node is left child of current node.
* Previous node is right child of current node.

Backtracking situation arises when either left or right child of current node becomes null.

When left child becomes null, we explore the right child.

When right child becomes null, we set

nextNode = currentNode.parent.

This assignment plays an important role in backtracking.

nextNode acts as the iteration variable.

**Flatten Binary Tree**:

Write a function that takes in a Binary Tree, flattens it, and returns its leftmost node.

A flattened Binary Tree is a structure that is nearly identical to a Doubly Linked List (except that nodes have left and right pointers instead of prev and next pointers), where nodes follow the original tree’s left-to-right order.

Note that if the input Binary Tree happens to be a valid Binary Search Tree, the nodes in the flattened tree will be sorted.

The flattening should be done in place, meaning that the original data structure should be mutated (no new structure should be created).

Each BinaryTree node has an integer value, a left child node, and a right child node. Child nodes can either be BinaryTree nodes themselves or None/null.

Left node is like the previous node and right node is like the next node.

Flattened tree is like a flat line.

The flattened tree is obtained by basically ordering the nodes from left to right in the original binary tree.

The nodes are going to be in order.

The prompt tells us this by mentioning the fact that if the binary tree happened to be a valid BST then the flattened tree will have nodes will be in sorted order.

Easier solution takes full advantage of the fact that the nodes in the flattened binary tree are in order.

We are going to traverse the binary tree in order using the in order tree traversal techinque.

Store the in order nodes in a list of nodes. And then we iterate through the list of nodes and update the left and right pointers of the previous node, next node and of the current node and we will have our flattened binary tree.

Order of the nodes in the list is the order of the nodes in the final flattened binary tree.

O(n) time, n is the number of nodes in the input binary tree.

O(n) space.

Can we do better ?

Do not use any auxiliary data structure.

Think about which of 2 complexities we can improve.

The time complexity is unlikely to improve upon because we have got n nodes in this binary tree.

We know that the flattened version of the binary tree effectively involves or requires us to update pointers on n nodes, therefore we may not be able to avoid having a linear time complexity.

O(n) space comes from auxiliary space.

If we traverse the tree recursively, we get O(d) space where d is the depth of the tree.

We have to figure out a way at any given node of the tree to get a reference to the new node that is supposed to come to its left and the new node that is supposed to come to its right.

For a node in the flattened tree,

The left child of the node will be the Rightmost node in the left subtree and right child of the node will be the Leftmost node in the right subtree.

Left child: Rightmost node in the left subtree.

Right child: Leftmost node in the right subtree.

Technically, they cannot be called as predecessor or successor as these terms are used in bst’s.

Formula or hint as to how we can get the left node and the right node of a given node in the final flattened tree.

In an interview, try to create a bigger tree if unable to find a pattern.

This is what ordering nodes in-order entails.

Recursive call to a given node returns the nodes that it has to connect to.

Once we grab the rightmost node of the left subtree and we make it to be this node’s left pointer, we also take the opportunity to grab this node’s right pointer and make it point to us.

O(d) space. Traverse the tree recursively.

At every node, we just did a few things to return references to each node and then updated the nodes as we went through the tree.

We updated the pointers as we traverse through the tree.

For perfectly balanced tree, O(logn) space.

In the code,

Each node stores its left most and rightmost nodes in an array and returns the array to the parent node.

For a leaf node, we assume left most node and right most node are the node itself.

Why?

Because we want to find the left most node of the whole tree in the end which will be a leaf node or a node with single child.

The question wants us to return the left most node of the flattened tree.

(Assuming in order traversal)

Important observation: the left most node and rightmost node of the original tree and the flattened tree will be the same.

Storing an array of both left and right most nodes makes it easier for us.

We can always use constant space in algorithms. Whether be it a single variable or an array of size 2.

To update pointers, we first need the left and right most nodes of the left and right child of the current node.

And after this, we set the left and right most pointers of current node separately.

If the left or right child is null, then we only set the leftmost and right most node of the current node without updating any pointers.

**Right Sibling Tree**:

Write a function that takes in a Binary Tree, transforms it into a Right Sibling Tree, and returns its root.

A right sibling tree is obtained by making every node in a Binary Tree have its right property point to its right sibling instead of its right child.

A node’s right sibling is the node immediately to its right on the same level or None/null if there is no node immediately to its right.

Note that once the transformation is complete, some nodes might no longer have a node pointing to them. For example, in the same output below, the node with value 10 no longer has any inbound pointers and is effectively unreachable.

The transformation should be done in place, meaning that the original data structure should be mutated (no new structure should be created).

Each BinaryTree node has an integer value, a left child node, and a right child node. Children nodes can either be BinaryTree nodes themselves or None/null.

If a node does not have another node immediately to the right of it, we don’t want it to point to a node that is on the same level and to the right.

Some nodes follow a particular pattern when assigning their right child.

Nodes 5 and 10 have to now point to the left child of the new node that there is parent is pointing to.

Identifying patterns, then progressing.

There will be 2 cases.

Assigning right pointer of left and right child of a given node.

For the left child nodes, we have to assign their right pointer to the right child of their parent.

We first do this, and then we move to the second case.

For all of the nodes that respected the first pattern, the first pattern being we are gonna make a node point to its parent’s right child. (Case 1, left child nodes)

The nodes that follow the second pattern, that is the 2nd case of right child nodes,

The pattern changes depending on whether a node is the left or right child of another node.

For a node that is a right child of another node, then the closest node to the right of it would be the left node of our parent’s right sibling.

Iterate through the binary tree, probably use DFS, and at every node, we figure out if the node that we are at is the left child of another node or the right child.

And depending on that, we do the transformation following the 2 patterns that we identified.

Keep track of the parent node in our algorithm.

If we are going to use recursion, we can pass the parent node to all of our recursive calls, or whatever parent node is relevant at any given node.

If we try to do the work in this recursion,

For 9 we will have to go back up 2 levels.

For node 10, we will have to go back up 3 levels.

We don’t have an easy way to grab the right sibling of our parent node.

It would be nice if we already had our parent node 4 pointing to its right sibling.

Instead of doing dfs, let’s just update the parent first and then we will call the recursive calls on the children nodes.

At the 4, since we are are at the left child of the 2, we update its right pointer to point to right child of our parent.

But now if we go the left, we will have lost our right pointer. So we won’t be able to access 9, which is 8’s new right pointer.

We cannot do this.

Maybe, we could pass in the right child to a left child when we make a left recursive call. That seems a little but cumbersome.

The best way to do this is to sequence our transformations properly.

We know that the left child of a given node needs that node’s right child.

If we are at 8, we need our parent’s right child. So we cannot lose this pointer.

But then, once we are at the right child of a parent node, we need that parent node’s right sibling.

We need to update the parent’s pointer to point to its right sibling, after we have done the recursive call on the left node, but before we do the recursive call on the right node.

In-Order

By this sequencing we should access to the right pointers at the right time.

We store the referece to the right child before updating the parent’s right reference.

Root doesn’t have parent, so we make its right child point to null.

O(n) time, we traverse every single node. N is the total number of nodes in the binary tree.

Space: O(d) space, recursive calls

At any point of time, we will have recursive calls equal to the depth of the tree.

For perfectly balanced tree, log(N) space

Think about the base case in recursion.

Here, we only want to update the right pointers for a node.

Big question, Why do we need a boolean ?

The logic of updating pointers for left and right children is different depending on whether the node is left child or right child.

We pass in true by default for the root node as its parent is null and thus its right node will be null.

Another variety of problem.

Here we have an option to do 2 things.

When we come out of a recursive call on a node, then it means that the work required to be done on the subtree of that node is completed.

Here the important base case will be for leaf nodes.

**Note**:

If we try to solve this without using the boolean, then we reach a case for the right leaf node where the algorithm would blow up and we will see the need for a boolean.

We only set the right pointer of a node.

**AlgoExpert solution code explanation**:

**All Kinds of Node Depths**:

The distance between a node in a Binary Tree and the tree’s root is called the node’s depth.

Write a function that takes in a Binary Tree and returns the sum of all of its subtrees’ nodes’ depths.

Each BinaryTree node has an integer value, a left child node, and a right child node. Children nodes can either be BinaryTree nodes themselves or None/null.

Similar to the question of node depths.

Depth of a node is the distance from the root node in a binary tree.

Leaf nodes have no depths when we take the subtree rooted at these nodes.

To sum up all of the node depths in a single binary tree, not the subtrees, we could write a very simple recursive method that basically adds up the depth of a given node.

We start at the root node with a depth of zero.

We add up the recursive calls to the children nodes of the root node and then of every node there after.

And whenever we make these recursive calls, we add 1 to the depth of the nodes.

Sum of node depths for one particular binary tree.

For 1 subtree,

nD(n, d) = d + nD(left, d+1) + nD(right, d+1)

Call this method on every subtree in this binary tree.

Trees rooted at every node.

Iterate through every node in the binary tree, and at every single node, treat that node as the root of a subtree in the input binary tree and call the node depths method on all of these nodes.

Node depths method takes O(n) time as it traverse through all the nodes in the tree that we call it on.

n is the number of nodes in the subtree.

O(h) space where h is the height of the binary tree on which node depths method is called.

For balanced tree, O(nlogn) algorithm (best)

At every binary tree that we call this node depths method on, we are effectively eliminating half of the nodes in the binary tree.

Overall,

Worst case, O(n^2)

Space: O(h), h is the height of the input binary tree.

We should optimize on time complexity.

Can we do better ?

O(n)

Can we solve this problem just by doing a single pass through this entire binary tree ?

Or may be a couple of passes or even multiple passes but not doing a bunch of traversals for each node.

Just singular passes through the entire binary tree.

Can we find some sort of relation between node depths at a given node and the node depths of the children of the node that we are looking at ?

The reason we previously used the recursive method on every node is because it does not look like that there is a relation between the value 16 and the other two values 6 and 2.

If we look at the binary rooted at 2, the depth of the node with value 2 relative to the binary tree rooted at 2 is 0.

The depth of the node with value 2 relative to the binary tree rooted at 1 is 1.

If we are looking at the binary tree rooted at 1 and the binary tree rooted at 2, the depth of the node with value 2 relative to these two binary trees differs by 1.

It gets incremented by 1 if we look at the binary tree rooted at 1.

4 and 5 relative to 1 have a depth of 2.

But relative to node 2 they have got a depth of 1.

Depth becomes one value less than the depth that they have relative to the upper level.

Every single node in the left subtree of node 1, has a depth relative to this 1 that is 1 greater than their depth relative to this 2.

Relative to node 2, if we add 1 to the nodes 4, 5, 8 and 9 then we get their depths relative to the subtree rooted at node 1.

Incrementing all these values by 1 is the same thing as grabbing the sums 6 and 2 here and adding the total number of nodes in the tree that is rooted at the node that the 6 and 2 were obtained from.

Incrementing all by 1 is same adding the total number of nodes.

New formula,

nD = nDL + #L + nDR + #R

To solve this,

We have to figure out a way to compute number of nodes.

nDL will follow the same formula, so computing it is not the issue.

To find #L or #R, we can just do a single pass through a binary tree and compute the number of nodes in every subtree.

We can count the number of nodes in linear time.

This is what is going to allow to solve the question in O(n) time.

Starting at the root node, we will traverse the tree and we will go all the way to the bottom and at the bottom, we have got base cases.

**Base Case**:

Number of nodes in the subtree rooted at the leaf nodes will be 1.

Node depths of leaf nodes will be 0.

We have stored the values of nDL, #L, etc somewhere, either on the node or in some sort of hash table or maybe we have implemented this algorithm in such a way that we return these values.

To get the final value, we can traverse the input tree once and add the stored node depths.

We can implement the algorithm in different ways.

One way could be to do with multiple passes through the binary tree.

One traversal through the n nodes to compute the green values and another to compute the brown values.

For this particular implementation, we will have O(n) time and space complexity.

We will be storing all of these values, storing N extra values or even 2N extra values.

There will also be a way to implement a solution without storing all these values, just by having 1 single recursive method which returns all of these values at every iteration and using them to build our final answer.

With this, we don’t have to store n extra values and we will just have the recursive calls on the call stack that are going to take up auxilliary space.

For this solution, it will be O(h), where h is the height of the tree.

**Max Profit With K Transactions**:

We are given an array of positive integers representing the prices of a single stock on various days (each index in the array represents a different day).

We are also given an integer k, which represents the number of transactions we are allowed to make. One transaction consists of buying the stock on a given day and selling it on another, later day.

Write a function that returns the maximum profit that we can make by buying and selling the stock, given k transactions.

Note that we can only hold one share of the stock at a time; in other words, we cannot buy more than one share of the stock on any given day, and we cannot buy a share of the stock if we are still holding another share.

Also, we don’t need to use all k transactions that we are allowed.

A transaction is defined as buying a share of stock and then selling it.

We can only buy one share of a stock at a time.

We can only buy one share at 5$. We cannot buy two shares.

Similarly we cannot buy a share at 5$ and then buy another share at 11$ without selling the first share.

And one transaction consists of buying a share and then selling it.

Dynamic programming

Build a table. 2d array

Have the prices of the stock in the column and have number of transactions in the rows of the table.

In our table, we will build up the rows starting from 0 transactions.

If we have 0 transactions, that means that we cannot buy or sell a single stock, so the maximum profit that we will be able to make will be 0.

We could buy the stock and sell it the same day but it wouldn’t do any benefit.

When we build the 2d array, the first column is always going to be 0 because if we only have 1 price, we can never make a profit of just 1 price.

When we reach 2nd row, then the calculation becomes harder.

We have to find a pattern that exists to relate all these values together.

For 2nd row and 4th colulmn, we cannot hold a stock that we buy at 5 all the way to the 50 if we want to buy another one at 3.

We can only hold one stock at a time, so if we buy a stock at 5 and we want to buy another at 3, we have to sell the stock that we bought at 5 before buying another stock at 3$.

Whenever we get to a specific column, a specific day with a specific price, we notice that we really have 2 choices.

Either we decide that we want to sell our stock at this price, in which case we are going to have to have bought it beforehand, and we will make the difference between whatever we bought it at and whatever we sell it at, or we say, “No, we don’t want to sell it here”

If we don’t sell our stock at a given price, our max profit is just the same as the max profit on the previous day, which in this case, the max profit on the previous day where we sold at 60 is 63$.

Either we sell at the current price, which is going to be 90, and we are going to have to have bought somewhere before that, or we don’t sell at 90, and we just make the max profit of the previous day.

profit[t][d]: Profit at transaction t on day d.

max { Deciding to sell at price of the current day, Not selling on this day and taking previous max profit

}

profit [t][d] = Max { profit [t][d-1],

prices[d] +

0<= x <d max(-prices[x] + profit [t-1][x] )

When t =2,

On a given day x, we look at the profit that we could make with 1 transaction.

At every iteration of the 2d array, we are applying this formula and comparing two values, one of which further includes iteration to find max value.

To compute a value on 2d array, we do atmost O(n) operations in the formula.

This leads to O(n^2\*k) time.

Can we do better ?

Yes

Whenever we want to try to see if we can do better, we have to check if there is any point in time where we are re-calculating values.

It might not be apparent here, in the way we did this example, if we went through each and every one of these values that we generated, and we applied this formula, and we wrote down the formula what we wrote down here,

x=0:

x=1: ….

we will see that we are re-doing the same work.

Before we generate the value 93, we have to generate the value 63.

To generate 63, what would we do?

For this we do the similar thing,

x=0:

x=1:

x=2:

x=3: ….

And we will get the same values for these.

(We are calculating 4 of the exact same values that we had when d=5.)

We can store the maximum value that we get here.

0<= x <d, max(-prices[x] + profit [t-1][x] )

We can store this part of the equation every time that we iterate through the array.

We compare the new value with the stored maximum previously.

Through this, we eliminate the entire O(n) operation for each value of the 2d array and turned it into a constant time operation.

Time complexity becomes O(nk)

2d-array, O(nk) space.

Can do better in O(n) space.

Optimizing space of 2d array.

To optimize space complexity in a dynamic programming problem, we have to go back to our formula that we are using to build the array and see what values we are using in this formula and what values we really rely on.

We are only relying on values either on the current row we are at and on the previous row.

And we are never really looking at any other row.

Store 2 rows of n length each.

**IMP Note for this problem**: We can buy and sell a stock on the same day.

To sell a stock, we must first buy it.

The second case that we have in the formula,

prices[d] +

max ( -prices[x] + profit [t-1][x] )

0<= x < d

-prices[x] represents that we bought the current stock on day x.

This accounts for 1 transaction.

We add profit [t-1][x] to take into account the previous transactions.

profit[t-1][x] can be formed by selling the stock on day x or not selling it.

If we did sell it on day x, then leads to the case that we sold the stock of previous transaction on this day and also bought a new stock in our next transaction.

Also, the transactions cannot happen in parallel.

2 variations of this problem:

* At max 1 transaction
* Can do as many transactions as we like.

What does it mean to not doing any transaction at all ? What if we already have bought a stock ?

Building the solution.

When we have the case when only 1 day is there and k=2, then profit = 0 as we do both the transactions on the same day with no profit.

IMP NOTE:

For the case when we don’t transact on a given day, then we won’t get a case when we have bought a stock but not sold it.

This is because a particular entry in the 2d array represents that we have already done the required number of transactions on previous days.

If we wish to do the transaction on a given day, then we iterate.

Good question to deal with the fact whether buying a stock on a given day is most profitable.

**Questions to ask**:

Can the array have negative numbers ?

**Palindrome Partitioning Min Cuts**:

Given a non-empty string, write a function that returns the minimum number of cuts needed to perform on the string such that each remaining substring is a palindrome.

Single character strings are palindromes.

For string “noonabbad”,

2

noon | abba | d

Non-empty string.

In this question we are dealing with palindromes and we are dealing with this idea of the minimum number of cuts.

We need 1 cut less than the number of substrings remaining.

What we want to do here is we want to find the palindromicity, whether or not a string is a palindrome or not of every substring in our input string.

For this, we will construct a 2D array.

In this 2d array, we will store the palindromicity of every substring in our input string.

At any given index, the row will represent the starting index of our substring and the column will represent the ending index.

Iterate through our array, do kind of double

for-loops, get every substring in our array, and for each of those substrings, call some kind of isPalindrome() function.

Just need half of the 2d array.

This 2d array tells us: At every starting index, whether the substring is a palindrome ending at the end index.

To build the 2d array, we didn’t use previous solutions to build next solutions.

Let’s build a new array that is going to hold the minimum number of cuts.

This array will be of same length as our input string.

At each index, it is going to hold the minimum number of cuts needed for the substring that starts at index 0 and that goes all the way to the index that we are at.

To initialize this array, we are going to put infinity nine times.

At the beginning we don’t know how many cuts we need for each substring to turn it into just remaining strings.

How do we find the real values that need to be in this ?

For index 0, check if the substring we are looking at here is an actual palindrome.

If a string is already a palindrome, then we don’t need any cuts to get all the remaining substrings to be palindromes.

Subproblems: substrings starting at index 0.

Whenever we get to a substring that is a palindrome, we just store a 0.

Next we check whether “no” is a palindrome.

If our current substring is not a palindrome and we know what the minimum number of cuts was for the previous string, for the string without this letter, which was ‘n’, we could say, that the minimum number of cuts will be the minimum number of cuts for the previous string + 1.

We can put one cut in between “n” and “o” and get palindromes.

Now, “noo” is not a palindrome.

We could say, look at what the minimum number of cuts was for the previous string, “no”

When we reach index 2,

Let’s iterate and check the palindromicity of every substring that starts at each subsequent index and let’s see if we can create a palindrome by adding the current letter that we at.

For “noo”, can we form a palindrome somewhere in between by adding an ‘o’ to the end.

If “oo” is a palindrome, then let’s look at the minimum number of cuts we needed for the index before new starting index which is where first ‘o’ is at.

“noona” is not a palindrome.

We can tentatively put a cut between “noon” and “a” or let’s check if we can do better.

We iterate from index 1.

Check if “oona” is a palindrome.

At every point here, we are trying to check, can we form a palindrome with a previous subtring and take the minimum cuts from the string that comes before it and see if we have a better solution.

We are using previous solutions to build up the next solutions and we are using our palindrome array to help us along the way.

The previous subproblems that we use to build our current solution are already optimal and thus produce an optimal solution.

When iterating, we only care when the new substring is a palindrome because only then we could get a better solution.

O(n^2) time to fill the array. Array has length n and at each indices, we are reiterating through all substrings that come before it.

To fill in the 2d array, we perform O(n^3) time

Slicing a string is O(n) operation.

This O(n^3) swallows O(n^2)

O(n^2) space.

Can we do better ? Yes

We can improve our time complexity.

Can we do better in terms of space complexity ?

No, because we need all those values.

For time, we know that we will have to iterate through n^2 substrings because the number of substrings converges to n^2.

We will have to eliminate isPalindrome() function as it brings us to n^3.

Clever way of building the array and using solutions that we store in this 2d array to build up other solutions.

To build the 2d array in O(n^2) time, we will use dynamic programming tactics without ever calling isPalindrome() function.

The first diagonal in the 2d array is always going to be true as the diagonal represents single letters which are palindromes.

If we are checking if the string is a palindrome, we need to check if the first and last letters are equal to each other and that the string in between those first and last letters is a palindrome as well.

What if we have already stored the solution in our array?

What if we have already stored the fact that the string in between 2 letters is a palindrome or not ?

If we do that, then computation of isPalindrome() function becomes O(1) time operation.

All we are doing is, comparing first and last letter, and then, we are saying, is the string in the middle, in between the first and last letter, a palindrome ?

**Recursive Naïve Approach**:

// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1

minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.

minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be

// calculated recursively using the following formula.

minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +

minPalPartion(str, k+1, j) }

where k varies from i to j-1

Variation of Matrix Chain Multiplication problem.

**Longest String Chain**:

Given a list of strings, write a function that returns the longest string chain that can be built from those strings.

A string chain is defined as follows:

Let string A be a string in the initial array; if removing any single character from string A yields a new string B that is contained in the initial array of strings, then strings A and B form a string chain of length 2. Similarly, if removing any single character from string B yields a new string C that is contained in the initial array of strings, then strings A, B, and C form a string chain of length 3.

The function should return the string chain in descending order (i.e., from the longest string to the shortest one).

Note that string chains of length 1 don’t exist; if the list of strings doesn’t contain any string chain formed by two or more strings, the function should return an empty array.

We can assume that there will only be one longest string chain.

Input: [ “abde”, “abc”, “abd”, “abcde”, “ade”, “ae”, “1abde”, “abcdef” ]

Output: [ “abcdef”, “abcde”, “abde”, “ade”, “ae” ]

To solve this, need practice with DP questions.

Patterns of memoization and caching

Imagine we have two strings in the input list, “abc” and “ab”

The fact that we can remove one letter from “abc”, ‘c’ we get “ab” which is present in the input list of strings.

These 2 strings now constitute a string chain of length 2.

We have to try removing every single letter from every string to see what happens.

But that is pretty computationally demanding. Removing letters from strings is not trivial. It takes time complexity, for instance.

If we don’t optimize our algorithm in some way, then we will have a really terrible algorithm.

If we have string “abcdef” then we will try removing every single letter.

For every single string in the array, we have to try removing stuff, and then redo the same sort of logic for every other string in the array.

The key to cracking this particular algorithm, is to realize that every single string in the array has a longest string chain, where that particular string is the largest string.

Every single string in the array of input strings has one longest string chain for which it is the largest string.

For example, “abde” has some longest string chain where it is the longest string, meaning it is at the head of the chain.

Same for other strings.

Second thing we should realize is that we should only ever compute the longest string chain for each of the input strings once, not twice, not three times, etc.

We should never do duplicate computations.

We should only compute the longest string chains for each of the strings once.

And once we then realize that, by computing them once we can sort of build off of them to compute the string chains of larger strings, then this algorithm becomes almost trivial.

The best way to solve this problem is to do so in a very intuitive way.

And to make it as intuitive as possible, we are actually going to sort our input list of strings by their length.

We have now sorted our input list of strings from the shortest string to the longest one.

If we go off of the two sort of key things that we said to realize in this problem is, we can iterate through these strings, from left to right meaning from shortest to longest.

At every string, we compute the longest string chain at that string.

We are going to try removing letters and seeing if the leftover strings are part of our input strings and so on.

Because we have sorted these strings by length, we can intuitively realize that,

If we come across a string at one point, when we remove a letter, that is in our input list of strings, we will have already computed that strings longest string chain because we are going from shortest to longest.

We can use that to avoid doing a bunch of unnecessary computations.

We can dump the input strings in a hash table to do the look up in constant time.

Instead of dumping the input strings in a hash table that just maps them to true, we are going to have every string mapped to the specialized object, that is gonna have the next string in the chain, for “ae” it is going to be the empty string and the length of the longest string chain starting at that string.

For “ae” it is going to be 1.

We only remove a single letter from a string. And not more than 1.

O(1) lookups from hash table.

Sorting helps in pre-computation.

By sorting it, we make this problem a lot more intuitive and we make prevent ourself from doing annoying checks in our code where we are on.

If we hadn’t sorted them, then it is possible that “ade” would have come after “abde”, then we would have said that “ade” is in our list of strings but we have not computed its longest string chain. And we would have to make a check to know if we have computed it already.

So now we will have to compute it. And it adds a little bit of complication to the algorithm.

We will be iterating through the input list of string once to dump them in hash table.

Time Complexity:

O(N\*M^2 + NlogN )

We iterate through every string at one point, and for every string, we iterate through that string and remove every letter

M: length of the longest string

All the strings will be of length M in the worst case.

When we remove a letter and create a new string without that letter, we are iterating through the entire string to re-build it, that’s another M leading to M^2.

We have sorted the strings by their length.

Sorting made the code intuitive and cleaner.

Tell this to the interviewer:

Unless we are dealing with a tiny M, unless we dealing with tiny strings, it is very likely logN will be smaller than M^2.

Then N\*logN doesn’t affect the algorithm.

Space: O(NM)

We are storing a hash table of N strings, where each string potentially points to another string of atmost length M.

String chain will have a length of atmost N where the strings in the string chain will be potentially starting at a string of length M all the way to a string of length 1.

And this converges to NM.

Bottom Up approach.

In the code, first we build our hash table and finally

for building the final list, we can iterate over the hash table using the input list of strings.

We will need input list of strings to iterate over the hash table.