**Algo Expert**:

Dividing 2 big numbers.   
There are many libraries in Java. Try to think of multiple solutions to the same problem.

Main focus is to solve problems.

int index = Arrays.binarySearch(sortedArray, key);

// works for arrays which can be of primitive data type also.

int index = Collections.binarySearch(sortedList, key);

// works for Collections like ArrayList and LinkedList.

If not found then index will be negative.

Arrays.copyOfRange(array, j, array.length+1)

If a subarray is being returned, be careful with its indices and the original array’s indices.

Subarray’s 0th index could be kth index of bigger array.

**int** answer[][] = **new** **int**[m][n] ;

**for** (**int**[] row : answer)

Arrays.*fill*(row, -1);

**int** x[] = **new** **int**[2];

Arrays.*fill*(x, -1);

Way to initialize a 1d and 2d array.

To clear a list, use the clear() method.

Printing a list displays the value while printing an array prints a hash code.

Make a diagram for thinking.

2 number sum, 3 number sum

Can solve these using a Hash Table.

Sorting and then O(n^2).

Better to use iterators rather than for loops.

In place algorithm: algorithm which transforms input using no auxiliary data structure.

int[] array

arr.length

List<Integer> array

array.size()

To convert an array to a List, we use Arrays.asList().

Move Element to End:

We can do this in O(n) time instead of O(nlogn) time.

For O(n) time, we use two pointers, one for the starting index and another for last index.

We use a pointer to the last index because we want to move the required number towards the end of the array.

**BST Construction**:

Insertion:

Average: O(log(n)) time, O(log(n)) space

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

Search:

Average: O(log(n)) time, O(log(n)) space

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

Deletion:

Average: O(log(n) time, O(log(n)) space

Worst: O(n) time, O(n) space

**IMP**: If we want to make a function that returns something but not able to think about the return logic at some places, then what we can do is create another function which is called inside the current function and this function might not return anything.

Can also implement the above 3 in O(1) space.

In a binary search, what if we make the root to be any random node and start the operations from there?

In deletion, if we can’t find the element then we don’t do anything.

For deleting node with 2 children, grab the left most node in the right subtree of that node or grab the right most node in the left subtree.

Replace the left most node value with the node to be deleted. And then call delete on this leftmost node (deletion for with 0 or 1 children).

If we run the algorithms recursively, then space complexity will be O(log(n)) in average and O(n) in worst case.

Recursion uses frames on the call stack.

See the deletion code using both recursion and iteration.

**Validate BST**:

Base case would be on leaf nodes. Leaf nodes are always binary search trees.

Can use divide and conquer.

Validate all the possible subtrees.

Create a helper method.

If we want something to be true and want to continue in our code when that condition is true then create an if when that condition is false. And then write the logic for continuation.

**BST traversal**:

In order: Left, root, right

Pre order: Root, left, right

Post Order: Left, Right, Root

Using Recursion:

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

Be careful with returns in Recursion.

Arrays.copyOfRange(x, 1, arr.length);

// returns an array from [ 1, arr.length-1 ]

**Maximum Subset Sum No Adjacent**:

**Monotonic Array**:

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

We can use some variable to make the code cleaner.

**Spiral Traverse**:

We want to visit every vertex only once.

O(mn) time, O(1) space

mn is the number of total elements.

Think cases.

In 2d array, taking 4 pointers can be helpful.

The final iteration can end with a single element, row wise, column wise or normally in a complete spiral.

If the goal is to visit all the elements, then we can take a variable which stores the number of elements visited.

**Longest Peak**:

Find the longest peak.

When looking for a strictly increasing sequence, if we find a decreasing number then we know that we have found a peak.

Store the length of the peak and compare with other peaks and return the longest peak.

java.util

“java”: Reference Variable.

Corner Case: When finding the strictly increasing the first time. Will have to increase the current length by 2 instead of 1.

For debugging, can use a list to store all the possible peaks one at a time and print them when they a particular peak has been constructed.

**Code Logic**:

Start with finding a strictly increasing sequence. If found then store the current value and look for strictly decreasing.

When looking for decreasing, the current length has to be greater than 1.

A peak is only possible if we are able enter the decreasing logic.

The corner case of incrementing the current length by 2 can also be seen in some other problems that require finding lengths.

Also see video solution.

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

**Maximum Subset Sum No Adjacent**:

Write a function that takes in an array of positive integers and returns the maximum sum of non-adjacent elements in the array.

DP question.

1d array cache.

Index of the cache represents the Maximum non adjacent subset sum having elements till given index.

Base Case: Fill the cache with the array elements itself.

After iteration through non adjacent solution, check whether the built solution is smaller than the adjacent previous solution.

If it is smaller then take that solution for this index.

The above approach takes O(n^2)

Can solve this in O(n) if the elements are all +ve.

O(n) time, O(n) space

Can reduce the space to O(1).

**Number of Ways To Make Change**:

Unlimited supply of coins.

Array of positive integers representing coin denominations and a single non-negative integer n representing a target amount of money.

What if the supply of coins is not unlimited ?

In dp, be careful with the index numbers.

The ith index in the cache could be the i-1th index of the denomination array.

**Min Number of Coins For Change**:

Return the smallest number of coins needed to make change.

Unlimited supply of coins.

**Important condition**:

When it is impossible to make change for the target amount, return -1.

Integer.***MAX\_VALUE*** + 1 gives Integer.MIN\_VALUE

Integer overflows, so be careful.

See the solution of Spiral and peak question.

In Dynamic programming, we can fill the cache row-wise or column wise.

**Min-Edit Distance**:

Using DP:

O(mn) time, O(mn) space.

Can reduce the space to O(min(m, n) )

See AlgoExpert video solution for this.

**Kadane’s Algorithm**:

Requires atleast one positive number.

O(n) time, O(n) space

Can also solve it in O(1) space.

**Single Cycle Check**:

Check whether the jumps in the array form a single cycle.

A single cycle occurs if, starting at any index in the array and following the jumps, every element in the array is visited exactly once before landing back on the starting index.

Cases:

There could be more than one cycle and not all elements may be visited.

**Constraint**: Visit all elements once and return to the starting index.

Here, we won’t iterate in an ordered fashion, so we should use a simple while loop or can also do the same thing using a for loop.

Math.ceil takes double as operands and not integer.

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

**Breadth First Search**:

Queue

Can use an array list as a simple implementation or

Can use LinkedList as an implement of Queue

**River Sizes**: or Number of Islands

Google interview question.

2d matrix of mxn containing only 0s and 1s.

Each 0 represents land and each 1 represents part of a river.

A river consists of any number of 1s that are either horizontally or vertically adjacent (but not diagonally adjacent). The number of adjacent 1s forming a river determines its size.

Write a function that returns an array of the sizes of all rivers represented in the input matrix. The sizes don’t need to be in any particular order.

**Approach**:

If we are at a 1, then explore downwards, rightwards, upwards or downwards. Mark the explored 1 as visited.

We need to explore the possibility of going in all 4 directions as there will be cases in which there will be 1s on the left and upwards.

Here the use of boolean[][] is very important.

When we are at a ‘1’ then going in either direction can lead to a ‘1’ which has already been explored. That’s why boolean[][] saves us.

O(mn) time and O(mn) space.

**Youngest/Lowest Common Ancestor in a binary tree**:

For a node, “ancestor” property points to their youngest ancestor.

Given 3 inputs, the first input is the top ancestor in an ancestral tree (the only instance that has no ancestor – its “ancestor” property points to null), and the other 2 inputs are descendants in the ancestral tree.

We have to find the youngest common ancestor to the 2 descendants.

From the point of youngest common ancestor, the path diverges in 2 different directions.

**Note**: Here the tree is in form of left and right pointers.

One solution is to traverse the path from the root node till the two nodes and see what is the common path between them.

Keep checking till the point at which the path diverges.

This requires extra space.

Using another approach, we can do this in linear time without taking extra space.

We search for either of the 2 nodes whose lowest common ancestor we are looking for starting from the root node. Any time, any of the nodes is found, we return that node to its parent in the binary tree.

Any time, any node gets a not null node from the left and a not null node from the right side, it knows that it is the lowest common ancestor and then it returns it’s node value to it’s parent.

(2, 5)

3

/ \

6 8

/ \ \

2 11 13

/ \ /

9 5 7

We start from 3, and check if 3 is either 2 or 5.

3 is not same as either 2 or 5, then we expand our search and go on the left side.

Check if 6 is same as 2 or 5. No, then go to the left side.

Check is 2 is same as 2 or 5. 2 is same as 2.

From this point, we return the node 2.

From 6, we go and search on the right side.

Check whether 11 is same as 2 or 5. No, then go to left side.

9 is not same as 2 or 5. We have reached a leaf node. So return null.

Go to right side.

5 is same as 5. Return node 5 to node 11.

11 cannot be the lowest common ancestor because it is getting a null from one side and not null from other side.

11 returns the not null node and returns it to 6.

6 gets not null node from both the sides.

6 returns itself to the parent. Now the root 13 expands the search to the right side. It gets null.

The final root returns 6 which is the lowest common ancestor.

We can optimize the search when we backtrack and reach the root node.

O(n) and O(1) space.

**3rd variation**: The tree is not necessarily a binary tree.

Suppose in a particular node we are only storing the youngest ancestor of the current node instead of left and right.

What does youngest ancestor of a node mean ?

Depends on the situation.

Can be the node itself or the first parent.

Let’s suppose it’s the parent.

We traverse upwards using the ancestor pointers.

When one node is the ancestor of the other, we return the upper node.

I initially did this using 2 lists.

See AlgoExpert video for O(1) space.

**Video**:

Graph problem and more specifically tree problem.

In this problem, we can only go upwards.

Get the two descendants on the same level in the tree. (T, I)

In the given example, we have to get node ‘T’ to the same level as node ‘H’

We have to bring, whichever descendant is lower than the other to the same level as the higher descendant.

Calculate the depth of two descendants in the tree and then add logic for equalizing the ancestor.

We can create any kind of tree by just having a reference to the parent node.

After reaching at the same height, keep traversing up until we are at the same spot (youngest common ancestor).

When one node is the ancestor of another node, then we return the higher node.

When working with make the levels equal,

When iterating upwards from the lower node we check whether we have reached the other node or not (For the case when one node is the ancestor of other).

What if the tree is a binary search tree ?

**Construct a Min Heap**:

Data structure

Not an abstract data type.

Priority queue abstract data type.

Heap is an implementation priority queue.

Abstract means that we can do it any way we want and do not care about the behaviour.

Heap can be used in problems where we need to find maximum or minimum value.

During insertion, we add an element at the last position.

When we add or remove items, we restore the heap.

**Property of min heap**: Parent is smaller than the child.

When inserting an item, we need to heapify upwards.

When deleting an item, we replace the last item with the root as we want the tree to be complete.

Now we heapify downwards.

If a node does not have a left child, then it is guaranteed that it won’t a right child as well as the tree is complete.

O( log(n) ) work in insertion and deletion

Build a Min Heap from an input array of integers.

Inserting integers in the heap.

Removing the heap’s minimum/root value.

Peeking at the heap’s minimum/root value.

Sifting integers up and down the heap, which is to be used when inserting and removing values.

Initially, we could have a random array with random elements. These elements may not satisfy the heap properties.

We iterate over some of the parents and heapify downwards to convert the given array into a heap.

**NOTE**:

It’s important to understand why we heapify downwards and not upwards when converting a given array into a heap.

**Remove Kth Node From Node**:

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

Strange behaviour when trying to delete the first element, that is k = size of list.

**Permutations:**

Distinct integers.

contains() method could be helpful to avoid Out of bounds exception or null pointer exception.

Also see the swap technique.

Use of “new” operator.

Also see the swap technique in the video solution.

**Powerset**:

Array of unique integers. Powerset is the set of all subsets.

**Search In Sorted Matrix**:

Distinct integers with sorted rows and sorted columns.

Height is not necessarily same as width.

Return an array of the row and column indices of the target integer if it is container in the matrix, otherwise return [ -1, -1 ]

Also see its video solution.

**Min Max Stack Construction:**

* Pushing and popping values off the stack
* Peeking at the value at the top of the stack
* Getting both the minimum and the maximum values in the stack at any given point in time.

All methods should run in O(1) time and space.

Create 2 lists.

One for the stack and another keeping track of min and max for a particular number when it is at the head of the stack.

**Balanced Brackets**:

String made up of brackets and other optional characters.

A string is said to be balanced if it has as many opening brackets of a certain type as it has closing brackets of that type and if no bracket is unmatched.

Note that an opening bracket cannot match a corresponding closing bracket that comes before it, and similarly, a closing bracket cannot match a corresponding opening bracket that comes after it.

Constraint: Brackets cannot overlap each other.

Stack problem

When we find a closing parentheses, 2 things need to happen.

It need to be the same type as what it is closing (one of the ‘n’ parentheses closing type)

And, it needs to be matched to an opening parentheses.

When we find a closing parentheses, we know that it has to match the most recent left parentheses that we see.

When there is a mismatch, we return false.

We are only storing opening parentheses in the stack.

If there are other characters like a number as part of the string, then we can create a list of all the parentheses characters and do nothing when other such character is encountered.

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

We need to check whether the stack is empty or not before peeking as peeking on empty stack could throw a null pointer exception.

Can do the balanced parentheses problem without a stack also: by having unique “open” counters for each parentheses.

**Longest Palindrome Substring**:

O(n^3) time, O(1) space

Can also do in O(n^2) time and O(1) space.

Brute Force: Get all substrings and get the longest palindrome.

This method runs in O(n^3). How ?

Expand from the middle.

Take every character as the middle of the substring and check whether a bigger palindrome is possible or not.

Here, we will not be checking every substring. We will be checking every centre of a potential palindrome.

Palindrome has to have a centre somewhere.

This centre could be the centre of an odd length or an even length palindrome.

**Group Anagrams**:

Anagrams are strings made up of exactly the same letters, where order does not matter.

Write a function that returns a list of anagrams in no particular order.

More optimal solution is easier or intuitive than the other solution.

Groupings do not need to be ordered in a specific way.

There could be different groupings of 2 strings with same lengths.

One approach could be to sort the individual strings.

And then sort the whole array.

How will we map the sorted strings to original array?

Create a new array of indices.

Time: O(WNlogN + NWlogW)

Space: O(WN)

To sort individual strings in the array, we can convert the individual string to a character array and sort this character array using Arrays.sort().

Now we sort the whole list.

We need to know the indices of the sorted list in an order.

For this we sort the indices list by passing the comparison of words in the comparator.

Lists have their own sort method. No need to use Arrays.sort for lists.

**Another approach**:

Once we have sorted every word, that is once we have found all the anagrams, we can put them in a hashtable and have all the anagrams bucketed together.

We iterate through all of the words. At each word, we sort its letters. We check our hash table to see whether we already have the anagram in there.

If we don’t have an anagram, we start a new group of anagrams or words.

O(WN): creating a hashtable.

Time: O(WNlogN)

We will be iterating through W strings/words and for each string, we sort the letters in the string which will be atmost NlogN

IntStream

**Suffix Trie Construction**:

Construct a class. The class should have a root property set to be the root node of the trie and should support:

* Creating the trie from a string
* Searching for strings in the trie.

Every string added to the trie should end with a special endSymbol character: “\*”

Suffix trees

“babc”

Root node: empty string

The child nodes contains the suffixes of the string.

We mark the end of every suffix with \*

We will search for a string inside the suffix tree.

We can create the suffix tree using Hash map.

Every node in the suffix will be a key in the hash table pointing to another hash table.

All of the values in the hash table to which it points will be other nodes in the tree whose keys will be a specific letter which comes after the previous letter and points at another hash table and so on.

Initially we have an empty hash table.

We iterate through a suffix “babc” and at each index starting at index 0, we say, let’s look at the entire substring starting at this index and add it to the suffix tree.

We iterate through each of the character of a suffix

Starting at index 0 of the given string, we check whether the letter ‘b’ is already stored in the tree.

We add ‘b’ to a branch.

That is in the hash map, we add a key which will be letter ‘b’ and currently it would point to another empty hash table.

As we build our suffix tree, we move along with it.

Now we move to letter ‘a’, we check whether ‘a’ is contained in the current node. Our current node moved a level down and is now at node ‘b’

2 for loops.

We iterate through all of the suffixes. When we start with a new suffix, we again start from the root node of the tree.

A node with more than 1 branches is basically a hashmap which has more than 1 key.

keySet(),

We can reuse already constructed suffixes if we want to build a new suffix.

**Searching**:

Start at the root and check whether that letter is contained in that node.

Also check for asterisk if we reach the end.

Original string: “babc”

If we search for “bab”, we should not find it. This could be done correctly if we check for \* at the end.

O(n): n is the length of the string we are searching for in the suffix tree.

Suffix tree only contains the suffix, so if we search for a non-suffix string, search function should return false. O(1) space

**Creation**:

Time: O(n^2) where n is the length of the input string we are looking for.

We are iterating through all of the suffixes. double for loops.

O(n^2) space, if we have repeated characters then some space can be saved.

If all the characters are same then the space will be O(n).

A real suffix tree can be built in O(n) time.

Trie could be useful in problems which require searching for strings, matching strings, etc.

**NOTE**: We can create a tree having nodes with any number of children using a HashMap.

**AlgoExpert solution walkthroughs**:

Trying other solutions as well.

3 number sum: can assume distinct integers.

O(n^2\*logn) time and O(n) space

3 for-loops can also work. O(n^3)

Can do better: O(n^2) time and O(n) space

**Questions to ask**:

Can a number be used any number of times?

Here, A number can be used any number of times as long as the required triplet is distinct.

Can use a hash table. But will be using extra space.

Sort the array. Use left and right pointer.

**2 number sum**:

O(n), O(nlogn) time

For O(n) time, we can use a hash map.

Interesting use of hash map: O(1) insertion and retrieval.

**Smallest Difference**:

2 non-empty array of integers. Find a pair of numbers (one from each array) whose absolute difference is closest to 0, and returns an array containing these 2 numbers, with the number from the first array in the first position.

Assumption: there will be one pair of numbers with the smallest difference.

**Brute Force**: Calculate all possible pairs of numbers and calculate their difference and pick the smallest difference.

Sort both the arrays.

Array manipulation. Take advantage of properties of a sorted array.

If the 2 numbers are equal (edge case), then that is the pair with the smallest difference.

**Move Element to End**:

Given an array of integers and an integer. Write a function that moves all instances of that integer in the array to the end of the array and returns the array.

Can sort the array and count the number of occurrences of the number we have to move.

This will take O(nlogn) time

We can do better.

Can do this in O(n) time.

We can take one pointer pointing to the start and another pointer pointing to the end of the array.

We take the pointer pointing to the end of the array because we want to move the required number to the end of the array.

Monotonic Array:

Think about the complexity of an optimal solution.

We can find the direction of the array (that is non-increasing or non-decreasing) by finding the difference of the first 2 distinct numbers.

If the starting elements are equal, then we still need to keep looking for a direction. Initially, we can set it to 0.

Once we get a direction, then we check whether this direction breaks once we traverse further in the array.

This can help the code to make cleaner.

**Spiral Traverse:**

Can be solved both recursively and iteratively.

Can keep track of direction and update it when we reach the edge.

Another approach: we traverse the outer perimeter of the array.

Once we traverse, we traverse the inner perimeter of this outer array.

Once we are able to traverse a perimeter, then we can apply the same logic to traverse the inner perimeter.

Starting row, ending row, starting column, ending column

Have to apply the same logic for inner perimeter.

Have to move the pointers inwards.

If starting row and ending row are equal but starting and ending columns are not, then the inner perimeter just has a single row.

If starting column and ending column are equal but starting and ending row are not, then the inner perimeter just has a single column.

If all 4 are equal then we are left with just a single element.

Once the pointers cross, we are done.

**Longest Peak:**

Peak: Atleast 3 integers that are positioned in such a way that the middle is the largest of the 3.

When traversing, we have to restart a potential peak when the properties of a peak get violated by incoming elements.

2 tasks: Find all the peaks and compare the length of peaks and get the longest peak.

Ask the interviewer whether there is more than 1 optimal peak (that is 2 peaks with maximum length)

Find a peak point,

A peak point is greater than its adjacent elements.

Check all indices for a potential peak.

First and last elements cannot be a peak.

Can merge the 2 tasks.

**BST Construction**:

When working with a BST, ask the interviewer about where the duplicate values will go, whether in left or right ?

Successor or Predecessor of a node in a BST.

**Validate BST**:

Validate all the subtrees.

Can do a Breadth first search for traversing elements.

list.size()>0 is slower than list.isEmpty()

**Invert BST**: Convert a BST into its mirror image.

**Levenshtein distance**: O(mn) time

Reduce space from O(mn) to O(min(m, n))

We insert a letter in the target string.

We can delete a letter from the source string.

We replace the last letter in the target string with the last letter in the source string.

Formula is a constant time operation.

Find the logic in DP.

In dynamic, think about whether we need the 2d array or cache.

We are only using values stored in our current row, current column and previous row and column.

We are using 2 entire rows. Reminded me of one optimization discussed in Stanford course on Algorithms.

All we need is 2 last rows. The complexity becomes O(min(M, N))

We can swap the rows and columns if we want, it doesn’t matter. We can fill the entries row wise or column wise.

When 2 references point to the same object (array in this case, then changing the object through any of the reference has an impact on both of the references.

**Kadane**: O(n) to O(1) space optimization

Can just use 2 variables.

“maxSoFar”, “maxEndingHere”

**Search in a Sorted Matrix**:

Can do better than O(mlogn) because the matrix is sorted both row and column wise.

We can start at top right corner or bottom left corner because only then we can move in 2 different directions on < and > logic.

**Palindrome question**:

Are special characters to be considered (comma, etc) ?

Single character is a palindrome.

**Manacher’s Algorithm**:

Find Longest Palindromic Substring in O(n) time

We don’t need to manually go to each index and expand to check the expansion length every time.

Manacher’s algorithm optimizes better than brute force, by using some insights on how palindromes work.

Can go from O(n^2) to O(n) using few insights that Manacher’s algorithm uses.

**Insight 1**: Expanding around the centre to find a palindrome.

Current algorithm: O(n^2)

What is hurting us?

* Expanding for every centre is hurting us.

What if we cut down the number of expansions?

Need to understand concepts of palindromes even better.

**Insight 2**: Palindromes are symmetric at the centres.

Using symmetry, copy the palindrome lengths at mirror centers.

If we add more characters at the boundaries, then palindromic lengths for each of the centre can change.

Once we copy the length of the mirror, we have to start expanding from the end of that length.

We know that this would be the minimum length.

**Code**:

Add some special character in between every character and also at the ends to denote an empty space. This is done to consider the case of even length palindrome.

In the worst case, we will do 2 iterations of the string. We are not expanding every centre till the end of the string. O(2N)

Also see the video solution of longest palindrome substring on Algo Expert.

Can also solve this problem using suffix trees in O(n).

**Longest Increasing Subsequence**:

In O(nlogn) time instead of O(n^2)

Space: O(n) in both cases