**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.

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.