BNY INTERVIEW EXPERIENCE:

RANVEER ARAKHARAO

B. Tech TY EXTC 2023 – 2027

Summer Internship 2025 [On-Campus]

Location: Pune

Stipend: 75K /month

[In – Person Campus Drive]

ELIGIBILITY CRITERIA:

1. CS, IT, ECE and Mechanical students.
2. CGPA:  7 or equivalent to [Till 3rd semester] 70% & above across all academic levels - 10th, 12th, No standing backlogs.

TIMELINE:

1. Registration Process: 16th July.
2. Online Assessment: 27th July.
3. Offline PPT: 4th August.
4. In-person Interview: 4th August [Just After the PPT].

ONLINE ASSESSMENT:

There were total of 4 coding questions:

It was divided into two sections:

Section 1: 1st problem was on Medium side, 2nd problem was on Medium to Hard side.

1. Problem Statement:

*Team of the Century*:

You are given two integer arrays: *lower* and *upper*.  
Each array represents constraints for developers based on their skill levels.

Let *n* be the number of developers, where the skill level of the *j*th developer is simply *j* (i.e., 1 ≤ *j* ≤ *n*).

Your task is to select the **largest possible team** such that every selected developer satisfies the following conditions:

For the *k*th developer (1 ≤ *k* ≤ *n*), they will join the team **only if**:

* At most *lower[k]* team members have a **lower skill level** than them.
* At most *upper[k]* team members have a **higher skill level** than them.

My approach:

It took me the first 5–6 minutes to understand the problem statement, it was quite confusing, to be honest.

For every team size from 1 to n, we would iterate and check if that team size is possible.

The *isPossible* function inside the loop works like this:

We consider a virtual array where we are trying to place developers from index 0 onward (say *idx* = 0). For each developer *j*, we check:

*lower[j] >= idx && upper[j] >= currentSize - 1 - idx*

This ensures that the *jth* developer can sit at the current index without violating the constraints (i.e., not too many people less skilled before them and not too many more skilled after them).

If the condition passes, we increment *idx*. If we manage to assign *currentSize* such developers, then it's possible to form a team of that size.

But wait, this gives a time complexity of *O(n \* n)*, and considering the constraints, it was only passing half of the test cases.

Then it struck me, since we are checking ranges (whether a size is possible or not), we can use **binary search on the answer**, which is a classic optimization for such problems.

So I applied binary search on the team size from 1 to n, and for each size, used the same *isPossible* logic.

This brought down the time complexity to *O(n \* log n)*, and it passed **all** the test cases.

1. Problem Statement:

*Stringy Stringy:*

You are given a **binary string** str, which contains only characters '0' and '1'.

In **one operation**, you can select any **contiguous substring** of str and **flip all its bits** — i.e., change '0' to '1' and '1' to '0'.

The **"happiness"** of a string is defined as the **number of '1's** present in it.  
For example:

* "10101" has a happiness value of 3.

Your task is to **count all possible unique values of happiness** that can be obtained by performing **at most one operation** (i.e., zero or one flip) on the given string.

My Approuch:

Just by looking at the problem, I knew I needed to use Kadane’s algorithm as I’ve solved a similar problem before in a LeetCode or CodeChef contest (can’t remember exactly).  
The brute-force way is to actually do what the question says, try every possible flip and calculate the number of 1s. But considering the constraints, I directly went for the optimal solution.

First, I counted the number of original 1s in the string to preserve the original happiness, since we’ll be flipping and shifting around it.

Then I created an int[] ref = new int[n], where n is the length of the string.

* If the character is '0', I stored +1 in the array (since flipping 0 to 1 adds to happiness).
* If the character is '1', I stored -1 (since flipping 1 to 0 subtracts from happiness).

Now the problem reduces to finding the best subarray to flip that gives either the maximum increase or minimum decrease in happiness. For that, I applied Kadane’s algorithm.  
While iterating, I tracked both the maximum and minimum subarray sums:

lower = Math.min(lower, ref[i] + lower)

upper = Math.max(upper, ref[i] + upper)

Then I shifted the results back to the original base using:

lower = Math.max(0, original + lower)

upper = Math.min(n, original + upper)

Now I had the valid range of happiness values. So the final answer was:

upper - lower + 1

This worked and passed all the test cases.

Section 2:

This section also consists of two problem which were Hard I believe If you haven’t solved similar problem it were quite difficult to came up with solutions which were optimal considering the constraints.

1. Problem Statement:

*How Geek Am I*

You are given an integer array int[] ref and an integer n representing the size of the array.

Your task is to **return the geekiness of the array**.

**Definition:**

* An array is said to have **geekiness** if it contains **geek numbers**.
* A **geek number** is defined as the **count of pairs** (i, j) such that:
  + i < j
  + gcd(ref[i], ref[j]) > 1

In other words, for each element at index i, you look for elements at index j > i such that their **GCD (Greatest Common Divisor)** with ref[i] is **greater than 1**.  
The **geekiness** of the array is the total number of such valid (i, j) pairs

My approach:

Id recommend you to first watch this two videos of Striver:

1. <https://www.youtube.com/watch?v=aBxjDBC4M1U>
2. <https://www.youtube.com/watch?v=g5Fuxn_AvSk>

If you know these two concepts well, this question was a piece of cake.

GCD(a, b) > 1 Which directly implies we’re talking about **common prime factors**

The idea is to **build a graph** where elements are connected if they share a **common prime factor** (since that guarantees GCD > 1).

To do this efficiently, I used **Disjoint Set Union (DSU)**, also known as Union-Find. With **path compression and union by size**, each operation takes nearly constant time (O(4α), where α is the inverse Ackermann function).

Here’s how I approached it:

* For each element i in the array, I computed all of its **prime factors**.
* Using a **HashMap**, I mapped each prime factor to the first index that had it.
* Then, for every new element j that has the same prime factor as a previously seen element i, I simply union(i, j).

After processing all elements:

* I created a result array of size n
* For each index k, I found its representative using find(k)
* Then filled the result using the size of the set:

for (...) {

res[k] = size[find(k)];

}

Hopefully this also helped me passed all the testcases.

1. Problem Statement:

*Only that left is my nodes:*

You are given a **tree** (an undirected, connected, acyclic graph).

Your task is to **find the minimum possible height** of the tree such that **at most bun operations** are used.

**What is a bun operation?**

A **bun operation** is defined as:

* Select a child node u of a parent node v (where v ≠ root).
* Remove the edge between u and v, i.e., (u, v).
* Reattach u as a **direct child of the root**, effectively reducing its depth to 1.

After applying **at most bun operations**, you must return the **minimum possible height** of the tree.

When I read the problem, I understood it’s about minimizing the height of a tree using at most “bun” operations. The bun operation is simply removing an edge (u, v) and making u a direct child of the root, which clearly reduces depth.

Straight away I thought of binary search on the answer. We want to find the minimum height possible, so we binary search on height from 1 to n.

Now the key part is writing a check function to see if a given height is possible using at most “bun” operations.

I used a DFS where for every node, I calculated the height of its subtree. If any child returns a height more than the current mid (i.e., target height), that means we’ll need to apply a bun operation to pull it directly under the root. For each node, I count how many such operations are needed in total.

If total operations used exceed the allowed bun, then that height is not possible.

Finally, if a height is possible, we try for smaller one (go left); otherwise we go higher (go right).

This gave a total complexity of O(n log n) and passed all the test cases.

I was managed to solve all the combine testcases and submitted the test before 30-40 minutes. Also before that I cross checked all the solutions again for my OCD.