ACP
Christian Alcala
Jose Lorenzo M. Cansana
Sean Pe
CSALGCM S15

Problem A: Dragon of Loowater
Status: Accepted
Difficulty Rating: 2
Solution Sketch/Write-Up/Narrative:

In the dragon of loowater, we are given 2 sets of data: one are the knights' height in centimeters, and the second set are the size of the dragon's heads in centimeters. In order for the knights to slay the dragon, they must be at least as tall of the size of the dragons head. To optimize the problem, we must use the lesser total of heights to slain the dragon as this will also be the amount that the King will pay the knights.

Knowing this, first, we must sort the two sets of data in ascending order so that the smaller integers are first. After which, we compare both sets (in this case, we use pointer N for the height set and M for the dragon's head set) by comparing the Nth and Mth element in the list. If the Nth element is greater than the Mth, we increment both pointers, otherwise, we increment N only. When we increment N and it reaches the end of the set while M is not at the end of the list then we can say that there is not enough knights to slay the dragon, outputting 'Loowater is doomed!'. If M reaches the end of its set regardless if N reaches its end or not, we can say that the dragon is slain. If the dragon is slain, we return the total of the heights of the knights on and before the pointer N. This is the least amount that the King has to pay for the services of the knights.

Problem B: Association for Computing Machinery
Status: Accepted
Difficulty Rating: 3
Solution Sketch/Write-Up/Narrative:

The Problem:

For this problem there are two things that we want to find for the optimal solution; how many problems we can solve, and the order which we can solve them for a minimal cost. The problem statement is a team which needs to solve as many problems as they can given that they know the time it will take for them to solve. The penalty is computed by getting the time of a successful submission of a problem (relative to the start of the contest), then getting the summation of each time that a problem was submitted. From this, we can establish a couple of things that can help us solve the problem;

1. The combined time it takes to solve all the problems we can solve will not exceed 300 minutes, the allotted time for the contest
2. In order to minimize the penalty we accumulate, we need to solve and submit problems as soon as possible. This means that *the order we solve is important.* For example, if it takes us 10 minutes to solve one problem and 1 minute to solve another, and we try to solve the 10 minute problem first, we will submit the first problem 10 minutes into the contest and the second problem 11 minutes into the contest, meaning that our penalty is 21. If we solve the 1 minute problem first, our submissions will occur at 1 minute and at 11 minutes into the contest, meaning that our penalty will only be 12.
3. Also, the team has already selected a problem that they want to solve, so we solve that first regardless and continue our solving from there.

<u>The Solution:</u>

Given our two established tenets, we can immediately see that this can be solved with a greedy algorithm. Our locally optimal solution would be to find all the problems that, when added together, would be less than 300. We can already tell intuitively from here that we should get the lowest solving time of all the problems, so that means sorting in ascending order, since we are trying to maximize the number of problems we can solve given the time limit. This also fortunately gives us the optimum way to solve our problems since we can also consider minimizing the cost. In order to minimize our cost, we simply need to solve the problems we can do faster first and the longer ones last, since that minimizes our penalty, as shown in the example in tenet 2.

Here are our steps to solve (after boilerplate):

We need to keep track of the total time spent solving and also our accumulated cost, as well as the number of problems we have solved, so these will be the variables we will be making modifications to and outputting.

1. Sort the array of solving times using an efficient sorting algorithm
2. Since the team has already decided on which problem they want to solve first, we remove that problem from the array and add the penalty for solving that problem.
    a. If the time it takes to solve that problem already exceeds the allotted contest time, then we output 0 and 0 since we cannot solve any problems within the contest time, and our penalty will be 0 for not submitting anything
    b. If it can be solved within the time limit, we increase our count of problems we can solve by 1.
3. After step 2, we then add the cost of our next problem to our current amount of time spent solving to our submission time.
    a. If the submission time for the problem will exceed 300, we don't count it as a problem we can solve and go to step 5.
    b. We then increment our count of problems we can solve if the submission time of this problem is not past our time limit, 300.

4. We then increment our penalty time by the submission time of the problem, then repeat step 3 until we hit our exit condition indicated in step 3a.
5. Finally, we output how many problems we can solve and our accumulated penalty time

This solution is an O(n log n) solution, due to the sorting, but the actual computation itself is only O(n).

Problem C: Shopaholic
Status: Accepted
Difficulty Rating: 3
Solution Sketch/Write-Up/Narrative:

In this problem, we are trying to minimize the spendings of the shopaholic friend by taking advantage of the discount. The discount gives the least expensive product for free in a group of purchases of 3 only. No discount is given in a purchase of 2 or less products.

With these, we can say that in order to get the most discount from a set of purchases, we need to group the greater amounts first to yield a bigger discount compared to grouping the least expensive first.

For example, in a set with 100, 200, 300, 400, 500, when we group the most expensive first, we get 300, 400, 500, and in this set the discount will be 300. If we group the least expensive first, we get 100, 200, 300, and get only 100 discount. In general, we can yield the best discount by getting the 3nth element starting from the back. For this to be possible, the set must be sorted in ascending order.

Total Items: 10
Index: 0 1 2 3 4 5 6 7 8 9
100 200 300 400 500 600 700 800 900 1000
Total discount = 1500

The algorithm we used to answer this problem use this concept but the other way around. By using modulo to find the reminder of the total amount of items divided by 3 and add this to the pointer set to 0, we are able to get the same set of items. We always add 3 to the pointer to get the next item.

Total Items: 10
Remainder: 10 % 3 = 1
Indices of Items: 0 + 1 = 1
3 + 1 = 4
6 + 1 = 7

As seen above, we were able to get the indices that will yield the most number of discount.

Problem E: DVDs
Status: Accepted
Difficulty Rating: 2
Solution Sketch/Write-Up/Narrative:
In this problem you are given the user is tasked to see how many operations is needed to sort an array of inputs. So the program will start with current of 1 because the the first element should be one (1) . However if the [input != curr] then curr will not not increase. If the [input == current] then curr will increment. This process will occur until the loop conditions are done and complied with and prints the. The approach the group used was subtracting the length of array to the curr (expected) and adding one. The solution used is greedy. Because it goes through the entire array looking for numbers who shouldn't be there.

5 1 2 4 3
Iteration 0 = 1 != 5 ; curr = 1..
1 = 1 = 1 ; curr = 2
2 = 2 = 2 ; curr = 3
3 = 3 !=4 ; curr = 3
4 = 3 = 3 ; curr= 4
Answer: 5 - 4 + 1 = 2 operations

When testing the code the group [1,2,3,4,5] as a test case. The expected output should be 0 since it is already sorted. But the result was -1 or an error, because it [5-6 = -1] , so we adjusted by adding one. So for [5,1,2,4,3] the number 5 and 3 are not in their expected position so the answer is two (2) as seen above/

Problem F: Virus Replication
Status: Run-time Error (probably something to do with the length of the string)
Difficulty Rating: 3
Solution Sketch/Write-Up/Narrative:

<u>The Problem:</u>

The problem statement here says that we are essentially searching for the number of modifications made to a string, given the string after all its modifications have been made. However, the problem that we did not note when trying to solve this problem that would have simplified our decision making process to submit a solution in time was to simply note that the DNA inserted was *consecutive* (read the problem thoroughly!). We assumed that the virus was able to insert DNA at any point, therefore we focused on solving the other problems first. However, given that we know that now, we can establish our tenets to solve;

1. Since the virus inserts DNA at any point and can overwrite the already pre-existing DNA, we need to find the length of that insertion; this could easily be misinterpreted as trying to find the number of differences. We can illustrate why that distinction is important later.

The Solution:

To understand the best way to solve this problem is to simply explain it first. Our algorithm for solving is as follows:

1. Starting from the start of each string, compare the original string to the modified string. As soon as there is a difference, discard everything from the modified and original string from the start to where the difference begins
2. Starting from the end of each string and headed to the start of the string, compare the original string to the modified string. As soon as there is a difference, cut out everything after where we find the difference
3. We then simply count the length of the string that remains, since we found the string that the virus inserted into our DNA (yikes!)

The reason this solution works is because since the insertion from the virus is a single consecutive sequence, and since the DNA string is one dimensional, it becomes quite easy to find where the virus inserted DNA. Since we know that the insertion is consecutive, we simply need to find where the insertion begins and where the insertion ends, which means simply finding the first time the modified string differs from the original. Going from left to right, once we find where the similarities between the original string and the modified string end, we can tell that this is the start of the inserted DNA. Going back the other way, we can find the end of the insertion. Since we can see the modified string as a whole, we then can see the whole of the insertion, and it is a simple matter of counting it. Also, since our code failed on a test case with a Run-Time error, we assume that flaw lies in our implementation in Python.

This solution is an O(n) solution, and the greedy property of this problem is finding the first difference between the two strings.

CODE REFERENCES

1. Dragon of Loowater

```
public static void main (String[] args)
   {
       int MAX = 20000;
       boolean cont = true;
       DragonOfLoowater s = new DragonOfLoowater();
       Scanner keyin = new Scanner (System.in);

       do
       {
           int n = keyin.nextInt();
           int m = keyin.nextInt();
           if (n == 0 && m == 0)
           {
               cont = false;
           }
           else
           {
               int ncm[] = new int[MAX];
               int mcm[] = new int[MAX];

               for (int i = 0; i < n; i++)
               {
                   ncm[i] = keyin.nextInt();
               }

               for (int i = 0; i < m; i++)
               {
                   mcm[i] = keyin.nextInt();
               }

               s.sort(ncm, 0, n-1);
               s.sort(mcm, 0, m-1);

               int np = 0;
               int mp = 0;
```

```
                int coins = 0;

                while (mp < m && np < n)
                {
                    if (mcm[mp] >= ncm[np])
                    {
                        coins += mcm[mp];
                        mp++;
                        np++;
                    }
                    else
                    {
                        mp++;
                    }
                }

                if (np < n || n > m)
                {
                    System.out.println("Loowater is doomed!");
                }
                else
                {
                    System.out.println(coins);
                }
            }
        } while (cont);

    }
```

2. Association of Computing Machinery

```
public static void main (String[] args)
    {
        AssociationForComputing sorter = new AssociationForComputing();
        Scanner keyin = new Scanner(System.in);

        int N = keyin.nextInt();
        int p = keyin.nextInt();
```

```
        int mins[] = new int[N];
        int pt = 0;
        int currSum = 0;
        int result[] = new int[N];
        int k = 0;

        for (int i = 0; i < N; i++)
        {
            mins[i] = keyin.nextInt();
        }

        if (mins[p] < 301)
        {
            result[0] = mins[p];
            mins[p] = -1;
            sorter.sort(mins, 0, N-1);

            currSum += result[0];
            pt += currSum;
            k = 1;
            while(k < N && currSum + mins[k] <= 300)
            {
                result[k] = mins[k];
                currSum += mins[k];
                pt += currSum;
                k++;
            }

        }

        System.out.println(String.format("%d %d", k, pt));
    }
```

3. Shopaholic

```
    public static void main (String[] args)
    {
        int MAX = 200000;
```

```
        Shopaholic c = new Shopaholic();
        Scanner keyin = new Scanner(System.in);
        int n = keyin.nextInt();
        int items[] = new int[MAX];
        for (int i = 0; i < n; i++)
            items[i] = keyin.nextInt();

        c.sort(items, 0, n-1);

        int np = 0;
        long discounts = 0;

        if (n < 3)
        {
            System.out.println("0");
        }
        else
        {
            while (n - np > 2)
            {
                discounts += items[np + (n % 3)];
                np += 3;
            }

            System.out.println(discounts);
        }
    }
```

4. DVD

```
#include<stdio.h>

int dvd(int length)
{
   int i;
   int answer=0;
   int curr = 1; // Starts DVD 1
   /*
      Case = 1
```

```
        length = 5
        5 1 2 4 3
        curr = 1
        Iteration 0 = 1 !=5 ; curr = 1..
                1 = 1 = 1 ; curr = 2
                2 = 2 = 2 ; curr = 3
                3 = 3 !=4 ; curr = 3
                4 = 3 = 3 ; curr= 4
        Answer:  5 - 4 + 1 = 2

    */
    int input;
    for (i = 0; i < length; i++)
    {
        scanf("%d" , &input);
        if (curr == input)
        curr++;
    }
    answer = length - curr + 1;
    return answer;
}

int main()
{
    int cases;
    scanf("%d" ,&cases);
    while(cases--)
    {
        int length;
        scanf("%d" ,&length);
        printf("%d\n", dvd(length));
    }
    return 0;
}
```

5. Virus Replication
Python 3.7

```
original = input()
modified = input()

start = 0
end = 0
```

```
Flag = False
for x in range(0, len(original)):
    if (original[x] != modified[x]):
        start = x
        Flag = True
        break

original = original[start::]
modified = modified[start::]
original = original[::-1]
modified = modified[::-1]


for x in range(0, len(modified)):
    if (original[x] != modified[x]):
        end = x
        break
modified = modified[end::]

if (Flag):
    print(len(modified))
else:
    print(0)
```