MITIT 2024 Round 2 Editorial (Beginner Division)

The MITIT Organizers

January 31, 2024

1 MITIT

Problem Idea: Claire Zhang
Problem Preparation: Claire Zhang
Analysis by: Claire Zhang

If a string S of length N is a good contest name, there are $O(N)$ possible ways to split it into ABB, as we have one degree of freedom to choose the lengths of either A or B. For each possible length $|B|$ of B, we can check if $S[N-|B|+1 \dots N]$ equals $S[N-2|B|+1 \dots N-|B|]$ with a for loop. As we check in $O(N)$ for each of $O(N)$ possible splits, we can determine in $O(N^2)$ time if S is a good contest name. If no checks pass, S cannot be a good contest name, so we print `NO`.

2 Taking an Exam

Problem Idea: Sam Zhang
Problem Preparation: Zixiang Zhou and Sam Zhang
Analysis by: Sam Zhang

2.1 Subtask 1

In this case, we will do every problem on the exam and submit immediately afterwards (the order in which we do the problems doesn't matter since the total time taken is unaffected; also it doesn't make sense to skip problems to get extra time, since the d points gained from the d extra minutes is less than the $d+1$ points from doing the problem). The answer is therefore the sum of point values for all problems, plus the bonus points for the remaining time.

It should be noted that in this subtask, it turns out that the answer always equals $M+N$. This is an instance of a more general observation which will be useful for subtask 3.

2.2 Subtask 2

All the problems are of the same difficulty, so we can just keep doing problems until we don't have time to do more problems; since the problems are all the same it doesn't matter which ones we choose.

2.3 Subtask 3

The key observation is the following:

Lemma 2.1. *Suppose that we do k of the N problems, and submit immediately afterwards. Then our score is $M+k$.*

Proof. Say that the problems we do have difficulties $d_1, \ldots, d_k$. Then we spend $d_1+\cdots+d_k$ minutes doing the problems, earning $d_1+1+d_2+1+\cdots+d_k+1$, or equivalently $k+d_1+\cdots+d_k$ points. Then we get $M-d_1-\cdots-d_k$ bonus points from the extra time, so our total score is $k+d_1+\cdots+d_k+M-d_1-\cdots-d_k = M+k$. □

Since M is a constant, maximizing our score is equivalent to maximizing k, the number of completed problems. This can be done by doing the problems in increasing order of difficulty (by first sorting them), until there is insufficient time to do another problem.

3 Delete One Digit

Problem Idea: Sam Zhang
Problem Preparation: Sam Zhang
Analysis by: Sam Zhang

3.1 Subtask 1

We don't have to delete any digit, since one nontrivial factor of N is simply 2.

3.2 Subtask 2

If N has an even number of 1 digits, then N is a multiple of 11, so we don't need to delete any digit, and we can choose 11 as our nontrivial factor. Otherwise N has an odd number of 1 digits, and we can delete one of them to make N a multiple of 11.

3.3 Subtasks 3 and 4

We can simply go through all possible choices of which digit (if any) to delete, and attempt to find a nontrivial factor for each of them. This will always succeed since the problem is always solvable (a fact that will be proven in the next section).

For subtask 3, it suffices to search for the nontrivial factor directly. For subtask 4, we can use the fact that a composite number x must have a nontrivial factor at most $\sqrt{x}$, so we only need to search for factors up to this value.

3.4 Subtask 5

We will provide an algorithm which will show that the problem always has a valid solution.

If N's digits are either only 1's or only 2's, we can apply the solutions to subtasks 2 and 1, respectively, so now assume N has both 1 and 2 digits.

We will use the fact that a number is a multiple of 3 if and only if the sum of its digits is a multiple of 3. Let S be the sum of digits of N. If S is a multiple of 3, then N is a multiple of 3 and we don't need to delete any digit. Otherwise, S is either one more than a multiple of 3 (in other words, $S \equiv 1 \pmod 3$), or two more than a multiple of 3. In the former case, we delete a 1 digit; in the latter case we delete a 2 digit.

(The proof also shows that the problem always has a solution where the nontrivial factor is at most 11. This gives another algorithm, which is to first loop over the nontrivial factor, and then loop over the choice of digit to delete. But this algorithm is tricky to implement in languages without builtin large-integer support, and its correctness is not obvious.)

4 Collecting Coins

Problem Idea: Sam Zhang
Problem Preparation: Claire Zhang and Zixiang Zhou
Analysis by: Sam Zhang

4.1 Subtask 1

Binary search on the number of coins we need at the start.

Go through the tunnels in order. If we ever enter a tunnel with $r_i > c_i$, we can repeatedly use this tunnel to get as many coins as possible, allowing us to reach the goal. In all other cases it doesn't make sense to turn back; in that case we just simulate going through the tunnels one by one, making sure we don't run out of coins in the process.

4.2 Subtask 2

In this case, the problem is simply finding the shortest path from building 1 to building N with the minimum sum of c_i's, which can be done with a shortest path algorithm such as Dijkstra's.

4.3 Subtask 3

Instead of finding the path with minimum total c_i, in this subtask we want to minimize the maximum c_i of any tunnel we use.

There are two possible algorithms:

- Still use a shortest path algorithm, but replace addition of weights with taking maximums.
- Binary search on the answer, using a union-find data structure to check if buildings 1 and N are connected with a subset of edges.

4.4 Subtask 4

Again we will binary search on the answer. For each building i, we will compute f_i: the maximum possible coins it is possible to hold at building i. We will use the following "Dijkstra-like" algorithm to find f_i:

- Initialize $f_1 = k$, where k is the number of coins held in the beginning (the value we are binary searching on), and set $f_2, \ldots, f_N = -\infty$.

- Each step, take the highest f_i that has not yet been chosen, and update f_j for buildings j adjacent to i by considering whether it is possible to enter each tunnel from building i starting with f_i coins, and if so, how many coins will be left afterwards.

The proof of correctness for this algorithm is similar to that of the usual Dijkstra's algorithm.

4.5 Subtask 5

This subtask is similar to subtask 4, except that there may be edges with $c_i < r_i$. Whenever we reach such an edge in the above algorithm, since that edge may be used an arbitrary number of times to gain an arbitrary number of coins, we can stop the algorithm immediately since we know building N will be reachable.

5 101 Things To Do Before You Graduate

Problem Idea: Ketevan Tsimakuridze
Problem Preparation: Ketevan Tsimakuridze and Richard Qi
Analysis by: Ketevan Tsimakuridze

Let $v_{i,j}$ denote the value of the subsegment $a_l, a_{l+1}...a_r$.

5.1 Subtask 1

For each r $(1 \leq r \leq n)$, let's count number of indices l $(1 \leq l < r)$ with $v_{l,r} = k$.

It should be observed that $\forall i, j$ $1 \leq i \leq j \leq n$ and $i < j - 2$ the condition $v_{i,j} \leq v_{i+1,j}$ is satisfied. Thus, for each index r, the valid indices l form a continuous subsegment.

Iterate over r in increasing order and maintain the subsegment l_1, l_2 of valid l-s.

For $r + 1$, to update l_1 and l_2 we have to find:

1. The rightmost index l^*, such that $a_{r+1} \oplus a_{l^*} < k$ (no subsegment with score k ending at r can contain l^*). We make $l_1 = max(l_1, l^* + 1)$.
2. The rightmost index r_*, such that $a_{r+1} \oplus a_{r^*} = k$ (as all the valid subsegments should contain two values with XOR k). We make $l_2 = min(l_2, r^*)$.

   To find l^*, we can simply iterate over all $l \leq r$ in $O(n)$.

   To find r_*, maintain a map of the value occurrences and look up the value (index) for key $a_{r+1} \oplus k$ in $O(\log A)$ where $A = max(a_1, a_2, .., a_n)$.

   The complexity of the solution is $O(n^2)$.

5.2 Subtask 2

When $k = 0$, for all $i < j$, $v_{i,j} > k$, thus we no longer need to find l^* and only maintain r_*-s.

Therefore, the complexity is $O(n \log A)$

5.3 Subtask 3

To solve this subtask, we should observe that when given three numbers $a \leq b \leq c$, $min(a \oplus b, b \oplus c) \leq c \oplus a$. Thus, when given an subsegment in non-decreasing order, minimum pairwise XOR can always be found within the set of XORs of adjacent elements.

Using this idea, when moving to $r + 1$ to update l^*, we only need to check $a_{r+1} \oplus a_r$.

Thus making the complexity $O(n \log A)$.

5.4 Subtask 4

To find l^*, for $r+1$ we maintain a trie of all the values $a_1, ..ar$ with their corresponding indices.

1. Search for the max index l^* with $a_{r+1} \oplus a_{l^*} < k$ in the trie.
2. Add the value a_{r+1} with index $r+1$ to the trie.

Both of the operations take $O(\log A)$, thus the problem is solved in $O(n \log A)$.

6 Beavers and Revaebs

Problem Idea: Ray Bai and Danny Mittal
Problem Preparation: Danny Mittal and Claire Zhang
Analysis by: Danny Mittal

Define M to be a bound on r_k for all k. We will use M alongside N to state the runtime complexities of the solutions below.

6.0.1 Subtask 1

For a given assignment of point values to problems, we can check whether it satisfies the constraint that only the N^{th} beaver and N^{th} revaeb have the same score by first computing the scores of each beaver and revaeb by looping through the problems (first forwards, then backwards) and maintaining a running sum of their point values, then adding these scores to a set and checking that the number of elements in the set is $2N - 1$. This takes $O(N)$ time.

Now, for each problem there are at most M choices of point values, meaning that the total number of assignments of point values that satisfy the constraints $l_k \leq p_k \leq r_k$ is at most M^N. We can therefore enumerate all such assignments and check for each one whether they satisfy the second constraint in $O(N)$ time. Since the number of assignments is $O(M^N)$, this takes $O(M^N N)$ time overall.

In this subtask, $N, M \leq 8$, meaning that $M^N N \leq 8^9 \approx 134,000,000$, which is fast enough to pass.

6.1 Idea for Other Subtasks

The central idea for the solutions to the other subtasks is that any valid assignment of point values can be obtained uniquely using the following process.

6.1.1 The process

We build the assignment of point values from both ends, meaning that we maintain a list of point values at the beginning and a list of point values at the end, and at each step of the process we add a point value to the end of one of those lists. Specifically, we always add a point value to the list whose current sum of point values is smaller. In order to obey the constraints of the problem, we make sure that the point value we add is in $[l_k, r_k]$ for the appropriate k and also that the two lists never have the same sum (except at the beginning, when they're both 0).

6.1.2 Example of the process

Here is an example with 5 problems. We start with no point values assigned:

```
. . . . .
```

We then choose to add a point value of 3 to the beginning:

```
3 . . . .
```

Now the beginning has a sum of point values equal to 3, and the end has a sum of point values equal to 0, so we need to add a point value to the end. We choose to add a point value of 5:

```
3 . . . 5
```

Now the beginning has the smaller sum, so we choose to add a point value of 1:

```
3 1 . . 5
```

The beginning is still smaller, so we add 2:

```
3 1 2 . 5
```

The end is smaller now, so we add 2 (note that we cannot add 1, because then both ends would be equal):

```
3 1 2 2 5
```

And that is our assignment of point values to problems.

6.1.3 Executing the process is equivalent to a valid assignment

Clearly, any valid assignment can be attained using this process. It is also almost unique, because at every step of the way, since the beginning and end must have different sums, we are forced to add to the lesser one – except at the beginning, where we have two choices; we can fix this by always adding to the beginning at the first step.

We can further prove that any assignment attained using this process is valid. First, such assignments clearly satisfy the constraints $l_k \leq p_k \leq r_k$ for all k.

Then, suppose that during the process we added b point values to the beginning and so $N - b$ to the end. We know that the scores of beavers solving at most b problems are distinct from the scores of revaebs solving at most $N - b$ problems because we guaranteed it during the process.

Now, let s be the sum of all point values in the assignment. Observe that the score of a beaver solving more than b problems is equal to s minus the score of a revaeb solving less than $N - b$ problems, and that the score of a revaeb solving more than $N - b$ problems is

equal to s minus the score of a beaver solving less than b problems. It follows from this that the scores of beavers solving more than b problems are distinct from the scores of revaebs solving more than $N - b$ problems (except for the beaver and revaeb that solved all the problems, since there are no corresponding rodents that solved 0 problems).

Continuing, suppose that the score of the beaver that solved b problems is more than the score of the revaeb that solved $N - b$ problems (otherwise, just switch beavers with revaebs in the following argument). We can conclude that the scores of beavers solving more than b problems are strictly greater than, and therefore distinct from, the scores of revaebs solving at most $N - b$ problems.

However, because in the process we always add a point value to the side with the currently smaller sum, since we added a b^{th} point value to the beginning, it must be that the score of the beaver solving $b - 1$ problems was smaller than the score of a revaeb solving at most $N - b$ problems. This means that the scores of the revaebs solving more than $N - b$ problems are strictly greater than, and therefore distinct from, the scores of revaebs solving less than b problems.

From these four cases we can see that the scores of beavers and revaebs are all distinct other than the two rodents that solved all the problems. We can thus conclude that the valid assignments are exactly those assignments produced by the process, and therefore we can count the valid assignments by counting the number of ways to execute the process.

6.2 Subtask 2

Define M so that $r_k = M$ for all k. In executing the process, at each step we have M possible values for the point value to add, suggesting that the answer is M^N. However, one of those values might make the two lists' sums equal.

To fix this, we need to observe an important property of the process that will also be used in the later subtasks, which is that because we always add the point value to the list with the smaller sum, and the point value we add is always at most M, even if that list now has a greater sum it can only be greater by at most M. Therefore, at all times the list with the greater sum only has the greater sum by at most M.

Thus, for this subtask since we always select the point value to be one of $1, \ldots, M$, we can note that since the difference between the sums of the two lists is always at most M, and other than the first step the difference is always at least one, the difference must be one of $1, \ldots, M$, meaning that there is always be a point value out of $1, \ldots, M$.

This means that at any step of the process other than the first, we have $M - 1$ choices for the point value. Meanwhile at the first step we still have M choices, so the number of ways to execute the process is $M(M - 1)^N$. We can thus solve this subtask by simply

computing $M(M-1)^N$ modulo 10^9+7, which can be done easily in $O(N)$ time or using modular exponentiation in $O(\lg N)$ time.

6.3 Subtask 3

In this subtask we will directly count the number of ways to execute the process using DP. Our DP state will be (a, b, d) where a is the current amount of point values at the beginning, b is the current amount of point values at the end, and d is the difference between sums of the two lists. We have that $a, b \in [0, N]$, and from the observation in subtask 2 we know that $d \in [-M, M]$, so the number of DP states is $O(N^2M)$.

To transition, we just choose what the next point value is. There are at most M choices, and the choice determines the next state, meaning that for each choice we have a constant time transition. The overall transition from each state is thus $O(M)$, making the overall runtime of the DP $O(N^2M^2)$. For this subtask we have $N^2M^2 \leq 50^2100^2 = 25,000,000$, which is fast enough.

6.4 Subtask 4

The full solution to the problem involves optimizing the DP from subtask 3 by using prefix sums. We transition to a given state (a, b, d) from two types of states: states of the form $(a-1, b, d')$ and states of the form $(a, b-1, d')$.

Thus, when computing the DP values of the states of the form (a, b, d) for a given a, b, we can first compute prefix sums for the states of the form $(a-1, b, d')$ and states of the form $(a, b-1, d')$, then use those sums to compute the value for each state of the form (a, b, d) in constant time.

The computation of prefix sums is $O(M)$, which is also the number of states of the form (a, b, d) for a given a, b. Because of this, the runtime complexity is proportional to the number of states, meaning that it is $O(N^2M)$. Under the constraints in the problems statement, $N^2M \leq 50^22000 = 5,000,000$, which is fast enough.