Day 1 Contest Analysis, Division C

March 9, 2019

Filipp Rukhovich

Moscow Workshops ICPC Summer 2019

Problems A,C

These problems were discussed during a lection.

The only thing is that in division C, there were no any guarantees that an initial array is sorted. So, before start calculating the dynamics, it is needed to sort the sequence. It can be done using, for example, std::sort in C++ or its analogue in Java.

B. Calculator

In this problem, we are to get given number n from 1 using operations $x \to x * 2$, $x \to x * 3$, $x \to x + 1$.

Let $dp[i]$, $i = 1, 2, \ldots, n$, is the minimal number of operations needed to get number i from 1 using these operations.

According for the definition, $dp[1] = 0$. For any $i > 1$, we need to do at least one action. In the optimal way of getting i, last action can be one of the following variants:

- $i/2 \to i$, if i is divided by 2. From optimality of getting i and, as a consequence, $i/2$, we see that $dp[i] = dp[i/2] + 1$;
- $i/3 \to i$, if i is divided by 3. Then, $dp[i] = dp[i/3] + 1$;
- $i - 1 \to i$. Then, $dp[i] = dp[i - 1] + 1$.

B. Calculator

It can be easily seen from above that for any i, $dp[i]$ is equal to 1 plus minimum of numbers $dp[i/2]$, $dp[i/3]$, $dp[i-1]$ (more precisely, minimum of correctly defined subset of these numbers). It give us a way to calculate all $dp-s$ in $O(n)$ time.

Also for each $i > 1$, we can also store "argmin" of $dp[i/2]$, $dp[i/3]$, $dp[i-1]$ to array pr. As in problem of LIS, this array gives the possibility to restore optimal sequence of getting n with linear complexity, too.

D. Tortile

In this problem, we are to find maximum possible score for tortile to get going through the rectangular grid with getting score in each cell in assumption that all steps could only one cell right or down.

To solve this, we'll use 2-dimensional dynamics. Let $dp[x][y]$, $1 \leqslant x \leqslant N$, $1 \leqslant y \leqslant M$, be a maximum possible score tortile can get going from cell $(1, 1)$ to cell (x, y).

Let $a[1..N][1..M]$ be an array of scores from input. Then, dp can be calculated by the following formulas:

- $dp[1][1] = A[1][1]$;
- $dp[1][y] = A[1][y] + dp[1][y-1]$, $2 \leqslant y \leqslant M$;
- $dp[x][1] = A[x][1] + dp[x-1][1]$, $2 \leqslant y \leqslant N$;
- $dp[x][y] = A[x][y] + max(dp[x-1][y], dp[x][y-1])$, $2 \leqslant x \leqslant N$, $2 \leqslant y \leqslant M$.

E. Sequentially-multiple sequence

Sequence of numbers $(B_1, B_2, \ldots, B_k)$ is called sequentially-multiple iff B_{i+1} is divided by B_i (in other words, B_{i+1} is a multiple of B_i), $i = 2, 3, \ldots, k$.

Given a sequence $(A_1, A_2, \ldots, A_n)$, it is to find a length of longest **sequentially-multiple** subsequence.

It's NOT a problem about longest *increasing* subsequence :) But nevertheless, the solution can be done on the same way. The only thing we need to change is to replace the condition "$if\,(A[i] < A[j]$ && $dp[i] < dp[j] + 1)$" to $(A[i]\%A[j] == 0$ && $dp[i] < dp[j] + 1)$.

DP: LCS problem

Next problem is the longest common subsequence problem, or LCS:

- given two sequences $A = (A[1], A[2], \ldots, A[n])$ and $B = (B[1], B[2], \ldots, B[m])$, we are to find such two sequences $1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant n$, $1 \leqslant j_1 < j_2 < \ldots < j_k \leqslant m$ for maximal possible k, such that $A[i_1] = B[j_1]$, $A[i_2] = B[j_2]$, ..., $A[i_k] = B[j_k]$.

In this problem, subproblems will be $dp[i][j]$, $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, for $dp[i][j]$ as length of the longest common subsequence of sequences $A[1..i] = (A[1], A[2], \ldots, A[i])$, $B[1..j] = (B[1], B[2], \ldots, B[j])$.

It's obvious that $dp[i][j] = 0$ if $i = 0$ or $j = 0$. Otherwise, there exist two possible cases:

DP: LCS problem

- $A[i] = B[j]$; then, it makes sense to take $A[i] = B[j]$ as last element of common subsequence; then, $dp[i][j] = 1 + dp[i-1][j-1]$;
- $A[i] \neq B[j]$; then, for and optimal $LCS(A[1..i], B[1..j])$, $A[i]$ does not lie into tis LCS or $B[j]$ does not lie into this LCS (these two statements can be true at the same time, but at least one must be true definitely). Then, $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$.

It brings us to the following simple code:

DP: LCS problem

```
int n, m;
int a[1..n], b[1..m], dp[0..n][0..m];

void calculateLCSDynamics() {
    for (int i = 0; i <= n; ++i)
        dp[i][0] = 0;
    for (int j = 0; j <= n; ++j)
        dp[0][j] = 0;

    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            if (a[i] == b[j])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = std::max(dp[i-1][j], dp[i][j-1]);
    }
}
```

The answer for the problem is $O(n^2)$. If it is necessary to restore LCS itself, one can build array $pr[1..n][1..m]$ and store information about one of two cases where the $dp[i][j]$ was found from, and then restore the answer by the same way as for LCS problem. The complexity of the solution is $O(n^2)$.

F: LCS problem

Next problem is the longest common subsequence problem, or LCS:

- given two sequences $A = (A[1], A[2], \ldots, A[n])$ and $B = (B[1], B[2], \ldots, B[m])$, we are to find such two sequences $1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant n$, $1 \leqslant j_1 < j_2 < \ldots < j_k \leqslant m$ for maximal possible k, such that $A[i_1] = B[j_1]$, $A[i_2] = B[j_2]$, ..., $A[i_k] = B[j_k]$.

In this problem, subproblems will be $dp[i][j]$, $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, for $dp[i][j]$ as length of the longest common subsequence of sequences $A[1..i] = (A[1], A[2], \ldots, A[i])$, $B[1..j] = (B[1], B[2], \ldots, B[j])$.

It's obvious that $dp[i][j] = 0$ if $i = 0$ or $j = 0$. Otherwise, there exist two possible cases:

DP: LCS problem

- $A[i] = B[j]$; then, it makes sense to take $A[i] = B[j]$ as last element of common subsequence; then, $dp[i][j] = 1 + dp[i-1][j-1]$;
- $A[i] \neq B[j]$; then, for and optimal $LCS(A[1..i], B[1..j])$, $A[i]$ does not lie into tis LCS or $B[j]$ does not lie into this LCS (these two statements can be true at the same time, but at least one must be true definitely). Then, $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$.

It brings us to the following simple code:

DP: LCS problem

```
int n, m;
int a[1..n], b[1..m], dp[0..n][0..m];

void calculateLCSDynamics() {
    for (int i = 0; i <= n; ++i)
        dp[i][0] = 0;
    for (int j = 0; j <= n; ++j)
        dp[0][j] = 0;

    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            if (a[i] == b[j])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = std::max(dp[i-1][j], dp[i][j-1]);
    }
}
```

The answer for the problem is $O(n^2)$. If it is necessary to restore LCS itself, one can build array $pr[1..n][1..m]$ and store information about one of two cases where the $dp[i][j]$ was found from, and then restore the answer by the same way as for LCS problem. The complexity of the solution is $O(n^2)$.

G. Stones

Given a pile of n stones. Two players play a game. On each move, player can take $x > 0$ stones from the pile, if $x \in S(i)$; here i is current number of remaining stones and $S(i)$ is a set of possible moves. The player who cannot make a move (if there no any stones) loses. The question is: who wins if both players are played optimally?

The structure of $S(i)$ is described in the statement (and in fact, $S(i)$ depends only on $i\%3$).

G. Stones

Let's $dp[i]$, $0 \leqslant i \leqslant n$ is equal to 1 in case that there are i stones in pile initially, first (making first move) player has a winning strategy (i.e. can win independently of strategy of second player), and 2 if the second player has such a strategy.

We will prove by induction that for any $i \in \{0, 1, \ldots, n\}$, for game of pile with i stones there exist a winning strategy for one of players. Also we describe an algorithm of calculating $dp[1..n]$.

For establishing a base of induction, we see that $dp[0] = 2$ because first player cannot move in case $i = 0$.

Then, suppose that we prove the existence of strategy for $i = 0, 1, \ldots, i_0 - 1$. Let's find a $dp[i_0]$. Note that if first player decided to take $x \in dp[i_0]$ stones then the following sutiation is equivalent to the game with $i_0 - x$ stones, in which first (for pile i) player plays as a second player and vice versa.

G. Stones

There are two possible cases:

- there is such $x \in A(i_0)$ that $dp[i_0 - x] = 2$. It means that if first player makes this move then he can win as second player in game with pile $i_0 - x$, so $dp[i_0]$ is proved to be 1;
- for any $x \in A(i_0)$, $dp[i_0 - x]$ is equal to 1. It means that independently of first move of first player, second player has a winning strategy as first player in pile $(i_0 - x)$. Then, $dp[i_0]$ is proved to be 2.

So, the rest of the solution is to write a program which calculates array dp in $O(n)$ time (because size of any $A(i)$ is not more than constant 3).

H. Staircases

In this problem, we are to calculate a number of different staircases which we can build using exactly n bricks.

Let $dp[i][j]$, $0 \leqslant j \leqslant i \leqslant n$, is the number of different staircases consisting of i bricks and has j bricks in the base layer.

If $i = j$ then $dp[i][j] = 1$ (obviously). Otherwise we need to build a smaller staircase of $i - j$ bricks with no more than j bricks in the base layer and stand it into a row of j bricks. It can be done by $\sum_{k=1}^{min(i-j,j)}$ ways.

The answer can be found as $\sum_{j=0}^{n} dp[n][i]$. The complexity of the solution is $O(n^3)$.