EGOI 2024 Editorial - Bouquet

Problem author: Jasmin Studer.

The problem

There are N tulips numbered from 0 to $N-1$ growing in a line along the road, in order from left to right. The tulip protection law assigns two integers l_i and r_i to every tulip. In case tulip i is picked, the l_i tulips immediately to the left of tulip i, and the r_i tulips immediately to the right of tulip i cannot be picked. What is the maximum amount of tulips that can be picked?

Test group 1: uniform tulips

In this test group, there exists a constant c such that $l_i = r_i = c$ for all i, that is all the l's and r's in the input have the same value.

This means that any two picked tulips must be at least $c+1$ apart. Thus, we can pick at most one tulip from every $c+1$ consecutive tulips, and we cannot do better than picking tulips $0, c+1, 2(c+1), \ldots$.

Therefore, the answer is $\lceil \frac{N}{c+1} \rceil$ (rounded up).

Test group 2: no constraints on the right

In this test group $r_i = 0$ for all i.

We can adapt the solution from test group 1 in the following greedy manner: First, we pick tulip $a_0 = 0$. Then, we pick the next tulip a_1 that can be picked according to the constraints, in other words the smallest a_1 that has $l_{a_1} < a_1$. Then, we pick the smallest a_2 such that $l_{a_2} + a_1 < a_2$, and so on.

To prove that this is optimal, we can use the exchange argument: suppose there is an optimal solution that does not pick a certain tulip a_k from the greedy solution above. Consider the next tulip a'_k that is picked by that solution instead, by the construction of a_k we have $a'_k > a_k$. We can then replace a'_k by a_k in that solution and obtain another valid optimal solution. By doing this repeatedly, we can prove that the greedy solution is optimal.

Test group 3: low N

In this test group, we need to solve the general case, but with $N \leq 1000$.

The greedy strategy is no longer optimal, as can be seen from the second sample. However, it is still true that the constraints are applied locally, on pairs of adjacent tulips. Therefore, if we pick tulips from left to right, at every moment only the location of the last picked tulip and the total number of picked tulips matter for the rest of the process.

This naturally leads to the following dynamic programming approach: we compute dp_i as maximum number of tulips that can be picked from tulips $0, 1, \ldots, i$ in such a way that tulip i is definitely picked.

We can use the following formula to compute dp_i in increasing order of i:

$$dp_i = \max(1, \max_{j:j<i, \max(r_j, l_i)<i-j} dp_j + 1)$$

This leads to a $O(N^2)$ solution which is fast enough for this test group.

Test group 4: small l_i, r_i

In this test group $l_i, r_i \leq 2$.

We can notice that any two picked tulips in an optimal solution cannot be 6 or more apart, since in that case we can add another tulip in the middle without violating the constraints, contradicting the fact that the solution is optimal.

Therefore we can only iterate over $j \geq i - 5$ in the above dynamic programming solution, which brings its running time down to $O(N)$.

Test group 5: general case

In this test group we need to solve the general case, and $N \leq 2 \cdot 10^5$.

There are multiple ways to approach this problem, for example by using a segment tree to speed up the $O(N^2)$ dynamic programming solution from test group 3.

Consider the condition that we have on j: $\max(r_j, l_i) < i - j$. Let us split this condition into two: $r_j < i - j$ and $l_i < i - j$, which can be rewritten as $j + r_j < i$ and $j < i - l_i$. If we put the array dp_j into a segment tree, then we can quickly find the maximum of all dp_j where $j < i - l_i$. However, how do we deal with the additional $j + r_j < i$ constraint?

In order to do that, we need to time the updates of the segment tree properly. More precisely, instead of storing the value of dp_j into the segment tree immediately after computing it, we will postpone it until we process $i = j + r_j + 1$. This way when we process the given i our segment tree contains exactly the values we need and we can simply query the maximum over all $j < i - l_i$.

This solution runs in $O(N \log N)$. However, the segment tree was not really necessary to solve this problem.

Instead of storing the values of dp_j in a segment tree, we can store an array b_k: the minimum value of j such that $dp_j \geq k$ for each k. In this case instead of querying the segment tree we can use binary search on this array to find the last value that is less than $i - l_i$, since the array will always be non-decreasing.

When we need to put a new value of dp_j into the array (at the step $i = j + r_j + 1$), it might seem that we need to change b_k for all k from 0 to dp_j. However, since when computing dp_j we have used a previous value from the array dp that is smaller by 1, actually at most one position in the array b_k needs to be changed: we simply need to do $b_{dp_j} = \min(b_{dp_j}, j)$.

This solution also runs in $O(N \log N)$ since it first sorts an array, and then does a binary search in another array N times.