Trampolines

Author : Ayman MOUTEI.
Solution : Akram EL OMRANI.
Preparation : Akram EL OMRANI and MOUTAHID Mohammed Amine.

- For the first subtask, one can just brute-force for each i, move to the next position while incrementing the bounces counter until the end is reached. You will end up with an $\mathcal{O}(n \cdot q)$ solution, which is too slow.

- For the second subtask, we will move from the back and try to create for each index i its answer with DP. Let our $dp[i]$ be a pair of the position you will end up in and the number of bounces if you start from the position i. By looping from the end, if we are at some position j, then all the values $j < i \leq n$ are precomputed. We do some case work on whether the next position is in bounds, i.e., if $j + t[j] \leq n$, then the bounces needed from j are the bounces needed from $j + t[j]$ plus 1, and the final position is exactly that of the final position if one starts from $j + t[j]$. We end up with an $O(n)$ solution if there are no queries.

There are various ways to solve this problem. We will discuss two of them which rely on sqrt decomposition techniques. So before moving any further, you should be familiar with these, and we recommend: Errichto's lecture: `https://codeforces.com/blog/entry/96713` that is paired with a YouTube video explaining them (link in the blog).

- First solution:

You divide the array into some blocks, each of size S. For each index i, you record the bounces needed to move to the next block and its position after reaching the next block. It can be seen that in each position, one only modifies the elements in its block, so $O(S)$ per update. For each query of type 1, one moves along blocks until the end is reached and will pass by a block at most once, so $\mathcal{O}(\frac{n}{S})$ per query. Picking $S = \sqrt{nq}$ yields an $\mathcal{O}(\sqrt{nq})$ solution.

Code: `https://ideone.com/mFiRag`

- Second solution:

It consists of the rebuilding-the-structure technique. The idea is that there aren't many indices that change if you modify a certain position i. This can be viewed as an extension of the subtask 2 solution.

For each query, we maintain a current set of indices that have been updated so far, and for each i in the query of type 1, we ask ourselves the following question: from a position i, if it was updated, we move to the position $i + t[i]$ and increment the bounces counter.

If i was not updated, we check if by moving through the path from i and making some jumps, we can reach a node that has been updated. Let this node be j. Then we move from i to j and increment the bounces counter by $dp[i].second - dp[j].second$, because each node in the path from i to j, excluding j, was not modified, and it is easy to notice that the number of bounces needed to get to j is exactly $dp[i].second - dp[j].second$.

Since the set is sorted, it is enough for us to loop over its elements and check the first one for which it appears in the path from i to the last node in its path before the updates.

Okay, that's great, but how do we efficiently check if a certain index j appears after i in the path? It can be viewed that the paths are in some trees rooted at the last element we bounce from. And to check if a node u is a parent of v, we can keep the times of entrance of u and v, and the time of exit. It is known that one needs to check if $\text{tin}[u] < \text{tin}[v]$ and $\text{tout}[v] < \text{tout}[u]$.

Now, if we ensure that the size of the update set doesn't exceed S, the complexity for each query would be $\mathcal{O}(S \log(n))$. Every $\mathcal{O}(\frac{q}{S})$ queries, we clear our set and rebuild our structure in $\mathcal{O}(n)$.

Picking $S = \sqrt{\frac{qn}{\log(n)}}$ achieves a complexity of $\mathcal{O}(\sqrt{qn \log(n)})$, which is good enough to pass. In practice, you may also fix constants like $S = 50$, and it'll pass comfortably.

Code: `https://ideone.com/mWHur1`

Yet Another Maximization Problem

Author : Akram EL OMRANI.
Preparation : Akram EL OMRANI.

The first subtask can be solved by taking the difference between the last and first element since the array is sorted. It can be seen from this subtask that the subarrays we pick should always be sorted either increasingly or decreasingly. In the first and second subtask, this observation is needed to solve it by keeping two values for the two options:

$$dp[i] = \max_{j<i}(dp[j]+a[i]-a[j], dp[j]+a[j]-a[i]) = \max\left(\left(\max_{j<i}(dp[j]-a[j])\right)+a[i], \left(\max_{j<i}(dp[j]+a[j])\right)-a[i]\right)$$

Such a thing can be implemented easily, and you already have 50 points! Code for subtask 1 + subtask 2: `https://ideone.com/DSVyYF`

Now, with the modifications, we'll need to make further observations. First, we'll introduce B, the difference array for the array a (formally defined as $B_i = a_{i+1} - a_i, \forall i \in [\![1, n-1]\!]$). Why, you might ask? It's because an array a is strictly decreasing (respectively strictly increasing) if and only if $B_i < 0$ (respectively $B_i > 0$) for all i.

Not only that, but the difference between the maximum and the minimum is exactly the absolute sum of the difference array if the arrays are strictly decreasing or increasing (one can see that with some case work).

But for a general array, if we choose the split $[L, R]$ into $[L, K]$ and $[K+1, R]$ such that both are strictly monotonic, then the answer is $\sum_{i=L, i\neq K}^{R} B_i$, since we'd have to skip the difference between K and $K+1$.

It is to be noted that range addition updates are super easy with difference arrays. Think about it: if you add x to a range $[L, R]$, only the differences B_{L-1} and B_R would be modified. Hence, any of our updates can be done easily.

Now that we know this, in a segment tree, we need to keep track of each interval for which a node i spans if we take the left side or right side—4 possible cases in total—and the merging should take that into consideration. Full solution: `https://ideone.com/Hcdojn`