Editorial
RMI 2020

1 Task Floppy

Author: Andrei-Costin Constantinescu
With thanks to Daniel-Ilie Apostol for the description below

1.1 Solution storing $N \lceil logN \rceil$ bits (28 points)

For this solution, we first observe that we can normalize the values of the array. That way all numbers will be in the range [0...N-1], which means we can store the first $\lceil logN \rceil$ bits of every number in the array. Now, we send the normalized array to the second function using $N \lceil logN \rceil$ bits in total.

For the queries we can use a segment tree, or any range query data structure that has $O(logN)$ or $O(\sqrt{N})$ complexity per query.

1.2 Solution storing only $2N$ bits (100 points)

We will consider the Cartesian Tree of the array.

The Cartesian Tree can be defined recursively as a tree where the root is the minimum value of the sequence, and the left and right subtrees are the Cartesian trees for the subsequences to the left and right of the root value. We can create this tree easily just by using a max stack (in linear time).

We know that the Cartesian Tree is unique and only defined for an array of distinct values.

Now, let's see how we can send this structure to the other function using 2N bits, so that we can restore it. Let's run a DFS algorithm from the root of the tree and add (in the order of the tree traversal) **2 bits** for each node (the first one indicating if the node has a **left son** and the second one if it has a **right son**).

We can restore the Cartesian Tree simulating the DFS algorithm again, because we know the adjacency list of every node in the order they were traversed in the previous DFS.

We can also restore the position of every node of the tree in the initial array. For every node in the tree its corresponding position in the initial array is equal to its position in the inorder of the tree traversal.

Let's name the corresponding node for every position $Tag[pos]$, where $0 \leq pos \leq N-1$.

Now, the answer for every query $[L..R]$ is simply $LCA(Tag[L], Tag[R])$.

2 Task Brperm

Author: Tamio Vesa-Nakajima
With thanks to Mihai-Cristian Popescu and Alexandru Petrescu for the description below

2.1 $O(2^K * K)$ per query

For every query, we compute the br-permutation of the given length on the spot, in $O(2^K * K)$ time (inverting the bits of each index one by one), and check the permuted string against the original one with brute-force in $O(2^K)$

2.2 $O(N * log^2(N) + Q)$

Given a generic string x, define:

- $basic\_hash(base, x_0, ..., x_{m-1}) = \sum_{i=0}^{m-1} base^{m-i-1} * x_i$
- R = largest integer such that $2^R \leq N$
- B = an integer denoting a base, which can be SIGMA + 1 (i.e. 27) or 28, or 31, or whatever
- for any integer $0 \leq$ k $\leq$ R, $straight\_hash(x_0..., x_{2k-1}) = basic\_hash(B^{2^{R-k}})$
- $BR\_hash(B, x_0, ..., x_{2^k-1}) = B^{2^{R-k}} * BR\_hash(B, x_0, x_2, ..., x_{2^k-2}) + BR\_hash(x_1, x_3, ..., x_{2^k-1})$

Notice that a string is br-palindrome if its straight_hash is the same as its BR_hash.

Let's precompute $a[k][x] = 2^k$ th ancestor of x, some partial straight hashes (so that we could get the value of a required straight_hash) and $jump[l][k][x] = basic\_hash(B^{2^{R-l}}, ch[x], ch[a[k][x]], ..., ch[a[k][a[k][...(2^l - 1) \text{ times})[a[k][x]]...])$.

For every query, we must only compare the corresponding straight hash with $jump[0][K][x]$.

2.3 $O(N * log(N) + Q)$

For better complexity we can change the way we calculate the BR_hash of a subsequence:

- $BR\_hash(B, x_0, ..., x_{2^k-1}) = B^{2^{R-k}} * BR\_hash(B, x_{2^{k-1}}, x_{2^{k-1}+1}, ..., x_{2^k-1}) + BR\_hash(B, x_0, x_1, ..., x_{2^{k-1}-1})$

For every query, we must only compare the corresponding straight hash with the coresponding BR_hash.

3 Task Peru

Author: Alexandru Petrescu
With thanks to Alexandra-Maria Udristoiu for the description below

3.1 $O(N^2)$

Instead of crushing the beetles with a strength smaller than an integer E in a contiguous subsequence of size K we can choose to crush all the beetles in a contiguous subsequence of size smaller or equal to K. The effort E will be equal with the maximum S_i in the chosen subsequence. We will now use the following dynamic programming: dp_i = minimum cost to crush the first i beetles.

$dp_i = \min(dp_j + \max(S_{j+1}, ..., S_i))$, where $i - j + 1 \leq K$.

3.2 $O(NlogN)$

Another observation is that the effort E of the contiguous subsequence won't change if we add beetles with strength smaller than E. Now, when calculating dp_i we can consider only those positions j for each S_j is maximum of the contiguous subsequence from j to i. Let us denote the array of these positions by P, from left to right. Now, we must consider, as candidates for dp_i the following:

$dp_{P_k} + S_{P_{k+1}}$

This is almost everything: We need to consider the case when position j is precisely $i - K$, i.e. have $P_0 = i - K$ artificially.

We can use a deque to get the array P. In order to find the minimum value of $cost(P_k) = dp_{P_k} + S_{P_{k+1}}$, we can use a min heap.

3.3 $O(N)$

Consider this array P. We know S_{P_k} is decreasing. Over time, array P gets values inserted on the right and gets values popped both on the right and on the left. Querying on array P can be split into two parts. Querying on those values that get popped on the right and querying on those values that get popped on the left.

We can predict whether a certain element of P will be popped on the right (denote by R) or on the left (denote by L): It's enough to know whether S_j is maximum over $S_{[j...j+K-1]}$.

We will keep two data structures, one for R and one for L. We will also simulate the operations on P in order to update these two data structures.

The purpose of the data structures is to query for the minimum of $cost()$ for the positions in place. Because we have divided the task in two, the operations for each subtask are manageable in linear time!

For R we can use a stack which also keeps for R_k the minimum of $cost()$ for the sequence R_0, ... R_k. Each element R_k will be removed from the stack by the next element with bigger strength (when R_k will also no longer belong to P)

For L we can use a deque to find the minimum $cost()$ of the elements which currently belong to L. Because each S_{L_j} is maximum over $S_{[L_j \ldots L_j+K-1]}$, it will be part of P until $i - j > K$, when it will also be popped on the left from the deque. If $cost(L_j)$ is bigger than $cost(L_{j+1})$ we can pop L_j from the deque, because L_{j+1} will belong to P for longer.

4 Task Nicelines

Author: Tamio Vesa-Nakajima, Adrian Panaete

In the following subsections, let V be the maximal coordinate of any point. Furthermore, let

First, consider the problem "given a line $x = x_0$, for some $x_0 \gg V$, find the intersection of the hidden lines with $x = x_0$". How can we solve this problem? There are two solutions that we take into account:

Solution with $N \log V$ queries. Consider the function

$$f(t) = \text{query}\,(x_0, t)$$

Observe that $f(t)$ is the sum of the distances from all the lines to (x_0, t). The distance from one line $y = ax + b$ to (x_0, t) is a piecewise linear function (in t), where the slope of the function changes precisely at the intersection of $y = ax+b$ with $x = x_0$. Furthermore the distance of one line to (x_0, t) is concave (in t). Thus the sum of these distances f is a concave piecewise linear function where the slope changes only at the intersection of the hidden lines with $x = x_0$. Therefore, to solve the subproblem we've asked ourselves, we need only to find the points at which f changes slope. To do this in $O(N \log V)$ queries is simple: just use a divide and conquer solution. To find all of the slope changes of f within an interval (x, y), first check if

$$\frac{f(x) + f(y)}{2} = f\left(\frac{x + y}{2}\right)$$

If so, then there are no slope changes. Otherwise, recursively find all the slope changes in $(x, \frac{x+y}{2})$ and $(\frac{x+y}{2}, y)$.

Solution with $4N + 2$ queries. The solution now is a rather cleverer divide and conquer solution, that operates more on f itself. Note that if we select $x_0 \geq 3V$, then we can prove that the intersections of the hidden lines with $x = x_0$ are within disjoint intervals of length $2V$ – at most one hidden line per interval – and that there are intervals of length V within which no intersection can appear. Translate this to f: it means that the places where f changes slope in certain disjoint intervals of length $2V$ at most once, and within other intervals of length V not at all. Note that since f is a convex piecewise linear function, it can be thought of as the maximum of a set of linear functions. Finding these

linear functions is equivalent to finding f, and lets us find the places where f changes slope (i.e. the intersection points of the linear functions that compose f).

First, use 4 queries to find the leftmost and rightmost linear functions that compose f. These queries are done at certain big/small enough coordinates, so that we can be sure that they find the correct lines. Next, we will use divide and conquer to find the rest. To find all the linear functions composing f with slopes between l_1 and l_2, first find the intersection of l_1 and l_2 – call it t_0. Now we have two cases:

- If t_0 is within an area where no intersection can exist, then we can prove that within that area, a lines different to both l_1 and l_2 exists. Find it with two queries within that area, calling it l_3, and then recursively find all the line segments between l_1 and l_3, and between l_3 and l_1.
- If t_0 is within an area where an intersection can exist, let $t_1 < t_0$ be the largest coordinate within an area without intersections, and let $t_2 > t_0$ be the smallest coordinate within an area without intersections. Check if the line segment $(t_0, f(t_0)), (t_1, f(t_1))$ is equal to the line segment l_2. If not, then similarly to the previous item, find a line l_3 within the area without intersections containing t_1, and continue recursively. If the two points are on l_2, then we can prove that no line segments other than l_2 exist to the right of t_0. Thus check if $(t_0, f(t_0)), (t_2, f(t_2))$ are on l_1. Likewise, if this is not the case, find a line l_3 within the area without intersections containing t_2, and continue recursively. Otherwise, no more line segments exist between l_1 and l_2 and we are done.

This can be prove to work in at most $4N + 2$ queries.

Now that we have solved this subproblem, how can we use it to solve the main problem? There are two ways of doing this:

With two line intersections. Find the intersections of the hidden lines with two vertical lines, at $2V$ and at $-2V$. We can prove that the hidden lines are ordered decreasingly w.r.t. their slope at $-2V$, and increasingly w.r.t. their slope at $2V$. Thus finding the intersections with $x = 2V$ and $x = -2V$ and pairing these up accordingly gives us the lines.

With one line intersection. Find the intersections of the hidden lines with $x = x_0$ for some $x_0 > 2V$. For a line $y = ax + b$, the intersection with $x = x_0$ is $ax_0 + b$. Computing the remainder of this value w.r.t. x_0 gives us b. From there we can deduce a. The condition $x_0 > 2V$ is because $|b| \leq V$, i.e. $-V \leq b \leq V$.

5 Task Sum Zero

Author: Alexandru Petrescu
With thanks to Theodor-Pierre Moroianu for the description below

5.1 Solution in $O(N * Q)$

We process each query independently.
For a query, given L and R denoting the interval we can choose from, we can have a dynamic programming solution, by setting dp_i to be the maximal number of subarrays we can form by only using elements $C_L, C_{L+1} \dots C_i$.

It can be easily verified that the recurrence is the following:

$$dp_i = max(dp_{i-1}, dp_{j-1} + 1, \ where\ C_j + ... + C_i = 0).$$

In other words, either we ignore the i-th element, in which case we come from dp_{i-1}, either i is the last element of a subarray of sum 0.
We can further note that the dp vector is strictly increasing, and so it only makes sense to try to close the smallest subarray with sum 0 (in other words for each i we only consider the biggest j with $C_j + \dots + C_i = 0$.

5.2 Solution in $O((N + Q) * log(N))$ time and $O(N * log(N))$ memory

We define a vector $next$, where

$$next_i = Min(j \mid \exists k, i \leq k \leq j, C_k + \dots + C_j = 0)$$

In other words, $next_i$ represents the smallest j such that the subarray $C_i \dots C_j$ contains a 0-sum subarray.

A greedy way to solve a query (L, R) is to iterate $L, next_L, next_{next_L} \dots$ until we either don't have a valid element or get an element bigger that R. The number of jumps is the maximal number of 0-sum subarrays.

We can therefore make a RMQ-like dynamic programming matrix:

$$jump_{i,j} = 2^j\text{th jump in the } next \text{ vector starting from } i$$

Using this dp we can now answer any query in logarithmic time.

5.3 Solution in $O((N+Q) * log(N))$ time and $O(N)$ memory

The main idea of optimizing the memory usage is to observe that the $next$ vector described above creates a forest (multiple trees), with the father of a vertex i being $next_i$.

Each query can now be translated into "What is the oldest ancestor of vertex L smaller than R? This can be computed with linear memory in multiple ways:

- Binary search on the linearization of the tree
- Computing the heavy-path decomposition of the tree, and finding the ancestor by jumping between paths

Note that most solutions optimising the memory usage fitted into the memory constraints.

6 Task Arboras

Authors: Tamio Vesa-Nakajima, Andrei-Costin Constantinescu, Dan-Constantin Spatarel
With thanks to Luca Perju-Verzotti for the description below

6.1 $O(N * Q)$

Let's first solve this problem without any updates, this can be done using the following dynamic programming: $dp[i][0]$ = the length of the longest chain starting from node i and going down in the subtree rooted in node i, and $dp[i][1]$ = the length of the longest chain going down from node i which doesn't intersect with the first chain.

Let j be the child of node i with the highest $dp[j][0]$, let's denote this child as the *main* child of node i, then:

$dp[i][0] = dp[j][0] + d_j$.

$dp[i][1] = \max(dp[k][0] + d_k)$, where $p_k{=}i$ and $k \neq j$.

The answer is then given by the sum of $dp[i][0] + dp[i][1]$ over all i.

6.2 $O(N + Q * maxHeight)$

We need a way to quickly modify the values of dp after each update. One key observation is the fact that when updating the edge $i - p_i$, the only positions where dp can change are on the path from node 0 to node p_i. Moreover, only a bottom part of this path will change, since if the value of dp hasn't changed in some ancestor of node i, then we can say for sure that it won't change in any of the higher ancestors.

Since updates only increase the value, all nodes where $dp[node][0]$ changes will in the end have their *main* child the one that lies on the path from *node* to i.

With this, when updating an edge, we can just check each ancestor of that edge, starting from the bottom, and update the dp accordingly.

6.3 $O(N + Q * log(N) * log(N))$

We will refer to the edges between a node and it's *main* child as *main* edges and all other edges as *weak* edges.

Let's say that after some update, the value of $dp[i][0]$ increases by some x. If the edge between i and p_i is a *main* one, then we can say for sure, that $dp[p_i][0]$ will also increase by x, and so on until we reach a *weak* edge.

We now have these chains of *main* edges which update by the same value so we can update them quickly using Heavy Path Decomposition and BIT or SegTree, but we still haven't solved the *weak* edges.

Let's now say that after some update, the value of $dp[i][0]$ increases by x and the edge between i and p_i is *weak*, we now have 3 cases:

- nothing changes and we stop,

  when $d_i + dp[i][0] + x \leq dp[p_i][1]$

- only $dp[p_i][1]$ changes and we stop,

  when $d_i + dp[i][0] + x \leq dp[p_i][0]$,

  but $d_i + dp[i][0] + x > dp[p_i][1]$,

  then $dp[p_i][1]$ becomes $d_i + dp[i][0] + x$

- $dp[p_i][0]$ changes, and we continue by changing the chain of *main* edges going up from p_i,

  when $d_i + dp[i][0] + x > dp[p_i][0]$

  then $dp[p_i][1]$ becomes $dp[p_i][0]$

  and $dp[p_i][0]$, the whole chain of *main* edges going up increases by (d_i + $dp[i][0] + x$) - $dp[p_i][0]$

  and the *main* child of p_i becomes i (while also changing the *main* edge)

So we will once again start from the updated edge, and climb up the tree increasing the values of dp, but this time, we will only check the *weak* edges, since the chains of *main* edges can be quickly updated. Finding the *weak* edges efficiently can either be done with a rope data structure, since all the chains of *main* edges can be seen as ropes, or with Heavy Path Decomposition by storing the positions of *weak* edges in a set for each Heavy Path.

The complexity is amortized, since after checking each *weak* edge, it will merge the 2 chains of *main* edges together so next time it won't have to check it again.