2nd Contest on Segment Trees

Mamnoon Siam

Draft September 1, 2021

A Sequential Operations 1

Let $f_{l,r+1}(x) = \texttt{query}(l, r, x)$ ($+1$ is just to introduce the half-open interval notation which we'll also use in segment trees). $f_{l,r}$ is a linear function i.e. $f_{l,r}(x) = ax + b$ for some real numbers a and b; you can show this using simple inductive argument: if $f_{l,r-1}$ is linear, $f_{l,r}$ must also be linear.

You can calculate the coefficients of $f_{l,r}$, that is, a and b, where $f_{l,r}(x) = ax+b$, from a_1 and b_1, the coefficients of $f_{l,m}$, and a_2 and b_2, the coefficients of $f_{m,r}$, for any $l \leqslant m < r$: $f_{l,r}$ is just the composition of those two functions i.e. $f_{l,r}(x) = f_{m,r}(f_{l,m}(x))$.

Obviously, we won't store $f_{l,r}$ for all l and r, but store only the functions for each segment tree node interval instead. That way, we can find the (at most) $2 \log n$ disjoint intervals that comprise the query interval, and find their composition function. To answer a query, we'll just evaluate the given x for the resultant function.

As finding composition of two linear functions can be done in constant time, merging two nodes in segment tree is $O(1)$. This allows us to do updates in the segment tree in $O(\log n)$ time.

B Sequential Operations 2

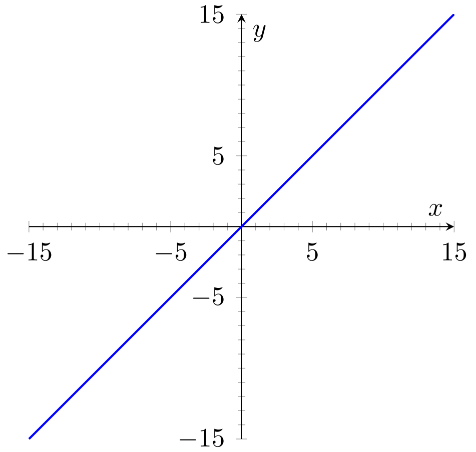
Let's recall the functions from the statement:

$$f_i(x) = \begin{cases} x + a_i & (t_i = 1) \\ \max(x, a_i) & (t_i = 2) \\ \min(x, a_i) & (t_i = 3) \end{cases}.$$

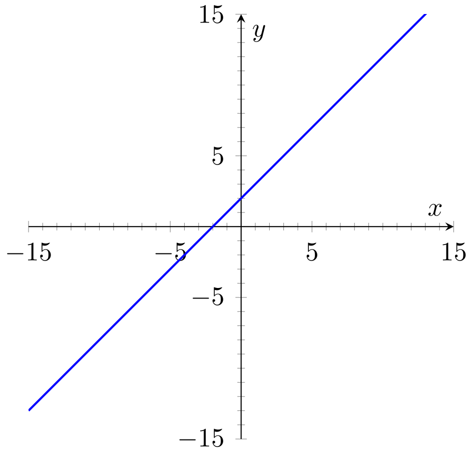
We will denote $f_r(f_{r-1}(\ldots f_{l+1}(f_l(x))\ldots))$ with $f_{l,r+1}$; so, $\texttt{query}(l, r, x) = f_{l,r+1}(x)$.

What do compositions of several functions f_i look like? Let's look at an example:

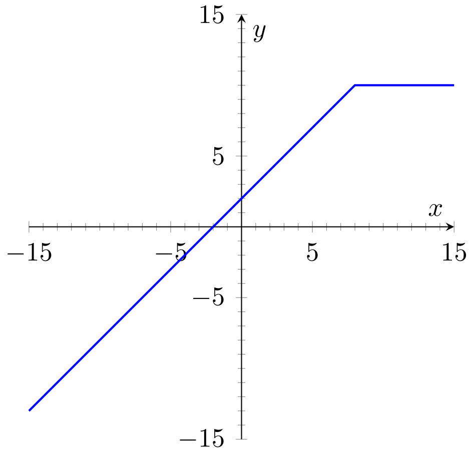
$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$	$f_5(x)$
$x+2$	$\min(x, 10)$	$\max(x, 3)$	$x-5$	$\max(x, 1)$



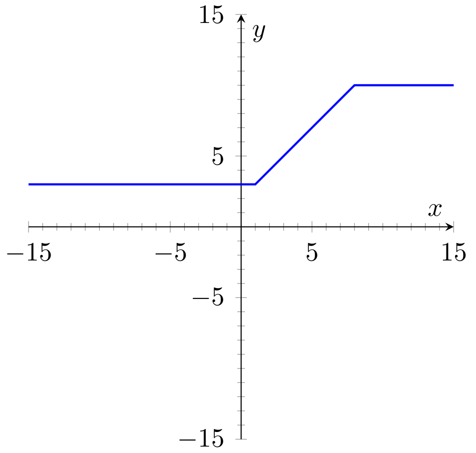
(a) $f_{1,1}(x) = \min(\infty, \max(-\infty, x + 0))$



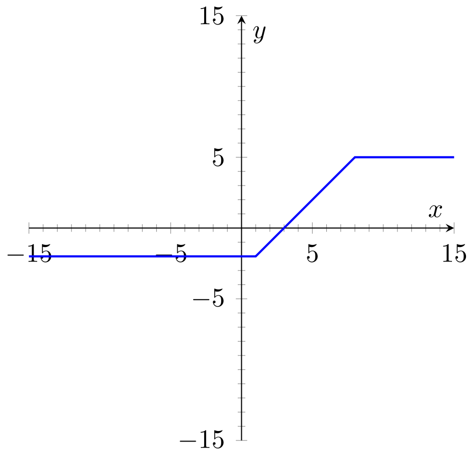
(b) $f_{1,2}(x) = \min(\infty, \max(-\infty, x + 2))$



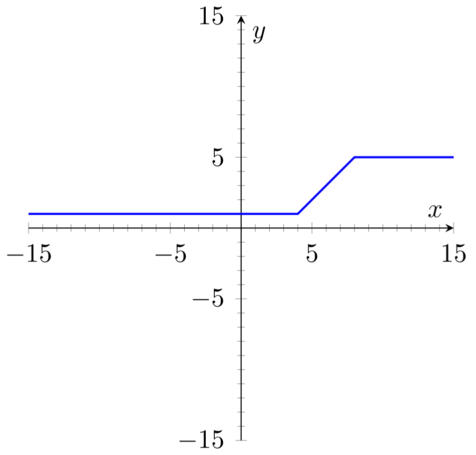
(c) $f_{1,3}(x) = \min(10, \max(-\infty, x + 2))$



(d) $f_{1,4}(x) = \min(10, \max(3, x + 2))$

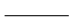
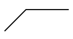
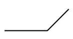
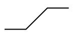


(e) $f_{1,5}(x) = \min(5, \max(-2, x - 2))$



(f) $f_{1,6}(x) = \min(5, \max(1, x - 2))$

Every function is of one of the following types:

1. 
2. 
3. 
4. 

To see why this is true, think about what happens when you apply f_r to $f_{l,r}$ i.e. take $f_r(f_{l,r}(x))$. If $t_r = 1$, the function just translates vertically. If $t_r = 2$, the function gets capped from the bottom, after which $1 \to 1$, $2 \to 1$ or 4, $3 \to 3$, and $4 \to 4$ or 1. $t_r = 3$ case is similar.

Any such function can be described using three numbers a, b, and c: $f(x) = \min(c, \max(b, x + a))$, which, in turn, means that any sequence of operations of any length can be compressing into an equivalent sequence of operations that has constant length (3 at most). How you want to store such functions is up to you – you can store the points where the slope changes, or you can just calculate the values of a, b, and c.

We can think of composing two functions $f(x) = \min(c_1, \max(b_1, x + a_1))$ and $g(x) = \min(c_2, \max(b_2, x + a_2))$, $g(f(x))$, as applying $x + a_2$, $\max(b_2, x)$ and $\min(c_2, x)$ on f successively.

Another way is to find the formula for a_3, b_3, and c_3 explicitly, where $g(f(x)) = \min(c_3, \max(b_3, x + a_3))$:

$$\begin{aligned} a_3 &= a_1 + a_2 \\ b_3 &= \max(b_2, \min(c_1 + a_2, b_1 + a_2)) \\ c_3 &= \min(c_2, \max(b_2, c_1 + a_2)) \end{aligned}$$

We can store $f_{l,r}$ for every node u in the segment tree over the interval $[l, r)$. Merging two nodes can be done in constant time, giving us an update and query complexity of $O(\log n)$.

C Sequential Operations 3

If you distribute the type-2 multiplications over the type-1 summations you will see that

$$\texttt{query}(l, r) = \sum_{\substack{l \leqslant i < j \leqslant r \\ a_i = 1 \\ a_j = 2}} b_i \cdot b_j.$$

For l, r ($1 \leqslant l \leqslant r \leqslant n + 1$), let $f_{l,r}$ be an object with three attributes x, y, and z, where

$$\begin{aligned} x &= \sum_{\substack{l \leqslant i < r \\ a_i = 1}} b_i, \\ y &= \sum_{\substack{l \leqslant i < r \\ a_i = 2}} b_i, \text{ and} \\ z &= \texttt{query}(l, r - 1). \end{aligned}$$

We can find the values of the attributes of $f_{l,r}$ using the values of the attributes of $f_{l,m}$ and $f_{m,r}$, where $l \leqslant m < r$.

Now, just superimpose a segment tree of these objects over the given sequences.

D Sequential Operations 4

Complex Numbers

We will represent 2d points using complex numbers: $P(x, y) \equiv x + yi$.

Properties of complex numbers allow us to describe rotation of a point Q wrt another point Q using only arithmetic operators:

$$\texttt{rotate}(P, Q, \theta) = (Q - P) \cdot \text{cis}(\theta) + P.$$

In our problem $\theta = 90° \cdot t$ for some $t \in \{1, 2, 3\}$. Therefore, $\text{cis}(\theta) = \omega^t$, where ω is the fourth root of unity (that is, $\omega = \text{cis}\left(\frac{\pi}{2}\right) = i$).

Rest of the solution is similar to problem A.

Matrices

Instead of using complex numbers, we can work with vectors and use matrix multiplications to describe rotations. Using matrices is a more general approach because we can then generalize for higher dimensions.

Let's represent points using vectors i.e. $P(x, y) \equiv \vec{P} = \begin{pmatrix} x & y \end{pmatrix}$. If we want to rotate vector $\vec{u}$ θ degrees ccw wrt the origin, then we should multiply $\vec{u}$ by the matrix M_θ, that is, $\vec{u}M_\theta$, where

$$M_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

So,

$$\texttt{rotate}(\vec{P}, \vec{Q}, \theta) = (\vec{Q} - \vec{P})M_\theta + \vec{P}.$$

As matrix multiplication is distributive over matrix addition, this approach also reduces to the operations in problem A.

Remark D.1. It is possible to represent a "rotate wrt another point" operation using only one matrix multiplication. The problem with standard matrix multiplication is that it doesn't allow any translate operation. One way of incorporating rotate + translate operation into one single matrix multiplication is adding one more dimension in the vector space, that is, introducing a new z coordinate in our 2d vectors; this z coordinate helps us copy some extra information while doing matrix multiplication. These are called homogeneous coordinates.

E Divisible Subsequences Queries

In each of the nodes u over the interval $[l, r)$ in the segment tree, store an array $f_{l,r}[\ldots]$, where $f_{l,r}[i]$ is the number of subsequences of $a[l, r)$ with sum modulo k equal to i.

We can calculate the array $f_{l,r}$ from the arrays $f_{l,m}$ and $f_{m,r}$ in $O(k^2)$ complexity. Update and query complexity: $O(k^2 \log n)$.

F Dot Maximizing Subsequence Queries

For an interval $[l, r)$ $(1 \leqslant l < r \leqslant n+1)$, let's define the cost table $f_{l,r}[\ldots][\ldots]$ as follows:

$$f_{l,r}[x][y] = \max_{l \leqslant i_1 < i_2 < \ldots < i_{b-a+1} < r} \sum_{j=x}^{y} a_{i_j} \cdot b_j,$$

where $1 \leqslant x \leqslant y \leqslant k$.

We can calculate the cost table $f_{l,r}$ from the cost tables $f_{l,m}$ and $f_{m,r}$ in $O(k^3)$ time:

$$f_{l,r}[x][y] = \max(\underbrace{f_{l,m}[x][y]}_{\substack{\text{either comes}\\\text{completely from}\\\text{the left side}}}, \underbrace{f_{m,r}[x][y]}_{\substack{\text{or completely from}\\\text{the right side}}}, \underbrace{\max_{z=x}^{y-1} f_{l,m}[x][z] + f_{m,r}[z+1][y]}_{\substack{\text{or partially from}\\\text{the left side}\\\text{and partially from}\\\text{the right side}}}).$$

By merging cost tables of segment tree intervals we can both answer queries and do updates in $O(k^3 \log n)$ complexity.

G Max Cost Subarray Split

The straightforward way to find the difference between the maximum and the minimum of an array would be

$$\max_{i=1}^{n} a_i - \min_{i=1}^{n} a_i.$$

A bit more unconventional approach would be to evaluate the following expression:

$$\max_{1 \leqslant i,j \leqslant n} a_i - a_j.$$

If we describe the *gap* of an array using the later expression, our task will then be calculating the following expression:

$$\underbrace{\max_{l \leqslant i_1 < i_2 < \ldots < i_{k-1} < i_k = r}}_{\text{fix the split}} \underbrace{\sum_{j=1}^{k} \underbrace{\max_{l \leqslant x_j, y_j \leqslant i_j} a_{x_j} - a_{y_j}}_{\substack{\text{gap of the}\\ j\text{-th block}}}}_{\text{sum of the gaps}}.$$

Since our expression is of the form $\max \sum \max$, the problem can be reformulated as follows:

Choose a sequence of $2k\,(1 \leqslant k \leqslant r-l+1)$ indices, $i_1, i_2, \dots, i_{2k-1}, i_{2k}$ and a sequence of $2k$ integers $s_1, s_2, \dots, s_{2k}$, which satisfy the following conditions:

1. $i_{2j-1} \leqslant i_{2j}$, $\forall 1 \leqslant j \leqslant k$,
2. $i_{2j} < i_{2j+1}$, $\forall 1 \leqslant j < k$,
3. $l \leqslant i_1$, $i_{2k} \leqslant r$,
4. $s_i \in \{-1, 1\}$, and
5. $s_{2k-1} \cdot s_{2k} = -1$; think of it as a paired plus-minus..

Find the maximum value of

$$\sum_{j=1}^{2k} a_{i_j} \cdot s_j \tag{1}$$

This looks somewhat similar to the previous problems. In this problem, we will calculate the maximum values of some portions of the expression (1). Therefore, for some interval $[l, r)\,(1 \leqslant l < r \leqslant n+1)$, we'll define the cost table $f_{l,r}[\dots][\dots]$ as follows:

$f_{l,r}[x][y]$ x and y can be regarded as objects $\in \{-1, \varnothing, 1\}$, which means that the portion of (1) we are trying to calculate the maximum value of, starts with x, then forms some paired plus-minuses, and finally ends with y, and is completely from the interval $[l, r)$; this entry of the table stores that maximum possible value. The object $\varnothing$ means that there's no unpaired plus/minus at the end/beginning.

While combining two portions, they must be compatible (the left portion is $x \dots y$ and the right portion is $y' \dots z$):

y	y'
-1	1
1	-1
$\varnothing$	$\varnothing$

We can calculate $f_{l,r}$ from $f_{l,m}$ and $f_{m,r}$ in $O(3^3)$ (well, constant time…), achieving an update and query complexity of $O(\log n)$ (with a constant factor of 27).

H Shortest Path Queries in Grid

For an interval $[l, r)\,(1 \leqslant l < r \leqslant n+1)$, let's define the cost matrix $f_{l,r}$ as follows: $f_{l,r}[x][y]$ is the length of the shortest path from (x, l) to $(y, r-1)$. We can calculate $f_{l,r}$ from $f_{l,m}$ and $f_{m,r}$ in $O(n^3)$ complexity using the following formula:

$$f_{l,r}[x][y] = \min_{\substack{1 \leqslant z,w \leqslant n \\ |z-w| \leqslant 1}} f_{l,m}[x][z] + f_{m,r}[w][y].$$

Maintain these matrices in each node of the segment tree.

I Range Connected Component Queries

We will consider the graph where the vertex set is the cells of the grid and two vertices have edge between them if their corresponding cells are adjacent and are of the same color.

For an interval $[l, r)$ $(1 \leqslant l < r \leqslant n+1)$, let's define few the following things:

$\mathcal{V}_{l,r}$	the set of vertices in the l-th and the $(r-1)$-th column
$\mathcal{C}_{l,r}$	connectivity information of the vertices from $\mathcal{V}_{l,r}$; this can be anything you want – a tiny dsu on $2n$ vertices, a list of groups of vertices from the same connected component etc.
$f_{l,r}$	the number of shaded connected components in the subgrid $[1, n] \times (l, r-1)$ – those that lie strictly inside the interval $[l, r)$ and are completely disconnected from the vertices $\in \mathcal{V}_{l,r}$.

According to the definitions, the answer to the query with segment $[l, r)$ will be:

$$f_{l,r} + (\text{the number of connected components in } \mathcal{C}_{l,r}).$$

The neat thing is that we can calculate $\mathcal{C}_{l,r}$ and $f_{l,r}$ from $\mathcal{C}_{l,m}$, $f_{l,m}$, $\mathcal{C}_{m,r}$, and $f_{m,r}$ in $O(n)$ or $O(n\,\alpha(n))$ complexity. Finally, we just need to superimpose a segment tree of these objects over the columns of the grid.

J OR Subsegment Queries

Let $U = \max a_i$.

First observation: if we take the prefix OR array of an array, it will have at most $\log_2 U$ different integers. Why? Because, each time the prefix OR changes, the new OR will have at least one more set bit than the old one, and the number of set bits cannot increase more than $\log_2 U$ times.

Now, for an interval $[l, r)$, let's define the following:

$f_{l,r}$	the value of $\texttt{count}(l, r-1)$
$p_{l,r}$	a sorted list of pairs containing the integers that occur in the prefix OR array of $a[l \ldots r)$ and their corresponding occurrences – sorted by the values; this list will have at most $\log U$ pairs.
$s_{l,r}$	a sorted list of pairs containing the integers that occur in the suffix OR array of $a[l \ldots r)$ and their corresponding occurrences – sorted by the values; this list will also have at most $\log U$ pairs.

We can calculate $p_{l,r}$ from $p_{l,m}$ and $p_{m,r}$, $s_{l,r}$ from $s_{l,m}$ and $s_{m,r}$, and $f_{l,r}$ from $f_{l,m}$, $f_{m,r}$, $s_{l,m}$, $p_{m,r}$ in $O(\log U)$ complexity:

$$f_{l,r} = f_{l,m} + f_{m,r} + \begin{pmatrix}\text{number of good subarrays starting} \\ \text{at } [l, m) \text{ and ending at } [m, r)\end{pmatrix}.$$

The rightmost term can be computed in $O(|s_{l,m}| + |p_{m,r}|)$ complexity using two pointers over $s_{l,m}$ and $p_{m,r}$.

As we can merge two adjacent intervals in $O(\log U)$ time, we can achieve the update and query complexity of $O(\log n \log U)$.

K Circular Recurrence Queries

Let $1 = c_1, c_2, \ldots, c_k = n + 1$ be the indices of the endpoints. By linearity of expectation, the answer is

$$\sum_{i=1}^{k-1} E(c_i, c_{i+1}),$$

where $E(l, r)$ is the expected number of days needed for Creatnx to go from l to r given that l is the only checkpoint.

If we can somehow manage to implement $E(l, r)$, we can maintain the sum dynamically.

First, let's see how we can calculate $E(1, n+1)$: let e_i be the expected number of days needed to go to $n + 1$, if Creatnx is currently at cell i. We have $e_{n+1} = 0$ and

$$e_i = p_i e_{i+1} + (1 - p_i)e_1$$

for every $i \in [1, n]$. Unfortunately, there are cycles in the recurrence relations; however, we can do the following: we can write every e_i in terms of only e_1 – $e_i = xe_1 + y$ for some x, y. We can find the expressions of all e_i in decreasing order of i, and finally get $e_1 = xe_1 + y$ from which we can determine the value of e_1.

We can generalize this trick further: for two cells l, r ($1 \leqslant l < r \leqslant n+1$), let's define $f_{l,r}(x, y)$ as the expected number of days needed to reach some destination d (note that, it's not necessarily r) if:

- Creatnx is currently at cell l,
- the expected number of days needed to go from r to d is y, and
- and the expected number of days needed to go from the rightmost checkpoint $\leqslant l$ to d is x.

$f_{l,r}(x, y)$ is a polynomial in x and y of degree 1 i.e. $f_{l,r}(x, y) = ax + by + c$ for some $a, b, c \in \mathbb{R}$.

To evaluate $E(l, r)$, we should solve the equation $x = f_{l,r}(x, 0)$.

We can use the following formula to calculate the coefficients of $f_{l,r}$ in $O(1)$:

$$f_{l,r}(x, y) = f_{l,m}(x, f_{m,r}(x, y)).$$

Thus, we can find $f_{l,r}$ for any l, r in $O(\log n)$ using segment tree.

L Majority Queries

L.1 2-Majority Queries

We'll use Boyer-Moore majority vote algorithm. This algorithm finds the majority element from a given list in linear time and constant space. The algorithm is mainly based on the following observation: if you see two different elements in the list, take them and dump them; the majority of the list doesn't change even after dumping those elements. We keep doing this eliminating as long as there's more than one distinct element. After that, if there's any element left, that *may* be a majority; we have to check that one element. However, if there's no element left, it's guaranteed that there's no majority. We can think of it as "the majority will always survive the elimination". The process is associative, meaning that if we were to split the list into two, run the process in both of them separately, combine them, and finally run the elimination process again, we wouldn't lose the majority information.

Instead of choosing pairs arbitrarily, the Boyer-Moore algorithm does so in a quite neat manner (from wikipedia):

- Initialize an element m and a counter i with $i = 0$
- For each element x of the input sequence:
  - If $i = 0$, then assign $m = x$ and $i = 1$
  - else if $m = x$, then assign $i = i + 1$
  - else assign $i = i - 1$
- Return m

The advantage of this algorithm is that all information of the process can be described using only two integers.

In each node u over the interval $[l, r)$ in the segment tree, we'll store the value of m and i that we'd get by running the algorithm on $a[l, r)$. As we can merge the (m, i) pairs of two intervals in $O(1)$, we can find the candidate element of any range in $O(\log n)$ time.

L.2 K-Majority Queries

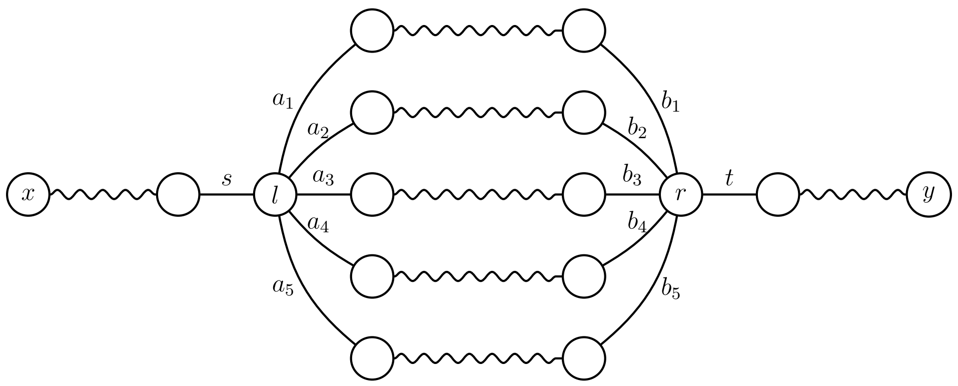
This is a generalization of the previous algorithm. Look up generalization of Boyer-Moore's majority vote algorithm or see G of this for details.

M Shortest Path Queries in Path Graph

Let's introduce some notations:

uv	a path from u to v
$P(x \dots y)$	a path whose first and last edge is of color x and y resp.
$\varnothing$	a sentinel color which is different from all other colors in the input
$\delta_{u,v}(a, b)$	the shortest path $P(x \dots y)$ from u to v such that $x \neq a$ and $y \neq b$; in particular, $\delta_{u,v}(\varnothing, \varnothing)$ is the plain shortest path from u to v.

Let's say that we are trying to find the shortest path between x and y, which have two nodes l and r between them ($x \leqslant l < r \leqslant y$), by concatenating the paths from x to l, l to r, and r to y. To do that, we need to select some lr-subpath. Which one should we choose? Should we try all of them? Definitely not, but it feels like we'd be *almost* correct if we had chosen the shortest path from l to r i.e. $\delta_{l,r}(\varnothing, \varnothing) = P_1(a_1 \dots b_1)$. Almost; we'd get into trouble if, in the xy-shortest path, the color (denote it with s) of the last edge of the xl-subpath is equal to a_1, or the color (denote it with t) of the first edge of the ry-subpath is equal to b_1. So, it's clear that we can't just get away with using only the trivial shortest path. What paths should we go through then? Let's draw a diagram:



It turns out that the following paths are sufficient:

$$\begin{aligned}
P_1(a_1 \dots b_1) &= \delta_{l,r}(\varnothing, \varnothing) \\
P_2(a_2 \dots b_2) &= \delta_{l,r}(a_1, \varnothing) \\
P_3(a_3 \dots b_3) &= \delta_{l,r}(a_1, b_2) \\
P_4(a_4 \dots b_4) &= \delta_{l,r}(\varnothing, b_1) \\
P_5(a_5 \dots b_5) &= \delta_{l,r}(a_4, b_1)
\end{aligned}$$

To understand why, go through the tree shown in the next page *in-order*.

Therefore, for two nodes l, r ($1 \leqslant l < r \leqslant n$), we should maintain only the "top 5" shortest paths. Let's denote this list with $\mathcal{T}_{l,r}$ (because T from Top 5, and it looks like the $\mathcal{T}$ is giving you a high-five). In order to calculate the list $\mathcal{T}_{l,r}$, we should *cross* multiply the lists $\mathcal{T}_{l,m}$ and $\mathcal{T}_{m,r}$, try (two paths can be combined if the color of the last edge of the first one is different than that of the second one) to combine two paths, and then take the top 5 paths from the crossed list. Although, crossing two lists would take only $O(5^2)$ time, extracting the top 5 paths from the crossed list would take $O(5|\mathcal{T}_{l,m}||\mathcal{T}_{m,r}|)$ time, because every call to $\delta_{*,*}(a, b)$ has to iterate over the whole list – thus, a fat complexity of $O(5^3)$ (cough $O(1)$ cough) to merge two intervals.

As we can merge two intervals in $O(1)$ time, we can answer queries and update the segment tree in $O(\log n)$ (with a constant factor of 125, yikes).

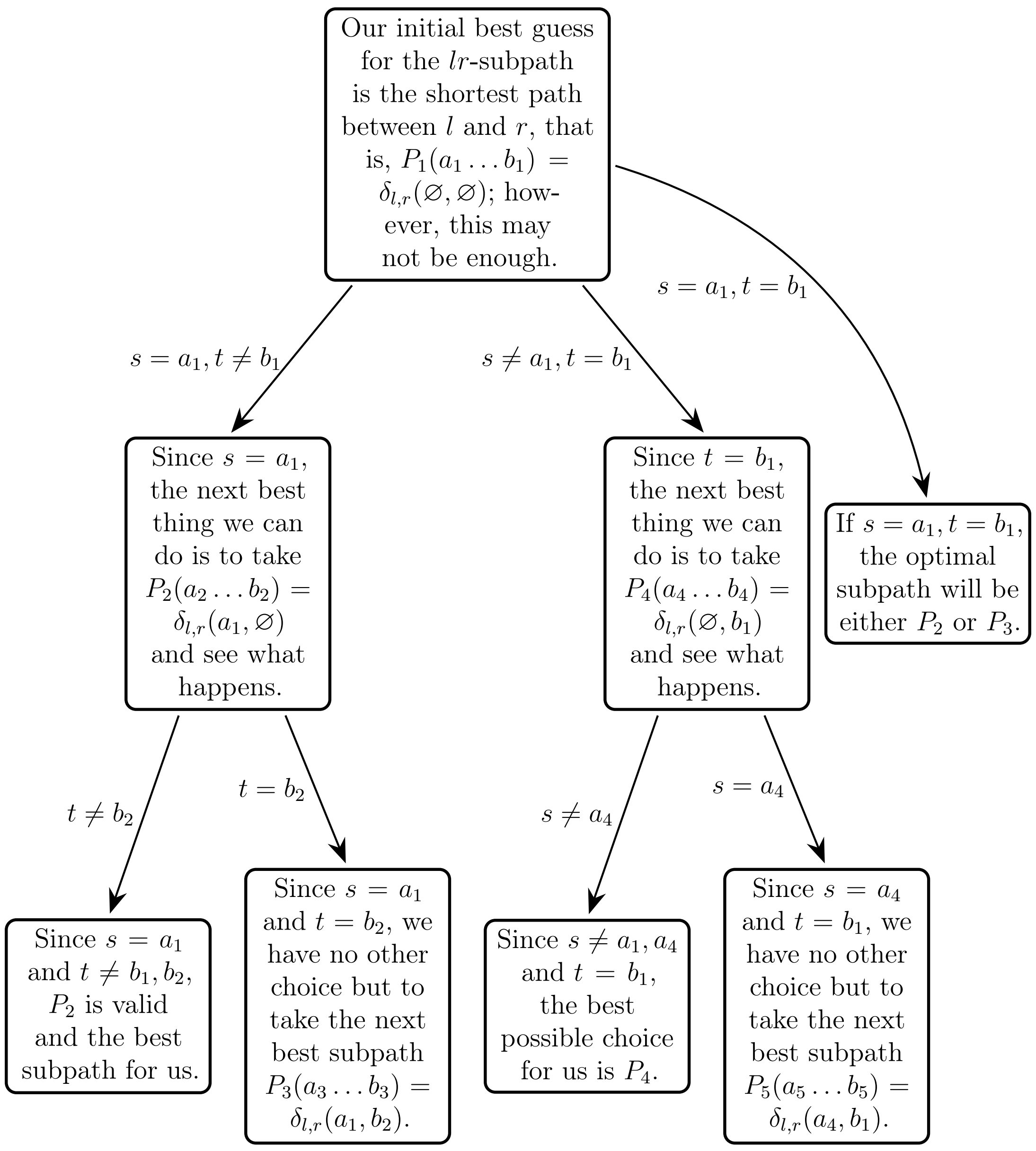
N Wild Boar

This is essentially the previous problem with the extra subproblem of running an all pair shortest path algorithm.

O Bitaro, Who Leaps through Time

Let's assume that $A_j < C_j$; the other direction can be solved by applying the same algorithm in the reversed array.

Furthermore, let's do $R_i := R_i - 1$ for all $i \in [1, n)$. Now, to pass through the i-th road, we must leave the city i at time x satisfying $L_i \leqslant x \leqslant R_i$.



Our initial best guess for the lr-subpath is the shortest path between l and r, that is, $P_1(a_1 \dots b_1) = \delta_{l,r}(\varnothing, \varnothing)$; however, this may not be enough.

Since $s = a_1$, the next best thing we can do is to take $P_2(a_2 \dots b_2) = \delta_{l,r}(a_1, \varnothing)$ and see what happens.

Since $t = b_1$, the next best thing we can do is to take $P_4(a_4 \dots b_4) = \delta_{l,r}(\varnothing, b_1)$ and see what happens.

If $s = a_1, t = b_1$, the optimal subpath will be either P_2 or P_3.

Since $s = a_1$ and $t \neq b_1, b_2$, P_2 is valid and the best subpath for us.

Since $s = a_1$ and $t = b_2$, we have no other choice but to take the next best subpath $P_3(a_3 \dots b_3) = \delta_{l,r}(a_1, b_2)$.

Since $s \neq a_1, a_4$ and $t = b_1$, the best possible choice for us is P_4.

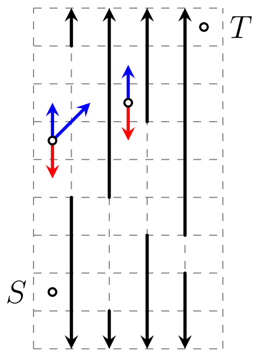
Since $s = a_4$ and $t = b_1$, we have no other choice but to take the next best subpath $P_5(a_5 \dots b_5) = \delta_{l,r}(a_4, b_1)$.

Bitaro's journey and the restrictions on the edges can be visualized in the following way:

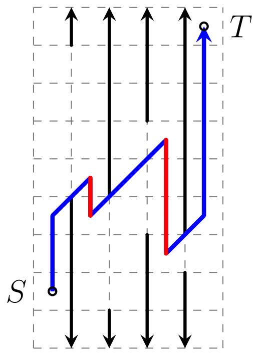
In the 2d plane, the x-axis is Bitaro's position and the y-axis is the current day.
For each i, there is an upward facing ray starting at the point (i, R_i) and a downward facing ray starting at the point (i, L_i).
A query (A_j, B_j, C_j, D_j) describing Bitaro's journey, starting at the city A_j on day B_j and ending at the city C_j on day D_j, means that Bitaro will start from point $S(A_j - \frac{1}{2}, B_j - \frac{1}{2})$ and end up at point $T(C_j - \frac{1}{2}, D_j - \frac{1}{2})$.
When Bitaro is at point (x, y), allowed moves for him are:

1	$(x+1, y+1)$	no cost
2	$(x, y+1)$	no cost
3	$(x, y-1)$	costs 1 coin

Of course, Bitaro can only make a move if he doesn't *cross* over any of the rays (i.e. while moving from one point to another, the line segment joining them cannot intersect any ray internally; however, it is allowed for the line segment to touch rays at their starting point).
Bitaro wants to go from point S to point T using minimum number of coins.



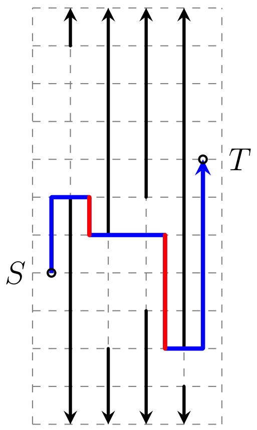
The following greedy algorithm works: whenever possible, he should apply the 1st move. Otherwise, if he's about to hit an upward facing ray, he should go down (move 3), and if he's about to hit a downward facing ray, go he should go up (move 2).



Bitaro has to pay for the red lines.

Diagonal lines are hard to comprehend. Let's make make them horizontal: do $L_i := L_i - i$, $R_i := R_i - i$, $B_j := B_j - A_j$, and $D_j := D_j - C_j$. The new moves are:

1	$(x+1, y)$	go right, has no cost
2	$(x, y+1)$	go up, has no cost
3	$(x, y-1)$	go down, costs 1 coin



In other words, we have applied the linear transformation $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$ on the whole system.

Let's forget about D_j for now; we'll add the extra cost at the end. For some l, r, x $(1 \leqslant l < r \leqslant n)$, we'll try to find the sum of red lines if we start from city l on day x and finish at city r on some day (we wish to find this day, y, as well). It can be done using the following algorithm:

```
Function Run(l, r, x):
    $s \leftarrow 0$;
    $y \leftarrow x$;
    for $i \leftarrow l$ to $r - 1$ do
        $s \leftarrow s + \max(0, y - R_i)$;
        $y \leftarrow \min(y, R_i)$;
        $y \leftarrow \max(y, L_i)$;
    return $y, s$;
```

y-value of $\texttt{Run}(l, r, x)$ is the day on which Bitaro will arrive at city r, and the s-value is the sum of the red lines.

For two cities l and r $(l \leqslant r)$, let's define the following two functions:

$f_{l,r}(x)$	the y-value of $\texttt{Run}(l, r, x)$
$g_{l,r}(x)$	the s-value of $\texttt{Run}(l, r, x)$

For any m $(l < m < r)$, the following equations are satisfied:

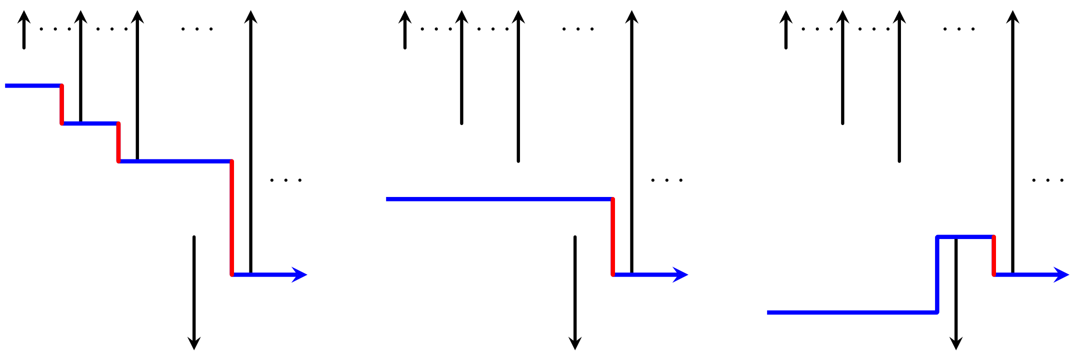
$$f_{l,r}(x) = f_{m,r}(f_{l,m}(x)) \tag{2}$$

$$g_{l,r}(x) = g_{l,m}(x) + g_{m,r}(f_{l,m}(x)) \tag{3}$$

It is easy to see that $f_{l,r}$ will be either a straight horizontal line or piecewise linear function that looks like _/‾.

Claim O.1. $g_{l,r}(x)$ looks like __/.

To verify this claim, let's look at some diagrams:



Dots mean that objects from those regions don't matter.

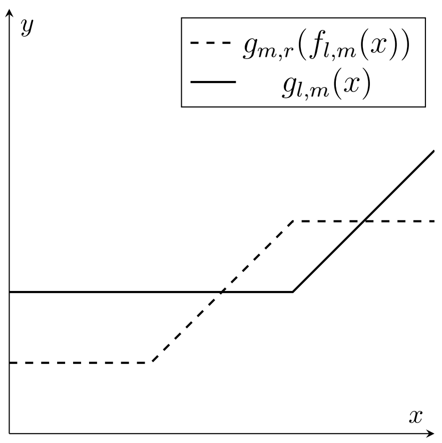
We will shoot the initial rightward ray from the left at an altitude of x (as in the argument of $g_{l,r}(x)$). We'll gradually decrease the altitude. Initially, first obstacle for the ray will be an upward facing ray. As long as the first obstacle is an upward facing ray, $g_{l,r}(x)$ will continue to decrease as we decrease x.

At some point, while decreasing the altitude, the rightward ray's first obstacle will be a downward facing ray. From that point on, value of $g_{l,r}(x)$ won't depend on x i.e. it will remain constant. This supports our claim.

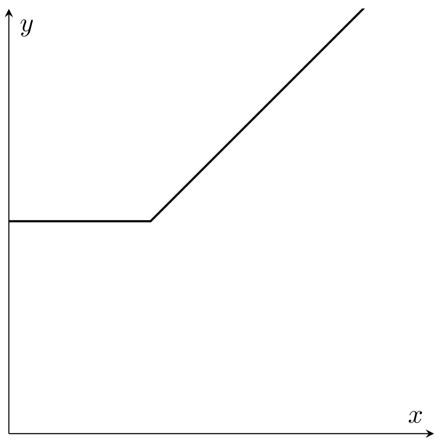
How do we find $f_{l,r}$ and $g_{l,r}$ from $f_{l,m}$, $g_{l,m}$, $f_{m,r}$, and $g_{m,r}$? $f_{l,r}$ is pretty simple; However, $g_{l,r}$ is a bit tricky. We have two cases:

Case 1 $f_{l,m}$ is an horizontal straight line. In such case, the rightmost term of (3) is a constant (and can be evaluated easily), which means that $g_{l,r}$ is just $g_{l,m}$ shifted upward.

Case 2 $f_{l,m}$ looks like _/‾. Since $g_{m,r}(x) =$ __/, the graph of $g_{m,r}(f_{l,m}(x))$ will also look like _/‾. Although $g_{l,m} + g_{r,m}(f_{l,m}(x)) =$ (__/) + (_/‾) isn't necessarily equal to __/, there are some special properties in this problem that we can use to make things a bit tidier. Notice that, for $f_{l,m}$ to look like _/‾, $\min_{i=l}^{m-1} R_i$ has to be larger than $\max_{i=l}^{m-1} L_i$, which, in turn, means that the horizontal piece of $f_{l,m}$ is $y = 0$, and the diagonal piece of $f_{l,m}$, $y = x - c$ (for some positive constant c), starts at $x = \min_{i=l}^{m-1} R_i$. This, combining with the fact that the later horizontal piece of $g_{r,m}(f_{l,m}(x))$ starts at $x = \min_{i=1}^{m-1} R_i$, means there will be no horizontal piece in between two diagonal pieces in the graph of $g_{l,r}$; $g_{l,r}$ is indeed a __/



(a) $g_{m,r}(f_{l,m}(x))$ and $g_{l,m}(x)$

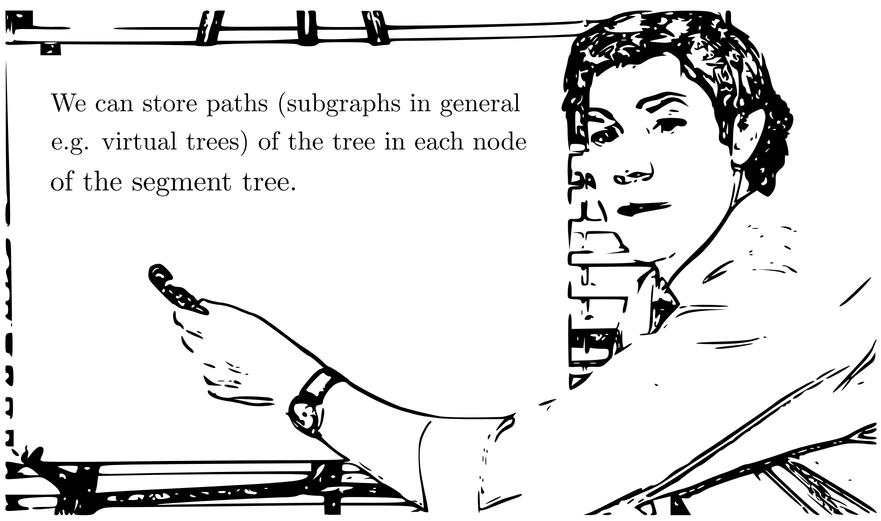


(b) $g_{l,r}(x) = g_{m,r}(f_{l,m}(x)) + g_{l,m}(x)$

Exact formulas to calculate the functions are left as an exercise.

As we can merge two intervals in constant time, updates and queries can be done in $O(\log N)$ time.

P Max Mex



We can store paths (subgraphs in general e.g. virtual trees) of the tree in each node of the segment tree.