JAG Contest Editorial

November 8th, 2016

by Maxim Akhmedov (MSU)

Moscow ACM ICPC Workshop, MIPT, 2016

A. Best Matched Pair

Given $n \leqslant 1000$ positive integers no larger than 10^4, find two of them, producing maximum *valid* product, or determine that there are no such pairs.

A number is *valid* if its decimal representation is a sequence of consecutive increasing digits (like 2, 23, 56789).

A. Best Matched Pair

Constraints are low enough to make the naive solution pass.

A. Best Matched Pair

Constraints are low enough to make the naive solution pass.

Check the product of each possible pair among all $\frac{n \cdot (n-1)}{2}$ pairs of integers.

A. Best Matched Pair

Constraints are low enough to make the naive solution pass.

Check the product of each possible pair among all $\frac{n \cdot (n-1)}{2}$ pairs of integers.

Product consists of no more than 10 digits, so such a solution makes about $\frac{1000^2}{2} \cdot 10 = 5 \cdot 10^6$ operations.

B. Help the Princess!

A princess and several soldiers are located inside the $W \times H$ rectangular maze. Some cells are blocked, exactly one cell contains the exit. In one turn each of the soldiers and princess may move to the cell adjacent by side or stay at the same cell. Find out if the princess may reach the exit without being caught by a soldier.

B. Help the Princess!

Consider the following modified version of the game. Allow soldier catch the princess **only** at the exit cell, i. e. arriving to the same cell with the princess doesn't exit the game unless this cell contains the exit.

B. Help the Princess!

Consider the following modified version of the game. Allow soldier catch the princess **only** at the exit cell, i. e. arriving to the same cell with the princess doesn't exit the game unless this cell contains the exit.

Let's show that this modification doesn't change the outcome of the game.

B. Help the Princess!

Consider the following modified version of the game. Allow soldier catch the princess **only** at the exit cell, i. e. arriving to the same cell with the princess doesn't exit the game unless this cell contains the exit.

Let's show that this modification doesn't change the outcome of the game.

- If the princess could've escaped, she may act as before and still escape.

B. Help the Princess!

Consider the following modified version of the game. Allow soldier catch the princess **only** at the exit cell, i. e. arriving to the same cell with the princess doesn't exit the game unless this cell contains the exit.

Let's show that this modification doesn't change the outcome of the game.

- If the princess could've escaped, she may act as before and still escape.
- If the soldiers could act such that princess is always caught somewhere at the field, let them act as before. When soldier gets to the cell containing princess, let him "escort" her to the exit and catch there.

B. Help the Princess!

Consider the following modified version of the game. Allow soldier catch the princess **only** at the exit cell, i. e. arriving to the same cell with the princess doesn't exit the game unless this cell contains the exit.

Let's show that this modification doesn't change the outcome of the game.

- If the princess could've escaped, she may act as before and still escape.
- If the soldiers could act such that princess is always caught somewhere at the field, let them act as before. When soldier gets to the cell containing princess, let him "escort" her to the exit and catch there.

So, the modified version of the game has exactly the same outcome.

B. Help the Princess!

The modified version of the game is much easier to analyze.

B. Help the Princess!

The modified version of the game is much easier to analyze.

It's easy to see that the optimal strategy for both the princess and the soldiers is to move to the exit using the shortest path.

B. Help the Princess!

The modified version of the game is much easier to analyze.

It's easy to see that the optimal strategy for both the princess and the soldiers is to move to the exit using the shortest path.

Calculate the lengths of the shortest paths to the exit by running a breadth-first search (BFS) from the exit cell.

B. Help the Princess!

The modified version of the game is much easier to analyze.

It's easy to see that the optimal strategy for both the princess and the soldiers is to move to the exit using the shortest path.

Calculate the lengths of the shortest paths to the exit by running a breadth-first search (BFS) from the exit cell.

If the distance from the princess cell is smaller or equal to the minimum of the distances from the soldier cells, the princess wins.

B. Help the Princess!

The modified version of the game is much easier to analyze.

It's easy to see that the optimal strategy for both the princess and the soldiers is to move to the exit using the shortest path.

Calculate the lengths of the shortest paths to the exit by running a breadth-first search (BFS) from the exit cell.

If the distance from the princess cell is smaller or equal to the minimum of the distances from the soldier cells, the princess wins.

Otherwise, the soldier wins.

B. Help the Princess!

The modified version of the game is much easier to analyze.

It's easy to see that the optimal strategy for both the princess and the soldiers is to move to the exit using the shortest path.

Calculate the lengths of the shortest paths to the exit by running a breadth-first search (BFS) from the exit cell.

If the distance from the princess cell is smaller or equal to the minimum of the distances from the soldier cells, the princess wins.

Otherwise, the soldier wins.

Complexity of the solution is $\mathcal{O}(WH)$.

C. We Don't Wanna Work!

You are given a set of ACM members with their motivation values and several queries about people entering/leaving ACM. Top-20% are always working hard and the rest of people never work. Your task is to track the events of a person changing its group affiliation after each query.

C. We Don't Wanna Work!

This task is about careful implementation of what is described in the statement. There are several approaches, the one that doesn't require the implementation of any special data structure is described in the further slides.

C. We Don't Wanna Work!

Let's keep all people in two sets W and I, that contain workers and idlers respectively.

C. We Don't Wanna Work!

Let's keep all people in two sets W and I, that contain workers and idlers respectively.

In each set people are sorted according to their motivation, and in case of a tie, according to their entrance date.

C. We Don't Wanna Work!

Let's keep all people in two sets W and I, that contain workers and idlers respectively.

In each set people are sorted according to their motivation, and in case of a tie, according to their entrance date.

At any moment of time two conditions will be fulfilled:

C. We Don't Wanna Work!

Let's keep all people in two sets W and I, that contain workers and idlers respectively.

In each set people are sorted according to their motivation, and in case of a tie, according to their entrance date.

At any moment of time two conditions will be fulfilled:

1. Any person from I is worse than any person from W in terms of motivation and entrance date.

C. We Don't Wanna Work!

Let's keep all people in two sets W and I, that contain workers and idlers respectively.

In each set people are sorted according to their motivation, and in case of a tie, according to their entrance date.

At any moment of time two conditions will be fulfilled:

1. Any person from I is worse than any person from W in terms of motivation and entrance date.
2. $|W| = \left\lceil \frac{|W|+|I|}{5} \right\rceil$.

C. We Don't Wanna Work!

1. Any person from I is worse than any person from W in terms of motivation and entrance date.
2. $|W| = \left\lceil \frac{|W|+|I|}{5} \right\rceil$.

How to satisfy those conditions when people enter or leave?

C. We Don't Wanna Work!

1. Any person from I is worse than any person from W in terms of motivation and entrance date.
2. $|W| = \left\lceil \frac{|W|+|I|}{5} \right\rceil$.

How to satisfy those conditions when people enter or leave?

Forget about second condition for a moment.

C. We Don't Wanna Work!

1. Any person from I is worse than any person from W in terms of motivation and entrance date.
2. $|W| = \left\lceil \frac{|W|+|I|}{5} \right\rceil$.

How to satisfy those conditions when people enter or leave?

Forget about second condition for a moment.

When a person leaves ACM, just remove him from the set it belongs to.

C. We Don't Wanna Work!

1. Any person from I is worse than any person from W in terms of motivation and entrance date.
2. $|W| = \left\lceil \frac{|W|+|I|}{5} \right\rceil$.

How to satisfy those conditions when people enter or leave?

Forget about second condition for a moment.

When a person leaves ACM, just remove him from the set it belongs to.

When a person enters ACM, if it is better than the worst person in W, add him to W, otherwise add him to I.

C. We Don't Wanna Work!

The equality from the second condition may actually become wrong after doing in this naive way.

C. We Don't Wanna Work!

The equality from the second condition may actually become wrong after doing in this naive way.

But if equality was held previously, it couldn't become *too* wrong.

C. We Don't Wanna Work!

The equality from the second condition may actually become wrong after doing in this naive way.

But if equality was held previously, it couldn't become *too* wrong.

The difference between the left hand and the right hand of the equality is bounded by $\mathcal{O}(1)$, in fact it is no more than 1 in absolute value.

In the other words, at most one person belongs to the wrong set.

C. We Don't Wanna Work!

The equality from the second condition may actually become wrong after doing in this naive way.

But if equality was held previously, it couldn't become *too* wrong.

The difference between the left hand and the right hand of the equality is bounded by $\mathcal{O}(1)$, in fact it is no more than 1 in absolute value.

In the other words, at most one person belongs to the wrong set.

After performing the described procedure, if the second condition is not fulfilled, move the best person from I to W, or the worst person from W to I.

C. We Don't Wanna Work!

All we need is to be able to insert people into the sets and to extract the best/worst person in each set.

C. We Don't Wanna Work!

All we need is to be able to insert people into the sets and to extract the best/worst person in each set.

One may use an `std::priority_queue`/`PriorityQueue` or an `std::set`/`TreeSet` to store these sets.

C. We Don't Wanna Work!

All we need is to be able to insert people into the sets and to extract the best/worst person in each set.

One may use an `std::priority_queue`/`PriorityQueue` or an `std::set`/`TreeSet` to store these sets.

Another more complicated ways known to the author are: using a self-written binary search tree like a Cartesian tree, storing everything in a set with keeping the iterator pointing to the first hard-working person, using segment trees…

C. We Don't Wanna Work!

All we need is to be able to insert people into the sets and to extract the best/worst person in each set.

One may use an `std::priority_queue`/`PriorityQueue` or an `std::set`/`TreeSet` to store these sets.

Another more complicated ways known to the author are: using a self-written binary search tree like a Cartesian tree, storing everything in a set with keeping the iterator pointing to the first hard-working person, using segment trees...

Ok, that's cool, but after all, how do we solve a problem?

C. We Don't Wanna Work!

Note that in a described solution the one or two people that change their affiliation are described explicitly:

C. We Don't Wanna Work!

Note that in a described solution the one or two people that change their affiliation are described explicitly:

- If a person enters, output it with its affiliation.

C. We Don't Wanna Work!

Note that in a described solution the one or two people that change their affiliation are described explicitly:

- If a person enters, output it with its affiliation.
- If an operation of moving a person from W to I or vice versa was performed, also output it.

C. We Don't Wanna Work!

Note that in a described solution the one or two people that change their affiliation are described explicitly:

- If a person enters, output it with its affiliation.
- If an operation of moving a person from W to I or vice versa was performed, also output it.

The complexity is $O(\log(|W| + |Q|))$ per query and $O((n+q)\log(n+q))$ overall.

D. Parentheses

In this problem you are to find the lexicographically smallest shortest bracket sequence, that can be transformed into a correct bracket sequence by performing exactly A swaps of adjacent characters and can't be transformed in a smaller number of swaps.

D. Parentheses

The easiest way to solve this problem is to find several first answers by hands or by implementing a naive bruteforce solution and then trying to find the answer pattern.

D. Parentheses

1 ->)(
2 ->)()(
3 ->))((
4 ->)())((
5 ->))()((
6 ->)))(((
7 ->)()))(((
8 ->))())(((
9 ->)))()(((
10 ->))))((((
11 ->)())))((((
12 ->))()))((((
13 ->)))())((((
14 ->))))()((((
15 ->)))))(((((
16 ->)()))))(((((
17 ->))())))(((((
18 ->)))()))(((((
19 ->))))())(((((
20 ->)))))()(((((
21 ->))))))((((((
22 ->)())))))((((((
23 ->))()))))((((((
24 ->)))())))((((((
25 ->))))()))((((((
26 ->)))))())((((((
27 ->))))))()((((((
28 ->)))))))(((((((

D. Parentheses

The sequence of answer strings consists of the groups of equal length of sizes $1, 2, 3, \ldots$.

D. Parentheses

The sequence of answer strings consists of the groups of equal length of sizes $1, 2, 3, \ldots$.

Consider a single group:

```
)()))))(((((
))())))(((((
)))()))(((((
))))())(((((
)))))()(((((
))))))((((((
```

D. Parentheses

The sequence of answer strings consists of the groups of equal length of sizes $1, 2, 3, \ldots$.

Consider a single group:

```
)()))))(((((
))())))(((((
)))()))(((((
))))())(((((
)))))()(((((
))))))((((((
```

It always consists of x strings containing $2x$ brackets.

The i-th (starting from 1) string in a group is always $i \times$ ')' $+$ '(' $+ (x - i) \times$ ')' $+ (x - 1) \times$ '('.

D. Parentheses

The length of the answer is $O(\sqrt{A})$, and the final algorithm is: first find out the desired value of x, and then output the answer by the rule above.

The complexity of the algorithm is $O(\sqrt{A})$.

E. Similarity of Subtrees

We are given the rooted tree consisting of n vertices. Two subtrees are called *similar* if they have the same number of vertices of each depth (if we calculate the depth of a vertex as a distance from the root of a subtree). Calculate the number of pairs of subtrees that are *similar*.

E. Similarity of Subtrees

Let's divide all subtrees into equivalence classes.

E. Similarity of Subtrees

Let's divide all subtrees into equivalence classes.

If we have an equivalence class of size x, it produces exactly $\frac{x(x-1)}{2}$ pairs of equivalent subtrees.

E. Similarity of Subtrees

For a tree T, define its depth-sequence $s(T) = s_0, s_1, s_2, \ldots$ where s_i is the number of vertices of depth i.

E. Similarity of Subtrees

For a tree T, define its depth-sequence $s(T) = s_0, s_1, s_2, \ldots$ where s_i is the number of vertices of depth i.

According to the definition of *similarity*, two trees T_1 and T_2 are similar iff $s(T_1) = s(T_2)$.

E. Similarity of Subtrees

For a tree T, define its depth-sequence $s(T) = s_0, s_1, s_2, \dots$ where s_i is the number of vertices of depth i.

According to the definition of *similarity*, two trees T_1 and T_2 are similar iff $s(T_1) = s(T_2)$.

Instead of comparing the depth-sequences, let's compare their polynomial hashes: $h(T) = (s_0 \cdot g^0 + s_1 \cdot g^1 + s_2 \cdot g^2 + \dots)$ mod P where P is some fixed prime modulo and $n < g < P$ is some fixed number modulo P.

E. Simiarity of Subtrees

Suppose that we have a tree with a root v whose children are $u_1, u_2, \ldots, u_k$. If we denote the whole tree with T_v and the subtrees of v with $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$, then $s(T_v)$ is $s(T_{u_1}) + s(T_{u_2}) + \ldots + s(T_{u_k})$ (a component-wise sum of sequences) prepended with a single 1.

E. Simiarity of Subtrees

Suppose that we have a tree with a root v whose children are $u_1, u_2, \ldots, u_k$. If we denote the whole tree with T_v and the subtrees of v with $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$, then $s(T_v)$ is $s(T_{u_1}) + s(T_{u_2}) + \ldots + s(T_{u_k})$ (a component-wise sum of sequences) prepended with a single 1.

In terms of hashes it means that
$h(T_v) = (1 + g \cdot (h(T_{u_1}) + h(T_{u_2}) + \ldots + h(T_{u_k}))) \mod P$.

E. Simiarity of Subtrees

Suppose that we have a tree with a root v whose children are $u_1, u_2, \ldots, u_k$. If we denote the whole tree with T_v and the subtrees of v with $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$, then $s(T_v)$ is $s(T_{u_1}) + s(T_{u_2}) + \ldots + s(T_{u_k})$ (a component-wise sum of sequences) prepended with a single 1.

In terms of hashes it means that
$h(T_v) = (1 + g \cdot (h(T_{u_1}) + h(T_{u_2}) + \ldots + h(T_{u_k}))) \mod P$.

Calculate hashes of all subtrees in a single DFS.

E. Simiarity of Subtrees

Suppose that we have a tree with a root v whose children are $u_1, u_2, \ldots, u_k$. If we denote the whole tree with T_v and the subtrees of v with $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$, then $s(T_v)$ is $s(T_{u_1}) + s(T_{u_2}) + \ldots + s(T_{u_k})$ (a component-wise sum of sequences) prepended with a single 1.

In terms of hashes it means that
$h(T_v) = (1 + g \cdot (h(T_{u_1}) + h(T_{u_2}) + \ldots + h(T_{u_k}))) \mod P$.

Calculate hashes of all subtrees in a single DFS.

Divide all subtrees into equivalence classes and calculate the asnwer.

E. Simiarity of Subtrees

Suppose that we have a tree with a root v whose children are $u_1, u_2, \dots, u_k$. If we denote the whole tree with T_v and the subtrees of v with $T_{u_1}, T_{u_2}, \dots, T_{u_k}$, then $s(T_v)$ is $s(T_{u_1}) + s(T_{u_2}) + \dots + s(T_{u_k})$ (a component-wise sum of sequences) prepended with a single 1.

In terms of hashes it means that $h(T_v) = (1 + g \cdot (h(T_{u_1}) + h(T_{u_2}) + \dots + h(T_{u_k}))) \mod P$.

Calculate hashes of all subtrees in a single DFS.

Divide all subtrees into equivalence classes and calculate the asnwer.

The solution complexity: $\mathcal{O}(n \log n)$ (or $\mathcal{O}(n)$ depending on how we calculate the equivalnce classes).

F. Escape From the Hell

You have a long rope of length L that connects hell and whatever is located above the hell. Your friend is trying to escape the hell by drinking energy drinks. When drinking the i-th drink, he climbs A_i meters up during the daytime and slides down B_i meters during the night. Also there are sinners that are trying to catch him by climbing C_i meters during the i-th night.

Find out what is the earliest day he can escape hell or find out that it is impossible.

F. Escape From the Hell

Let's find how the optimal sequence of energy drinks looks like.

F. Escape From the Hell

Let's find how the optimal sequence of energy drinks looks like.

Consider all drinks except the last one. Let them be (A_1, B_1), (A_2, B_2), ..., (A_k, B_k), and let the last drink be (A^*, B^*).

F. Escape From the Hell

Let's find how the optimal sequence of energy drinks looks like.

Consider all drinks except the last one. Let them be (A_1, B_1), (A_2, B_2), ..., (A_k, B_k), and let the last drink be (A^*, B^*).

We know that $\sum A_i - \sum B_i \geqslant L - A^*$.

F. Escape From the Hell

Let's find how the optimal sequence of energy drinks looks like.

Consider all drinks except the last one. Let them be (A_1, B_1), (A_2, B_2), …, (A_k, B_k), and let the last drink be (A^*, B^*).

We know that $\sum A_i - \sum B_i \geqslant L - A^*$.

By the end of the night i we will arrive to the point $H_i = (A_1 + \ldots + A_i) - (B_1 + \ldots + B_i)$.

F. Escape From the Hell

Let's show that in the optimal answer $(A_i - B_i) \geqslant (A_{i+1} - B_{i+1})$, i. e. all drinks except the last one follow in the order of decreasing "effectiveness", where effectiveness is $A_i - B_i$.

F. Escape From the Hell

Let's show that in the optimal answer $(A_i - B_i) \geqslant (A_{i+1} - B_{i+1})$, i. e. all drinks except the last one follow in the order of decreasing "effectiveness", where effectiveness is $A_i - B_i$.

Indeed, if we have $(A_i - B_i) < (A_{i+1} - B_{i+1})$, let's swap those two drinks. All H_j will remain the same with the only exception of H_i that will become larger. It's easy to see that this is only better for us (i. e. if we weren't caught by the sinners before this modification, we won't also be caught now).

F. Escape From the Hell

Put all drinks in the descending order of $(A_i - B_i)$. We will now choose some particular drink (A^*, B^*) as the last one, remove it from this order, check if we are not caught by the sinners, and find out the number of days we will get to the height of $L - A^*$.

F. Escape From the Hell

Put all drinks in the descending order of $(A_i - B_i)$. We will now choose some particular drink (A^*, B^*) as the last one, remove it from this order, check if we are not caught by the sinners, and find out the number of days we will get to the height of $L - A^*$.

Let's try drinks to be final in order from left to right one by one. Denote as p the first day that we are caught by sinners.

F. Escape From the Hell

Put all drinks in the descending order of $(A_i - B_i)$. We will now choose some particular drink (A^*, B^*) as the last one, remove it from this order, check if we are not caught by the sinners, and find out the number of days we will get to the height of $L - A^*$.

Let's try drinks to be final in order from left to right one by one. Denote as p the first day that we are caught by sinners.

It is true that while we choose each new drink as a final one, p never decreases. This is true since while we do this, H_i only increases for all i from 1 to $n - 1$.

F. Escape From the Hell

Put all drinks in the descending order of $(A_i - B_i)$. We will now choose some particular drink (A^*, B^*) as the last one, remove it from this order, check if we are not caught by the sinners, and find out the number of days we will get to the height of $L - A^*$.

Let's try drinks to be final in order from left to right one by one. Denote as p the first day that we are caught by sinners.

It is true that while we choose each new drink as a final one, p never decreases. This is true since while we do this, H_i only increases for all i from 1 to $n - 1$.

Actually process the final drinks one by one, and keep the current value of p.

F. Escape From the Hell

At each step check if by the moment of p we are at least on the hight $L - A^*$, and if yes, find out the moment when this happened using the binary search.

F. Escape From the Hell

At each step check if by the moment of p we are at least on the hight $L - A^*$, and if yes, find out the moment when this happened using the binary search.

It may be convenient to use some data structure here, like a Cartesian tree. It is possible to implement this solution without a single data structure (using only the deque), but the implementation I came up with contained a lot of ± 1 in indices, and was almost harder then the one with some auxillary Cartesian tree.

G. Share the Ruins Preservation

You are given n distinct points on a plane. Drawing an arbitrary vertical line without crossing any of the points, and calculate the sum of areas of convex hulls of two formed set of points. Your goal is to find the minimum possible sum of these areas among all choices of a vertical line.

G. Share the Ruins Preservation

There are at most $n + 1$ different choices of a vertical line.

G. Share the Ruins Preservation

There are at most $n + 1$ different choices of a vertical line.

For each of the choices we will calculate both the convex hull area of the left part and the right part.

G. Share the Ruins Preservation

There are at most $n + 1$ different choices of a vertical line.

For each of the choices we will calculate both the convex hull area of the left part and the right part.

We will start with calculating the areas of the left convex hulls. After that we will deal with the right part in a similar way.

G. Share the Ruins Preservation

Process possible locations of a vertical line from left to right.
Some points will be added to our convex hull.

G. Share the Ruins Preservation

Process possible locations of a vertical line from left to right. Some points will be added to our convex hull.

It is possible to implement a data structure for keeping a dynamic convex hull of points that allows adding arbitrary points in amortized $O(\log n)$ per query, but it is pretty complex.

G. Share the Ruins Preservation

Process possible locations of a vertical line from left to right. Some points will be added to our convex hull.

It is possible to implement a data structure for keeping a dynamic convex hull of points that allows adding arbitrary points in amortized $O(\log n)$ per query, but it is pretty complex.

We didn't utilize the important property of the points that are added.

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

We will build the upper hull and the lower hull separately. Consider the upper hull.

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

We will build the upper hull and the lower hull separately. Consider the upper hull.

Add points from left to right. Store all points of the current upper hull in the stack.

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

We will build the upper hull and the lower hull separately. Consider the upper hull.

Add points from left to right. Store all points of the current upper hull in the stack.

When adding a new point, remove the points from the top of the stack, until the newly added point forms the concave angle with two previous ones.

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

We will build the upper hull and the lower hull separately. Consider the upper hull.

Add points from left to right. Store all points of the current upper hull in the stack.

When adding a new point, remove the points from the top of the stack, until the newly added point forms the concave angle with two previous ones.

This works in $O(n)$ after sorting the points in order of increasing x.

G. Share the Ruins Preservation

Recall the Andrew's convex hull algorithm (also known as a Graham-Andrew algorithm according to e-maxx).

We will build the upper hull and the lower hull separately. Consider the upper hull.

Add points from left to right. Store all points of the current upper hull in the stack.

When adding a new point, remove the points from the top of the stack, until the newly added point forms the concave angle with two previous ones.

This works in $O(n)$ after sorting the points in order of increasing x.

Sounds pretty useful in our case, and this algorithm actually knows all the intermediate convex hulls, let's use it!

G. Share the Ruins Preservation

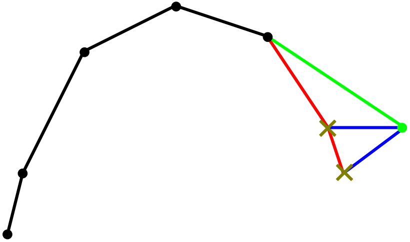

Figure: One step of an Andrew's convex hull algorithm

G. Share the Ruins Preservation

Let's keep not only the hull points in the stack, but also the trapezoid areas below the hull.

When removing the points from the stack, subtract their trapezoid araes from the hull area, when adding a new point, add the newly formed trapezoid area to the hull area.

G. Share the Ruins Preservation

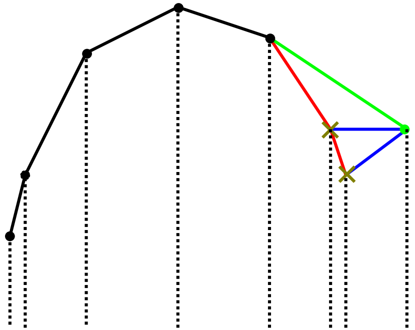

Figure: One step of an Andrew's convex hull algorithm

G. Share the Ruins Preservation

This allows us to know the area below the upper hull and also the area below the lower hull.

G. Share the Ruins Preservation

This allows us to know the area below the upper hull and also the area below the lower hull.

The desired convex hull area is the difference of those two areas.

G. Share the Ruins Preservation

This allows us to know the area below the upper hull and also the area below the lower hull.

The desired convex hull area is the difference of those two areas.

This provides us with an $O(n \log n)$ solution (here the sorting phase takes $O(n \log n)$, the rest of the solution works in $O(n)$).

H. Pipe Fitter and the Fierce Dogs

You have a rectangular field $W \times H$ where W and H are odd. In each cell with odd coordinates there is a house. You may connect K of these house with an underground source of water. The other houses should be connected indirectly with pipes.

You may put a pipe that is oriented like '/', '|' or '\' in the cells in even rows making the water flow from the upper house to the lower house. Some of the cells in even rows contain dogs, and putting a pipe in such a cell is twice as expensive comparing to the usual pipe.

Find out the minimum cost of connecting all houses to water.

H. Pipe Fitter and the Fierce Dogs

First of all, each of the $(W+1)/2$ topmost houces should be connected to the water using the underground pipe. If K pipes isn't enough for that then the answer desired is impossible.

Otherwise we can always connect all houses vertically.

H. Pipe Fitter and the Fierce Dogs

In an ideal world where there are no dogs, we would obviously use $\max(\frac{W+1}{2} \cdot \frac{H-1}{2} - K, 0)$ pipes and spend exactly that much money.

H. Pipe Fitter and the Fierce Dogs

In an ideal world where there are no dogs, we would obviously use $\max(\frac{W+1}{2} \cdot \frac{H-1}{2} - K, 0)$ pipes and spend exactly that much money.

In a real world, we will first try to build a network using no extra underground sources that minimzes the number of pipes passing through the dogs.

H. Pipe Fitter and the Fierce Dogs

In an ideal world where there are no dogs, we would obviously use $\max(\frac{W+1}{2} \cdot \frac{H-1}{2} - K, 0)$ pipes and spend exactly that much money.

In a real world, we will first try to build a network using no extra underground sources that minimzes the number of pipes passing through the dogs.

After that by spending one extra underground connection, we can discard one pipe passing through the dog.

H. Pipe Fitter and the Fierce Dogs

Consider two consecutive rows of houses.

H. Pipe Fitter and the Fierce Dogs

Consider two consecutive rows of houses.

Note that we may find an optimal layout without two pipes crossing in form of `X` through the cell with a dog: they may always be replaced with two parallel vertical pipes.

H. Pipe Fitter and the Fierce Dogs

Consider two consecutive rows of houses.

Note that we may find an optimal layout without two pipes crossing in form of `X` through the cell with a dog: they may always be replaced with two parallel vertical pipes.

Call the dogs staying at the even columns *even* dogs, and the dogs staying at the odd columns *odd* dogs (recall that the houses are located in the odd columns).

H. Pipe Fitter and the Fierce Dogs

Extract the segments free from the even dogs. In such segments any four houses forming a square may be connected in a form of `X`.

H. Pipe Fitter and the Fierce Dogs

Extract the segments free from the even dogs. In such segments any four houses forming a square may be connected in a form of X.

If such segment consists of the even number of houses in each row, just connect them by pairs using X-connections.

H. Pipe Fitter and the Fierce Dogs

Extract the segments free from the even dogs. In such segments any four houses forming a square may be connected in a form of `X`.

If such segment consists of the even number of houses in each row, just connect them by pairs using `X`-connections.

Otherwise, check if there exists a pair of houses that may be connected vertically, such that there is an even number of houses to the left and to the right of it. If it exists, use it. Otherwise we have to make exactly one vertical connection passing through the dog cell.

H. Pipe Fitter and the Fierce Dogs

Calculate the minimum number of affected dogs using the described algorithm, then discard at most $K - \frac{W+1}{2}$ of them and calculate the final cost of such a layout.

The complexity of the described solution is $O(n \log n)$.

I. Multisect

You have some generalized version of a binary search. You know that $f(l) = 0$, $f(r) = 1$, $f(x)$ is always 0 or 1 and it is non-decreasing. Your task is to find the critical value c such that $f(c) = 0$ and $f(c+1) = 1$.

You know that c is chosen equiprobably from l to $r-1$, and you may perform the following operation. You choose $p \leqslant k$ points $x_1, \ldots, x_p$, calculate $f(x_1), \ldots, f(x_p)$, and spend T_z time where z is the number of points x_i such that $f(x_i)$ equals to 0 and $T_0, T_1, \ldots, T_k$ are given constants.

Find out the best possible expected time of finding c.

I. Multisect

This is a typical “find the best strategy problem” that is solved using DP.

I. Multisect

This is a typical "find the best strategy problem" that is solved using DP.

At any moment the state is described using the only number: we know the largest l such that $f(l) = 0$ and the smallest r such that $f(r) = 1$, and the only thing that is important for us here is the value of $r - l$.

I. Multisect

This is a typical "find the best strategy problem" that is solved using DP.

At any moment the state is described using the only number: we know the largest l such that $f(l) = 0$ and the smallest r such that $f(r) = 1$, and the only thing that is important for us here is the value of $r - l$.

Let's try to calculate the value $D[x]$: the best expected time that we will spend on investigation if current segment length is exactly x.

I. Multisect

This is a typical "find the best strategy problem" that is solved using DP.

At any moment the state is described using the only number: we know the largest l such that $f(l) = 0$ and the smallest r such that $f(r) = 1$, and the only thing that is important for us here is the value of $r - l$.

Let's try to calculate the value $D[x]$: the best expected time that we will spend on investigation if current segment length is exactly x.

Obviously, $D[1] = 0$ since when $r - l = 1$, we definitely know that the optimal c is l.

I. Multisect

Suppose that we've chosen p points $0 < x_1 < x_2 < \ldots < x_p < x$ (assume $l = 0$, $r = x$). What is the expected time we will spend in such case?

I. Multisect

Suppose that we've chosen p points $0 < x_1 < x_2 < \ldots < x_p < x$ (assume $l = 0$, $r = x$). What is the expected time we will spend in such case?

The expected time for a current turn is:
$\frac{x_1}{x} T_0 + \frac{x_2 - x_1}{x} T_1 + \ldots + \frac{x_p - x_{p-1}}{x} T_{p-1} + \frac{x - x_p}{x} T_p$.

I. Multisect

Suppose that we've chosen p points $0 < x_1 < x_2 < \ldots < x_p < x$ (assume $l = 0$, $r = x$). What is the expected time we will spend in such case?

The expected time for a current turn is:
$\frac{x_1}{x} T_0 + \frac{x_2 - x_1}{x} T_1 + \ldots + \frac{x_p - x_{p-1}}{x} T_{p-1} + \frac{x - x_p}{x} T_p$.

The expected time for the further investigation is:
$\frac{x_1}{x} D[x_1] + \frac{x_2 - x_1}{x} D[x_2 - x_1] + \ldots + \frac{x_p - x_{p-1}}{x} D[x_p - x_{p-1}] + \frac{x - x_p}{x} D[x - x_p]$.

I. Multisect

Suppose that we've chosen p points $0 < x_1 < x_2 < \ldots < x_p < x$ (assume $l = 0$, $r = x$). What is the expected time we will spend in such case?

The expected time for a current turn is:
$\frac{x_1}{x} T_0 + \frac{x_2 - x_1}{x} T_1 + \ldots + \frac{x_p - x_{p-1}}{x} T_{p-1} + \frac{x - x_p}{x} T_p$.

The expected time for the further investigation is:
$\frac{x_1}{x} D[x_1] + \frac{x_2 - x_1}{x} D[x_2 - x_1] + \ldots + \frac{x_p - x_{p-1}}{x} D[x_p - x_{p-1}] + \frac{x - x_p}{x} D[x - x_p]$.

That looks like a formula for a DP, but we can't just iterate over all possible choices of $(x_1, x_2, \ldots, x_p)$, there are too many of them.

I. Multisect

Let's introduce a supplementary value that we will also calculate using the DP: $F[p][x_p] =$ the smallest possible sum of first p terms in two previously written sums with a given value of x_p.

I. Multisect

Let's introduce a supplementary value that we will also calculate using the DP: $F[p][x_p] =$ the smallest possible sum of first p terms in two previously written sums with a given value of x_p.

It's easy to express $F[p][x_p]$ through $F[p-1][x_{p-1}]$ and two extra summands.

I. Multisect

Let's introduce a supplementary value that we will also calculate using the DP: $F[p][x_p]$ = the smallest possible sum of first p terms in two previously written sums with a given value of x_p.

It's easy to express $F[p][x_p]$ through $F[p-1][x_{p-1}]$ and two extra summands.

The overall complexity of both these two DPs is $O(x^2k)$ where $x = r - l$.

J. Compressed Formula

You are given an RLE (run-length encoding) of a long arithmetic expression consisting only of digits, addition, subtraction and multiplication without brackets. Calculate its value modulo $10^9 + 7$.

J. Compressed Formula

An insight: when you see a problem where you are asked to perform up to 10^9 repetitions of some actions, it is a good idea to think about...

J. Compressed Formula

An insight: when you see a problem where you are asked to perform up to 10^9 repetitions of some actions, it is a good idea to think about...

Ah, whatever. Let's just solve the problem.

J. Compressed Formula

Some of the parts that we have in the input may consist of a single character (for example, '+'). Thus, it is useful to understand how the value of the expression is modified when we add a single character to it.

J. Compressed Formula

Add characters one by one. At any moment our formula looks like the following:

$$C \pm K \cdot X$$

- Constant C is the sum of all terms to the left of the one we are currently processing;
- Constant K is the product of all numbers to the left of the number in the last term we are currently processing;
- Constant X is the last number in the last term.

J. Compressed Formula

Examples:

- $12 \cdot 34 + 56 \cdot 78 \cdot 901$: $C = 12 \cdot 34$, $K = 56 \cdot 78$, $X = 901$;
- $34 + 56$: $C = 34$, $K = 1$, $X = 56$ (if K consists of zero numbers, define it with 1);
- $123 - 57 \cdot 24$: $C = 123$, $K = -57$, $X = 24$ (if there is a minus before K, make K negative itself);
- 42: $C = 0$, $K = 1$, $X = 42$ (if there are no summands before the last one, make $C = 0$).

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

If a is a digit, then C and K remain the same and X is replaced with $Xa = 10 \cdot X + a$.

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

If a is a digit, then C and K remain the same and X is replaced with $Xa = 10 \cdot X + a$.

If a is `'*'`, then C remains the same, K is replaced with $K \cdot X$ and X is replaced with 0.

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

If a is a digit, then C and K remain the same and X is replaced with $Xa = 10 \cdot X + a$.

If a is '*', then C remains the same, K is replaced with $K \cdot X$ and X is replaced with 0.

If a is '+' or '−', then C is replaced with $C + K \cdot X$, K is replaced with ± 1 and X is replaced with 0.

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

If a is a digit, then C and K remain the same and X is replaced with $Xa = 10 \cdot X + a$.

If a is `'*'`, then C remains the same, K is replaced with $K \cdot X$ and X is replaced with 0.

If a is `'+'` or `'-'`, then C is replaced with $C + K \cdot X$, K is replaced with ± 1 and X is replaced with 0.

That sounds like a bunch of linear operations, we can use matrix multiplication for performing all of this!

J. Compressed Formula

Suppose we have added a single character a to the whole string. How have C, K, X changed?

If a is a digit, then C and K remain the same and X is replaced with $Xa = 10 \cdot X + a$.

If a is `'*'`, then C remains the same, K is replaced with $K \cdot X$ and X is replaced with 0.

If a is `'+'` or `'-'`, then C is replaced with $C + K \cdot X$, K is replaced with ± 1 and X is replaced with 0.

That sounds like a bunch of linear operations, we can use matrix multiplication for performing all of this!

Aw, snap. We cannot multiply one variable by another using matrix multiplication ($K \to K \cdot X$).

J. Compressed Formula

Our issue is that we can add a lot of digits and they they affect K only when we "flush" them with a new '*'. The way they affect K is pretty complicated: multiplication is kind of a hard operation, and linear transformations do not have any kind of a large memory, so it's better to keep something else that is affected by new digits immediately.

J. Compressed Formula

Our issue is that we can add a lot of digits and they they affect K only when we "flush" them with a new '$*$'. The way they affect K is pretty complicated: multiplication is kind of a hard operation, and linear transformations do not have any kind of a large memory, so it's better to keep something else that is affected by new digits immediately.

Don't worry if you didn't understand what is written above, I just tried to sound cool.

J. Compressed Formula

Let's deal with C, K and $K \cdot X$. Suppose we have added a character a.

J. Compressed Formula

Let's deal with C, K and $K \cdot X$. Suppose we have added a character a.

If a is a digit, then C and K remain the same and $K \cdot X$ is replaced with $K \cdot Xa = K \cdot (10 \cdot X + a) = 10 \cdot K \cdot X + a \cdot K$.

J. Compressed Formula

Let's deal with C, K and $K \cdot X$. Suppose we have added a character a.

If a is a digit, then C and K remain the same and $K \cdot X$ is replaced with $K \cdot Xa = K \cdot (10 \cdot X + a) = 10 \cdot K \cdot X + a \cdot K$.

If a is '$*$', then C remains the same, K is replaced with $K \cdot X$ and $K \cdot X$ is replaced with 0.

J. Compressed Formula

Let's deal with C, K and $K \cdot X$. Suppose we have added a character a.

If a is a digit, then C and K remain the same and $K \cdot X$ is replaced with $K \cdot Xa = K \cdot (10 \cdot X + a) = 10 \cdot K \cdot X + a \cdot K$.

If a is '*', then C remains the same, K is replaced with $K \cdot X$ and $K \cdot X$ is replaced with 0.

If a is '+' or '−', then C is replaced with $C + K \cdot X$, K is replaced with ± 1 and $K \cdot X$ is replaced with 0.

J. Compressed Formula

Let's deal with C, K and $K \cdot X$. Suppose we have added a character a.

If a is a digit, then C and K remain the same and $K \cdot X$ is replaced with $K \cdot Xa = K \cdot (10 \cdot X + a) = 10 \cdot K \cdot X + a \cdot K$.

If a is '*', then C remains the same, K is replaced with $K \cdot X$ and $K \cdot X$ is replaced with 0.

If a is '+' or '-', then C is replaced with $C + K \cdot X$, K is replaced with ± 1 and $K \cdot X$ is replaced with 0.

We did it! All operations now look like "we add one variable to another with some constant coefficient".

J. Compressed Formula

The only small detail to discuss is how to replace K with ± 1 in the last step. By multiplying matrix onto a vector we can't add a constant to its component.

J. Compressed Formula

The only small detail to discuss is how to replace K with ± 1 in the last step. By multiplying matrix onto a vector we can't add a constant to its component.

That's usually not a big deal. Just create one more variable $E = 1$ that doesn't change under any transformation, i. e. it always replaced with itself.

J. Compressed Formula

The only small detail to discuss is how to replace K with ± 1 in the last step. By multiplying matrix onto a vector we can't add a constant to its component.

That's usually not a big deal. Just create one more variable $E = 1$ that doesn't change under any transformation, i. e. it always replaced with itself.

Hooray, now we have 13 matrices corresponding to any of the characters that may appear in the input stream.

J. Compressed Formula

The only small detail to discuss is how to replace K with ± 1 in the last step. By multiplying matrix onto a vector we can't add a constant to its component.

That's usually not a big deal. Just create one more variable $E = 1$ that doesn't change under any transformation, i. e. it always replaced with itself.

Hooray, now we have 13 matrices corresponding to any of the characters that may appear in the input stream.

Calculate the product of all matrices corresponding to the input, and multiply it by the vector $(C = 0, K = 1, K \cdot X = 0, E = 1)$. The answer is $C + K \cdot X$.

J. Compressed Formula

Perform all calculations modulo $10^9 + 7$. Raise matrix to the powers using the fast power algorithm.

J. Compressed Formula

Perform all calculations modulo $10^9 + 7$. Raise matrix to the powers using the fast power algorithm.

The overall complexity is $\mathcal{O}(\sum |s_i| \log \max r_i)$.

An interesting fact: the model solution for this problem consists of 1180 lines of code with lots of strange comments in Japanese. The solution described above can be implemented in 94 lines of code with zero strange comments in Japanese. Matrix multiplication rules!

K. Non-redundant Drive

You are given an undirected tree, whose edges have positive lengths. In each vertex there is a gas station with a certain amount of fuel. You are driving a car that uses 1 liter of gas per kilometer, your task is to find a longest simple path that you make take using a car.

K. Non-redundant Drive

The problem asks you to find the path that has some complicated properties.

K. Non-redundant Drive

The problem asks you to find the path that has some complicated properties.

The good direction for a thinking in such a kind of problems is a *centroid decomposition*.

K. Non-redundant Drive

Consider one layer of a centroid decomposition that is a connected subtree with its center c.

K. Non-redundant Drive

Consider one layer of a centroid decomposition that is a connected subtree with its center c.

Let's find the longest valid path that passes through c, then remove c from consideration and run our solution recursively from all the remaining parts.

K. Non-redundant Drive

Consider one layer of a centroid decomposition that is a connected subtree with its center c.

Let's find the longest valid path that passes through c, then remove c from consideration and run our solution recursively from all the remaining parts.

This is a common scheme for a centroid decomposition solution.

K. Non-redundant Drive

Make c to be the root of a current layer. Each path passing through c consists of two parts: an ascending part and a descending part. Suppose that the path starts in a vertex s and ends in a vertex t. Let's deal with each of these two parts.

K. Non-redundant Drive

An ascending path from s is valid if we can get to c without going out of fuel. Temporarily allow having the negative amount of fuel.

K. Non-redundant Drive

An ascending path from s is valid if we can get to c without going out of fuel. Temporarily allow having the negative amount of fuel.

For each vertex s calculate two values:

K. Non-redundant Drive

An ascending path from s is valid if we can get to c without going out of fuel. Temporarily allow having the negative amount of fuel.

For each vertex s calculate two values:

- $asc\_bal[s]$ is the final fuel balance if we get from s to the root **without** charging at the root (this will be convenient in the further calculations)

K. Non-redundant Drive

An ascending path from s is valid if we can get to c without going out of fuel. Temporarily allow having the negative amount of fuel.

For each vertex s calculate two values:

- $asc\_bal[s]$ is the final fuel balance if we get from s to the root **without** charging at the root (this will be convenient in the further calculations)
- $asc\_min\_bal[s]$ is the minimum fuel balance on the path from s to the root

K. Non-redundant Drive

An ascending path from s is valid if we can get to c without going out of fuel. Temporarily allow having the negative amount of fuel.

For each vertex s calculate two values:

- $asc\_bal[s]$ is the final fuel balance if we get from s to the root **without** charging at the root (this will be convenient in the further calculations)
- $asc\_min\_bal[s]$ is the minimum fuel balance on the path from s to the root

An ascending path from s is valid iff $asc\_min\_bal[s] \geqslant 0$.

K. Non-redundant Drive

Consider a descending path ending in t. Suppose that we start in the root vertex with charging in it.

K. Non-redundant Drive

Consider a descending path ending in t. Suppose that we start in the root vertex with charging in it.

For each vertex t calculate the following value:

K. Non-redundant Drive

Consider a descending path ending in t. Suppose that we start in the root vertex with charging in it.

For each vertex t calculate the following value:

$desc\_min\_bal[s]$ that is equal to the minimum fuel balance on the path from the root to the t.

K. Non-redundant Drive

Consider a descending path ending in t. Suppose that we start in the root vertex with charging in it.

For each vertex t calculate the following value:

$desc\_min\_bal[s]$ that is equal to the minimum fuel balance on the path from the root to the t.

Depending on it, we can not say definitely if the path from root to t is valid or not because it also depends on the ascending part of the whole path. Let's try to use the previously calculated values.

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

- $asc\_min\_bal[s] \geqslant 0$;

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

- $asc\_min\_bal[s] \geqslant 0$;
- $asc\_bal[s] + desc\_min\_bal[t] \geqslant 0$;

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

- $asc\_min\_bal[s] \geqslant 0$;
- $asc\_bal[s] + desc\_min\_bal[t] \geqslant 0$;
- s and t are from different subtrees of the root.

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

- $asc\_min\_bal[s] \geqslant 0$;
- $asc\_bal[s] + desc\_min\_bal[t] \geqslant 0$;
- s and t are from different subtrees of the root.

The first condition is easy to fulfill: remove all violating vertices s from the consideration.

K. Non-redundant Drive

A pair of s and t forms a valid path if the following conditions are held:

- $asc\_min\_bal[s] \geqslant 0$;
- $asc\_bal[s] + desc\_min\_bal[t] \geqslant 0$;
- s and t are from different subtrees of the root.

The first condition is easy to fulfill: remove all violating vertices s from the consideration.

To fulfill the second and the third conditions, one should call upon the Dark Forces of the Data Structures!

K. Non-redundant Drive

Let's fix an arbitrary order on the subtrees. Suppose that s comes from the earlier subtree than t (and then do the same in the reversed order of subtrees).

K. Non-redundant Drive

Let's fix an arbitrary order on the subtrees. Suppose that s comes from the earlier subtree than t (and then do the same in the reversed order of subtrees).

Summon a Fenwick Tree from the Eternal Abyss of Logarithmic Data Structures, ask it to deal with a max operation on suffixes (don't know how to work with suffixes in a Fenwick Tree? You can always work with prefixes using the Sacred Power of Symmetry).

K. Non-redundant Drive

Let's fix an arbitrary order on the subtrees. Suppose that s comes from the earlier subtree than t (and then do the same in the reversed order of subtrees).

Summon a Fenwick Tree from the Eternal Abyss of Logarithmic Data Structures, ask it to deal with a max operation on suffixes (don't know how to work with suffixes in a Fenwick Tree? You can always work with prefixes using the Sacred Power of Symmetry).

We will store s-vertices in it, using $asc\_bal[s]$ as a key and $depth[s]$ as a value.

K. Non-redundant Drive

Fix a subtree for t. At this point we will have all s vertices from the previous trees stored in a Fenwick Tree. Now for each vertex t consider the suffix in the Fenwick tree defined by the inequality $asc\_bal[s] \geqslant -desc\_min\_bal[t]$, and find the minimum value of $depth[s]$ over it. This will provide us with the best choice of s for a given t.

K. Non-redundant Drive

Fix a subtree for t. At this point we will have all s vertices from the previous trees stored in a Fenwick Tree. Now for each vertex t consider the suffix in the Fenwick tree defined by the inequality $asc\_bal[s] \geqslant -desc\_min\_bal[t]$, and find the minimum value of $depth[s]$ over it. This will provide us with the best choice of s for a given t.

Relax the answer with $depth[s] + depth[t] + 1$.

K. Non-redundant Drive

Fix a subtree for t. At this point we will have all s vertices from the previous trees stored in a Fenwick Tree. Now for each vertex t consider the suffix in the Fenwick tree defined by the inequality $asc\_bal[s] \geqslant -desc\_min\_bal[t]$, and find the minimum value of $depth[s]$ over it. This will provide us with the best choice of s for a given t.

Relax the answer with $depth[s] + depth[t] + 1$.

Feel the power that flows through your veins.

K. Non-redundant Drive

Fix a subtree for t. At this point we will have all s vertices from the previous trees stored in a Fenwick Tree. Now for each vertex t consider the suffix in the Fenwick tree defined by the inequality $asc\_bal[s] \geqslant -desc\_min\_bal[t]$, and find the minimum value of $depth[s]$ over it. This will provide us with the best choice of s for a given t.

Relax the answer with $depth[s] + depth[t] + 1$.

Feel the power that flows through your veins.

As we process a single layer in $\mathcal{O}(s \log s)$, where s is the size of the current layer, the overall complexity is $\mathcal{O}(n \log^2 n)$.

K. Non-redundant Drive

Fix a subtree for t. At this point we will have all s vertices from the previous trees stored in a Fenwick Tree. Now for each vertex t consider the suffix in the Fenwick tree defined by the inequality $asc\_bal[s] \geqslant -desc\_min\_bal[t]$, and find the minimum value of $depth[s]$ over it. This will provide us with the best choice of s for a given t.

Relax the answer with $depth[s] + depth[t] + 1$.

Feel the power that flows through your veins.

As we process a single layer in $\mathcal{O}(s \log s)$, where s is the size of the current layer, the overall complexity is $\mathcal{O}(n \log^2 n)$.

~~Sorry, my sense of humour is terrible.~~