Problem A. Hill

- $n = 1$. 12 **points.**

  Enumerate all possible pairs (L_1, R_1) and pick the best one. It can be done in $O(m^2)$ or $O(m)$.

- **No blocked cells.** 7 **points.**

  It is always optimal to choose the whole table as a hill. So the answer is just the sum of all values.

- $n, m \leq 50$. 25 **points.**

  Let's maintain $dp_{i,l,r}$ — the best sum of a hill we can achieve so far if it ended on row i with a segment (l, r).

  To calculate it, enumerate all pairs (l_p, r_p) such that $l \leq l_p \leq r_p \leq r$ and try to extend dp_{i-1,l_p,r_p}. Also consider the case when you start a new hill in the current row.

  The total time complexity will be $O(nm^4)$ with a small constant.

- $n, m \leq 300$. 22 **points.**

  Exactly the same solution as above, but this time instead of enumerating (l_p, r_p) we can notice that we are just looking for the rectangle maximum over all $l \leq l_p \leq r_p \leq r$. That can be maintained with another DP, say $best_{i,l,r}$. For the transition you will consider $dp_{i,l,r}$, $best_{i,l+1,r}$ or $best_{i,l,r-1}$.

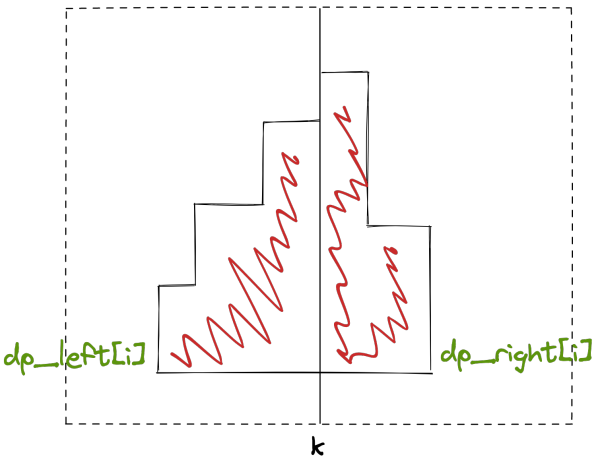
  The total time complexity is now $O(nm^2)$, if implemented carefully.

- **No further constraints.** 34 **points.**

  A hill, in its current definition, is quite hard to compute directly. So let's try to make it simpler instead.

  By looking at the hill's properties, we notice that it will be increasing up to some column, and then decreasing. Let's enumerate that column k, and split the hill into left and right part. The left should be increasing and the right should be decreasing. We find that they are completely symmetrical, so let's focus on calculating the left part only.

  Let's define dp_i — the best possible sum of the hill, if its **bottom** starts at row i (alternatively, $e = i$) and its bottom-left starts somewhere to the left of column k. Similarly, we will define such a dp on the right side and try to combine the results.

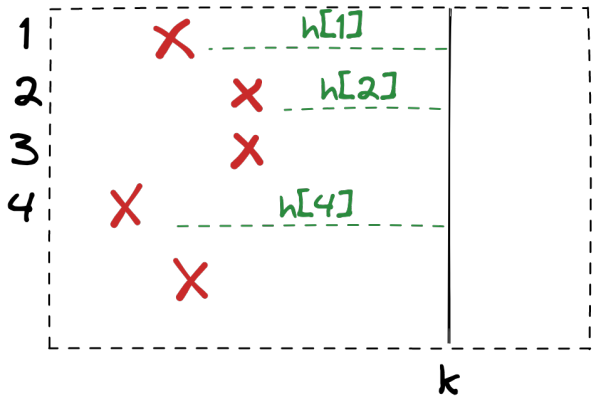


Hill splitted at column k.

It is clear that once we have calculated $dp_1, \ldots, dp_n$ values both on the left and the right, we could combine them by enumerating e from 1 to n and taking the biggest possible sum $(dpLeft_e + dpRight_e)$.

So if we are somehow able to calculate dp in $O(n \log n)$ or $O(n)$ time, the problem will be solved.

Let's focus on calculating dp_1. We should try to take the longest row as possible, until we hit a blocked cell. Let h_1 denote the maximum number of cells we can take. Similarly, we will define $h_2, \ldots, h_n$. Each one of them can be found in $O(1)$, if you pre-calculate the closest blocked cell on the left and on the right for each cell (i, j).

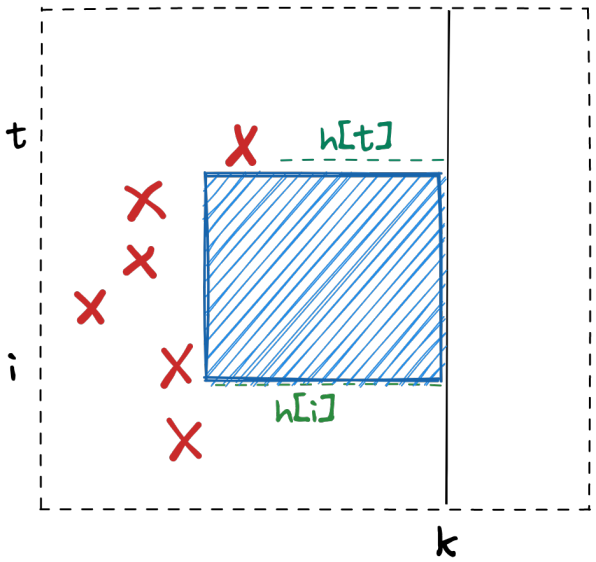


How values h_i are defined.

Now it should be more clear how to calculate the value of dp_i. We will first start by taking all h_i cells from i-th row. Then, we will try to take c_j cells for all rows $j < i$, where $c_j \le c_{j+1}$ and $c_j \le h_j$. We could compute those c values in a greedy manner, by going from i to 1 and keeping the current value c_j. But this would result in $O(n^2 m)$ solution, which is still quite slow.

Instead, let's try to re-use the previous values of dp_j. Let's find the largest t, such that $h_t < h_i$. For all rows in a range $[t+1, i]$, all values c_j will be equal to h_i. And that range will exactly form a rectangle, so we can take its value in $O(1)$ with 2D prefix sums.

The range $[1, t]$ will be exactly the same as in dp_t, so we can conclude $dp_i = dp_t + sum(t, i)$, where sum is the rectangle sum mentioned above.



Calculating dp_i.

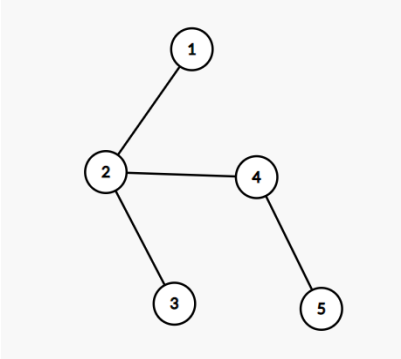
We can find such t for all i in a lot of ways. The cleanest way is to maintain a monotone stack. Other ways include binary searching or keeping a set instead, but that would add an extra $O(\log n)$ factor to the complexity.

To sum up, we get a pretty nice solution in $O(nm)$ time and memory complexity.

Problem B. Two tree

- **For the** 1 **subtask**, you can calculate subtree sizes using DFS, and count the answer.
- **For the** 2 **subtask**, note that answer for node x, which is a neighbor of 1 will not change answer dramatically because after rerooting the tree on x only sizes of subtree 1 and x nodes changes. Let us calculate the answer naively for a 1 node and have a pointer for each tree on that node. We can create queries like for node v, answer will be found after moving both pointers to node v and comparing the changed nodes values. To make it optimal, we can use MO algorithm.
- **For the** 4 **subtask**, note that the answer for v is same as answer for some node u, except for the nodes in the path between. Since the tree in this subtask is complete binary tree, the distance between two nodes are small, so you can iterate over it and compute the answer.
- **Now for the subtask** 3, let us solve the problem for x in which v $sub_1(x) > sub_2(x)$ will be satisfied. $sub_1(x)$ can have $deg_1(x) + 1$ different values and $sub_2(x)$ $deg_2(x) + 1$ different values, where $deg_1(x)$ is number of neighbors in the first tree, and $deg_2(x)$ is number of neighbors in the second tree.

  Let us conder the following tree with 1 node as root:



  Subtree sizes are $sub_1 = 5$, $sub_2 = 4$, $sub_3 = 1$, $sub_4 = 2$, $sub_5 = 1$. So when we consider edge $2, 4$ (so root v will be in this case either 4 or 5) and the size 2 will be 3 since $n - sub_4$. In the same way, we can calculate for an edge from 2 to other sons. Still, for this node we also have an edge from parent 1, in that case, our component size is sub_2 and another component size is $n - sub_2$.

  If x is root of trees, then the $sub_1(x)$ and $sub_2(x)$ both equal to n. This case never will be $sub_1(x) > sub_2(x)$.

  Let us consider if our parent in the first tree is a, b in the second tree. So sub_1 will be equal to the size of the component with x when we delete edge a, x from the first tree, similarly $sub_2(x)$ is the equal size of the component with x when we delete edge b, x from the second tree. So if $sub_1(x) > sub_2(x)$, then it holds for every v which is inside both in the same component with a (after deleting edge a, x from the first tree) and in the same component with b (after deleting edge b, x from the second tree).

  You can use the Euler tour to check v is inside a component. Note that every component is like a segment in the Euler tour.

  Let us consider node v as a point in the $2D$ axis —- tin_v in the first tree as the x-axis and tin_v in the second tree as the y-axis. You can naively iterate over every pair of neighbors (one from the first tree, the other from the second tree), and build adding in subrectangle queries, which can be solved offline in $n \cdot log(n)$ time. But still we have $deg_1(v) \cdot deg_2(v)$ which is fine for 1, 3 and 4 subtaks.
- **Full solution.** To optimize that part we will use two pointers. So let is sort neighbors of every node by the size of subtrees in both trees separately, and let is root both tree at 1 node. We do not need to naively iterate over every pair of neighbors, instead let us consider neighbor a, and our component size will be equal to $n - sub_1(a)$, and these values are decreasing as *subtree* of neighbors are sorted, and same is with second tree where $n - sub_2(b)$. We can use two pointers to find first

such b so $n - sub_1(a) > n - sub_2(b)$ for a, and now all the subtree of b and subtrees of neighbors after it will hold. Note that you need to consider parent of node x in the first tree, and parent in the second tree separately with each other, and with sons of node x.

Problem C. Golf

We define $v_a = 1$, $v_b = 2$ and $v_c = \text{s}$, first, second terminal vertex and root vertex respectively.

1. $n = 100$ $a + b = 4$. 10 points.

   Solve it by hand.

2. $n = 100$ $a + b = 32$. 10 points.

   Build a complete binary tree with 32 leaves. Point first a leaves to the v_a and remaining b to v_b.

   Note, all leaves have equal probability.

3. $n = 50$ $a + b = 2^{30}$. 10 points.

   (a) If $a = b$. Point edges from v_c to v_a and v_b.

   (b) If a and b is even. Just divide a and b by two. Probability doesn't change.

   (c) If $a \leq b$, a and b are odd.
   Create a new node u and point edges from v_c to v_b and u. Now we can recursively build a graph with a, $\frac{b-a}{2}$ starting from u.
   Note $a + \frac{b-a}{2} = \frac{a+b}{2}$ still power of two.

   (d) If $a \geq b$, a and b are odd. Similar to the previous case.

4. $n = 33$ $a, b \leq 15$. 10 points.

   Generate a random graph and compare probability by simulating random walks.

5. $n = 64$ 10 points

   Find smallest k s.t. $2^k \geq a + b$.

   Build a complete binary tree with $2^k \geq a + b$ leaves.

   Point first a leaves to the v_a and next b to v_b and remaining to v_c.

   If both ends of the node point to the same node then we can delete this node and point all incoming edges to the next node.

   Since all leaves pointing to v_a on continuous interval we can compress them to $2 * k$. Similar to the segment tree.

   Same for v_b and v_c. At the end, we will use $2 * k$ nodes

   Note that if some end of the interval is equal to the end of all tree it will not create new nodes. And the right end of v_a and the left end v_b are the same and will reuse nodes. Same for v_b and v_c.

6. $n = 50$, 10 points

   Lets try to build segments optimally.

   $a + b + c = 2^k$. Here c number of nodes pointing to the root.

   Node at the depth d corresponds to 2^{k-d} leaves.

   We start at the root from depth 1.

   If $a \geq 2^{k-1}$ point first edge to v_a and solve from second child with $a - 2^{k-1}, b, c$ and $k - 1$.

   If $2^{k-2} \leq a, b \leq 2^{k-1}$ point first child to v_a and v_b and solve second child with $a - 2^{k-2}, b - 2^{k-2}, c$ and $k - 1$.

Other cases are similar or.

We create new nodes only in the second case. It can be proven number will be less than $\frac{2}{3}k$.

In the end, we will use $\frac{5}{3} * k$ nodes.

7. $n = 36$, 10 points

   Note that the nodes created in the second case point to two nodes from v_a, v_b, v_c. There are only 3 choices. We can create them before and reuse them.

   In the end, we will use $k + 3$ nodes.

8. $n = 35$, 10 points

   Note that the last node also points to two nodes from v_a, v_b, v_c. We can also reuse it.

   In the end, we will use $k + 2$ nodes.

9. $n = 34$, 10 points

   Its easy to see that this subtask is useless.

   In the end, we will use $k + 1$ nodes.

10. $n = 33$, 10 points

    Suppose we have a graph that solves this problem for a, b, c where $a + b + c = 2^k$.

    Lets look to next operation:

    (a) Create new node u.
    (b) Add two edges from v_a to v_b and u. Mark u as the first terminal.

    In new graph we get $a, a + 2 * b, 2 * c$ where $a + b + c = 2^{k+1}$.

    Reverse of this operation looks like $a, b, c \to a, (b - a)/2, c/2$.

    To apply the operation we need $b \geq a$, b and a the same parity, even c.

    We have 6 options for such an operation. Since $a + b + c$ is even, at least one of them can be used.

    In the end, we will use k nodes.

Problem D. Atoms

Let's call $[Da]$ atoms as *red points*, $[Bs]$ atoms as *white points* and $[Km]$ atoms as *black points*.

- $n = 3$. **9 points.**

  Any brute force solution should work. You can even try to write all cases by hand.

- $d_1 = d_2 = \cdots = d_n$. **8 points.**

  All red points have the same coordinate. Find all white points that are to the left of the red points and denote their count as x. Similarly, find all black points to the right and denote their count as y. We can form at most $k = min(x, y)$ triples, so we should just take k red points with highest energy output.

- $k_i \geq b_j$ **and** $k_i \geq d_j$ **for each** $1 \leq i, j \leq n$. **11 points.**

  All *black points* are located to the right of all other points. It means we can choose any of them for any triple we might form, so we can just ignore all black points and focus on the pairs of white and red instead.

  The following greedy will work here. We will iterate over all white points in order of decreasing coordinate. Suppose we are now considering a white point at coordinate x. Out of all red points to the right of x, we will choose the one which has the highest energy output and delete it.

Why does it work? We maintain a set of red points. As we go from right to left, some new red points get added to the set or we meet a white point that can be used to form a new pair. Since any red point that has been added to the set will always be available later on, it is clear that picking the local optimum works.

- $c_1 = c_2 = \cdots = c_n = 1$**. 11 points.**

  All red points have an energy output of 1, which means we just need to select the largest amount of them. Here we have a different kind of greedy. Let's go through all red points from left to right. We will try to form a triple which involves the current red point.

  If our point can be paired with several white points, we can choose any of them since it will not affect anything. If our point can be paired with several black points, we should choose the closest one, because that black point is the first one to become unavailable later on. If we can form pairs on both sides, then we form a triple and erase all three points.

- $n \leq 300$**. 12 points.**

  Let's suppose in the optimal answer we will end up taking k red points. Each red point will need a white point from the left and a black point from the right. So it is always optimal to select only the **leftmost** white points and only the **rightmost** black points.

  So essentially we have selected some white points ($L_1 \leq \ldots \leq L_k$) and some black points ($R_1 \leq \ldots \leq R_k$). Now we need to select k red points ($M_1, \ldots, M_k$) with the highest total energy output. Notice that for some pair of red points $M_i \leq M_j$, conditions $L_i \leq L_j$ and $R_i \leq R_j$ should also hold.

  That means we can form pairs (L_i, R_i) and solve the problem of choosing k red points, such that each point is inside its corresponding segment and their total energy output is largest possible.

  This can be done by maintaining $dp_{i,j}$ — the largest answer if we formed first j segments and considered first i red points. The transitions are trivial — either we put the i-th point into j-th segment, or we don't. This way we get a solution with $O(n^3)$ time complexity.

- $n \leq 2000$**. 12 points.**

  Clearly, we can do better than $O(n^2)$ for a fixed k, right? Indeed, there exists a nice greedy solution.

  This time, we will go through all black points in the order of increasing coordinates. Suppose our current black point is located at x. Out of all red points to the left of x, we want to choose the one that has a white point to the left of it and has the highest energy output. If there is such a red point, we will take it. Additionally, over all white points to the left of it, we will choose the closest one.

  First, how to implement it fast? Since we go over all black point in the order of increasing x, we will have to deal with following updates:

  1. **Add a new red point with coordinate x and energy output y.** We will maintain all red points in a set in the order of decreasing y.
  2. **Add a new white point with coordinate x.** We will maintain white point in another set in order of increasing x.
  3. **Check if we can form a triple.** Let's take the red point with highest y. If it is located to the left of the leftmost white point, we cannot take it into a triple anymore and we will never be able to. So we can delete it from the set and repeat again. Otherwise, we found the optimal red point. Now we find the closest white point to the left of it and erase it.

  Since each operation takes $O(\log n)$ time and we make $O(n)$ operations amortized, we get the time complexity of $O(n \log n)$ for a fixed k, giving us $O(n^2 \log n)$ in total.

  Now let's get to the proof. Again, since we are iterating in the order of increasing x, if at any time a new red point gets added to our set, it will always be available later on. So, similar to *subtask 3*,

our current choice of a red point will not mess up anything in the future, so we should greedily pick the best one. And once we found the red point, it is obvious why we need to pair it with the closest white point.

- **No additional constraints. 37 points.**

  Now, let $f(k)$ denote the result of the greedy above for a fixed k. Clearly, $f(k)$ will be undefined for some $k > c$. If that is the case, we will consider $f(k)$ to be equal to $-\infty$.

  We would like to find the maximum value of $f(k)$. Now, we get to the very interesting part.

  Lemma. Value of $f(k) - f(k-1)$ is non-increasing. That is, $f(k)$ first increases up to some point t, then decreases afterwards.

  Proof. The formal proof follows from the *Min-Cost-Max-Flow* algorithm. We can represent our problem as a network, where the first layer contains all white points, second layer contains all red points and third layer contains all black points. All edges will have unit capacities. To ensure each red point is taken only once, we will duplicate it into two vertices u and v, connect u with all white points, connect v with all black points and connect u and v with a single edge of unit capacity and cost of $-c_i$, where c_i is the energy output of the current red point.

  It is clear that the answer we are looking for will be the value of *MCMF* on this network, taken with an opposite sign. So if we consider how *MCMF* actually works, we will notice that it starts with flow $k = 0$ and tries to expand to flow $k + 1$ by finding a shortest path in the network that can be saturated. The *Min-Cost-Max-Flow* variation keeps increasing k until its no longer possible, so we are actually interested in the *Min-Cost-Flow* variation where we don't care about the exact value of k, we care only about minimizing the cost. And in that variation, the algorithm stops right at the exact value of k which maximizes $f(k)$. Hence, we consider it proven.

  Knowing this lemma, now we can find the optimal k using a ternary search. You could implement it in a binary search style, comparing values of $f(m)$ and $f(m+1)$, but a traditional ternary search also passes.

  In conclusion, we get a solution that has a $O(n \log^2 n)$ or $O(n \log n \log_{1.5} n)$ time complexity.

Problem E. Tree game

- $u_i = 1, v_i = i + 1$, $r_i = b_i = 1$. **9 points.**

  The tree forms a star with all vertices connected to 1. If the first player starts at 1, the score will be $(n-1) : 1$. Otherwise, the first player will move to 1, claiming the vertex of the other player, so the score will be $n : 0$.

- $u_i = i, v_i = i + 1$, $r_i = b_i = 1$. **13 points.**

  The tree forms a chain $1 - 2 - \ldots - n$. Let a be the vertex of first player and b be the vertex of second player. WLOG $a \leq b$.

  If the distance between them is 1, the score for each player will be the number of nodes on his side of the tree $(a : n - b + 1)$.

  If the distance between them is 2, first player will move in the middle and win with a score $n : 0$.

  If the distance is more than 2, the case gets more interesting. It can be shown that both of them will move towards each other until the distance gets down to 3. And from that point onwards, it is never optimal for them to move closer, because the distance becomes 2 and the other player will win in the middle. So both of them will be making any other moves until one of them has no more moves. That player will be forced to move to the middle, and the score will be easy to calculate from there.

  It requires a bit of casework, considering both the parity of the distance and the number of extra moves both players have.

- $n, q \leq 10$. 11 **points.**

  Here any correctly implemented bruteforce simulation should work.

- $q = 1$, $r_i = b_i = 1$. 14 **points.**

  Again, as in the subtask 2, let's consider the distance between the vertices of players. The strategy remains exactly the same, except this time you will have to do it on a tree. Since $q = 1$, you can implement this naively in $O(n)$.

- $r_i = b_i = 1$. 20 **points.**

  Same as the subtask above, but you will have to calculate the following things fast:

  1. The distance between two vertices a and b.
  2. The k-th vertex on the path between a and b.
  3. The number of vertices in a subtree of v if the root of the tree will be u.

  All of these are quite classical things that often appear in tasks on trees and each one can be implemented in $O(\log n)$ or $O(1)$.

- $q = 1$. 17 **points.**

  Here we are given an ongoing game so we have more cases to consider.

  If there is at least one edge e such that its both endpoints are colored in the opposite colors, it means that no move in the future will recolor any vertex. So we can delete this edge and solve for the remaining trees separately. In fast, in can the proven that in each of the remaining we will have at most one color, and that color will be the corresponding endpoint of e. So the answer can be calculated with a simple DFS.

  If there is no such edge, it means we have two connected components of vertices R and B. Let's find the pair of vertices (a, b) such that $a \in R$, $b \in B$ and $dist(a, b)$ is the least possible. We can find that pair as follows. Take any vertex $a' \in R$, find the closest vertex $b \in B$ to it. Similarly, take any vertex $b' \in B$, find the closest vertex $a \in B$ to it. The desired pair is the (a, b) we found in this process.

  After we found the pair (a, b), this game state can be seen equivalent to the state where players **start** with vertices (a, b) with just some other vertices already pre-colored for them. So the solution from now onwards will be exactly the same as in the subtask above.

- **No other constraints.** 16 **points.**

  Once again, we will have to implement the same solution in an efficient way. Finding an edge e can be done in $O(|R| + |B|)$ as follows. Let's assume the tree is rooted at an arbitrary node. Then each edge e will connect some vertex v with its parent p_v. So, it is enough just to consider all pairs $(a \in A, p_a)$ and $(b \in B, p_b)$ to find those edges.

  If we found such edges e, now we will have to calculate the sizes of subtrees. We can already solve that fast because it is derived from this task: *The number of vertices in a subtree of v if the root of the tree will be u*. So the whole thing here works in $O(|R| + |B|)$ or $O((|R| + |B|) \log n)$.

  If there are no such edges e, we will need to find (a, b) in the way described above. The naive implementation already works in $O((|R| + |B|) \log n)$ time complexity. And since after this the solution comes down to the case $r_i = b_i = 1$, the whole problem is solved.

  Overall, we get time complexity $O(n \log n + (|R| + |B|) \log n)$ with some minor differences depending on the implementation of LCA and other algorithms.

Problem F. Researchers

- $n, m, q \leq 100$, $d_i \leq 100$, $r_i \leq 100$. **5 points.**

  Build the graph for all years from 1 to 100, and for each query check the reachability with a simple DFS. The time complexity is $O(max\{r_i\}(n+m)q)$.

- $n, m, q \leq 3000$, $d_i \leq 3000$, $r_i \leq 3000$. **7 points.**

  Similarly, build the graph for all years from 1 to 3000. Color all components with a DFS. Let c_v be the color of vertex v. Now, when you go through all queries, you can check reachability by simply checking if $c_{x_i} = c_{y_i}$.

  The time complexity is $O(max\{r_i\}(n+m+q))$.

- $m = n-1$, $a_i = i$, $b_i = i+1$. **12 points.**

  The graph is a chain $1-2-\ldots-n$. For the path $u-v$ to exist ($u < v$), all edges between them should be present. So it boils down to calculating the intersection of segments in a range $[u, v-1]$, which can be done with a segment tree. You will need to find $max\{l_i\}$ on a range and $min\{r_i\}$ on a range.

  The time complexity is $O(n + q \log n)$.

- $d_i = 10^9$. **16 points.**

  $d_i = 10^9$ essentially means that the edges will never get deleted. So let's sort all edges in the order of their addition time. Now, for each query (u, v) we can try to binary search the first time t when vertices u and v will be connected. The easiest and most popular way to do it is *Parallel binary search*.

  The time complexity is $O((n+m+q)\log 10^9)$.

- $l_i = r_i$. **12 points.**

  Let's convert the problem to the following. You have to process 3 kinds of events in the chronological order:

  1. Add an edge (u, v).
  2. Remove an edge (u, v).
  3. Query if vertices u and v are connected.

  This is a very classical problem called *Dynamic Connectivity*. In this particular case, we can do it in offline using *divide and conquer* with DSU with rollbacks.

  $O((n+m+q)\log^2 n)$.

- $n, m, q \leq 40000$. **27 points.**

  Let's create $2m$ events of two types in chronological order:

  1. Add an edge (u, v).
  2. Remove an edge (u, v).

  We will do *Square root decomposition on the events*.

  Each query corresponds to some range (l, r) of events. Let's enumerate over all blocks of events. For each block, there will be some queries which cover it completely and there will be some queries which cover it only partially.

  Also, inside a block of size k, there will be at most $2k$ relevant vertices and edges that will be affected. So let's compress the whole graph and get rid of useless vertices and edges.

  Now, we can easily deal with partial queries. Exactly the same as we did in subtask 2, let's deal with the events in a straightforward way one by one and color all the components using a DFS.

Then we will go over all partial queries, and if a query covers the current event, we will update the answer to it accordingly.

This part will work in $O(k^2)$ per block. And a query will be considered partial in at most two blocks, so the query will contribute only $O(k)$ to the overall time complexity.

Now, let's deal with queries that cover the block completely. There could be potentially many of them per each block. But here is an important thing: after we compressed the graph we have only $O(k)$ vertices, so we have only $O(k^2)$ pairs of vertices we care about. So if we maintain a two-dimensional array $ans_{u,v}$ for each pair (u, v) and calculate it the same way as we did with partial queries, we will get $O(k^3)$ time complexity per block.

Now, let's think about the optimal k. We will have n/k blocks overall, in each of them we will have to do about $O(n+m+q)$ to compress the graph and look for the relevant queries. Inside the block, we will do $O(k^3)$ operations. So if we try to make the two sides equal, we get that the optimal k should be around $(n+m+q)^{1/3}$.

So the total time complexity we get is $O(n+m+q)^{5/3}$, which should be good enough to pass this subtask.

- **No additional constraints.** 21 **points.**

  In fact, the solution from the previous subtask could be optimized to the degree that it will even pass this subtask. But here, we will discuss a better solution.

  The bottleneck of our solution was the $O(k^3)$ part, where we calculate $ans_{u,v}$ for all pairs (u, v) in a block. Can we do it faster? The answer is fortunately yes.

  Let's notice that we essentially have the same original problem, but this time $n = O(k)$, $m = O(k)$ and $q = O(k^2)$ with all queries being on the whole range as well.

  So can we apply the same solution to that sub-problem? Take a block $t = \sqrt{k}$, and solve for blocks of size t again? Theoretically, we can, but it will probably end up slower than the $O(k^3)$ discussed above.

  But instead, we can analyze this more and figure out that for this sub-problem the optimal block size t will be $n/2$. In other words, it is optimal to solve this sub-problem as a divide and conquer algorithm. Split all events right in the middle, compress all vertices and edges on both sides, solve recursively and combine the answers.

  Let's analyze the time complexity. We have $T(n) = 2T(n/2) + O(n^2)$. If we expand, we will get $T(n) = O(n^2 + 2 \times (n/2)^2 + 4 \times (n/4)^2 + \ldots) = O(n^2 \times (1 + 1/2 + 1/4 + \ldots) = O(n^2)$. So for $n = k$ we will be able to solve the problem in $O(k^2)$, which is exactly what we were looking for.

  Taking k to be approximately $O(\sqrt{n+m+q})$, we end up with a $O((n+m+q)\sqrt{n+m+q})$ solution, although with a very high constant.

Problem A. Төбешік

- $n = 1$. 12 **ұпай.**

  Барлық мүмкін болатын (L_1, R_1) жұптарын санап, ең жақсысын таңдаңыз. Оны $O(m^2)$ немесе $O(m)$ уақытында жасауға болады.

- **Бұғатталған ұяшықтар жоқ.** 7 **ұпай.**

  Бүкіл үстелді төбешік ретінде таңдау әрқашан оңтайлы. Сондықтан жауап барлық мәндердің қосындысы ғана.

- $n, m \leq 50$. 25 **ұпай.**

  $dp_{i,l,r}$ — төбешік i жолында (l, r) сегментімен аяқталса, біз әлі қол жеткізе алатын төбешіктің ең жақсы сомасын сақтайық.

  Оны есептеу үшін $l \leq l_p \leq r_p \leq r$ болатындай барлық (l_p, r_p) жұптарын санап, dp_{i-1,l_p,r_p} кеңейтіп көріңіз. Осы қатарда жаңа төбені бастаған кездегі жағдайды да қарастырыңыз.

  Жалпы уақыт күрделілігі $O(nm^4)$ болады.

- $n, m \leq 300$. 22 **ұпай.**

  Дәл жоғарыдағыдай шешім, бірақ бұл жолы (l_p, r_p) санаудың орнына, біз $l \leq l_p \leq r_p \leq r$ бойынша максимум тіктөртбұрышты іздеп жатқанымызды байқаймыз. Мұны басқа DP арқылы сақтауға болады, айталық $best_{i,l,r}$. Өту үшін $dp_{i,l,r}$, $best_{i,l+1,r}$ немесе $best_{i,l,r-1}$ қарастырасыз.

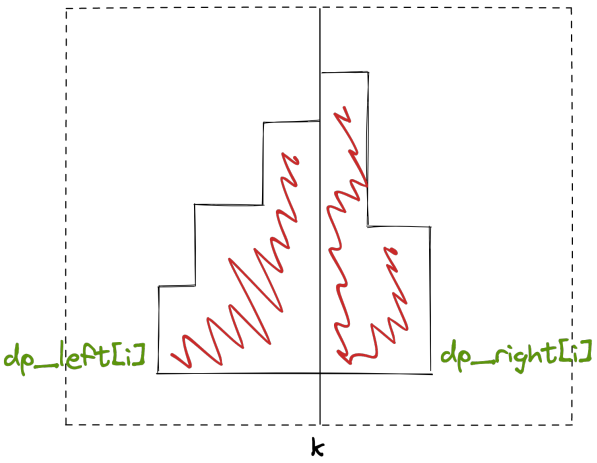
  Егер мұқият орындалса, жалпы уақыт күрделілігі қазір $O(nm^2)$ құрайды.

- **Басқа шектеулер жоқ.** 34 **ұпай.**

  Төбешік, оның қазіргі анықтамасы бойынша, тікелей есептеу өте қиын. Олай болса, оның орнына қарапайым етіп жасауға тырысайық.

  Төбешіктің қасиеттеріне қарай отырып, біз оның бір бағанға дейін өсетінін, содан кейін төмендейтінін байқаймыз. k бағанасын санап, төбешікті сол және оң жақ бөліктерге бөлейік. Сол жақ өсу керек, ал оң жақ төмендеу керек. Біз олардың толығымен симметриялы екенін анықтаймыз, сондықтан тек сол жақ бөлігін есептеуге назар аударайық.

  dp_i — төбешіктің мүмкін болатын ең жақсы сомасын анықтайық, егер оның **төменгі** i жолынан басталса (балама түрде $e = i$) және оның төменгі сол жақ бөлігі бағанның сол жағында бір жерде басталса k. Сол сияқты, біз оң жақта осындай dp анықтаймыз және нәтижелерді біріктіруге тырысамыз.

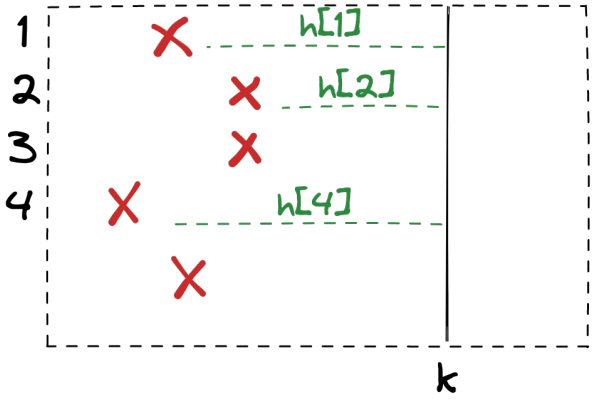


Горка объединенная в столбце k.

Сол жақта да, оң жақта да $dp_1, \ldots, dp_n$ мәндерін есептегеннен кейін, біз оларды e-ні 1-ден n-ге дейін қайталау және максималды мүмкін соманы $(dpLeft_e + dpRight_e)$ алу арқылы біріктіре алатынымыз анық.

Егер біз $O(n \log n)$ немесе $O(n)$ уақытында dp-ды қандай да бір жолмен есептей алатын болсақ, мәселе шешіледі.

dp_1 есептеуге назар аударайық. Біз бұғатталған ұяшыққа тигенше, мүмкіндігінше ұзын жолды алуға тырысуымыз керек. h_1 біз алатын ұяшықтардың максималды санын белгілейік. Сол сияқты біз $h_2, \ldots, h_n$ анықтаймыз. Олардың әрқайсысын $O(1)$ ішінен табуға болады, егер сіз әрбір (i, j) ұяшығы үшін сол және оң жақтағы ең жақын блокталған ұяшықты алдын ала есептесеңіз.

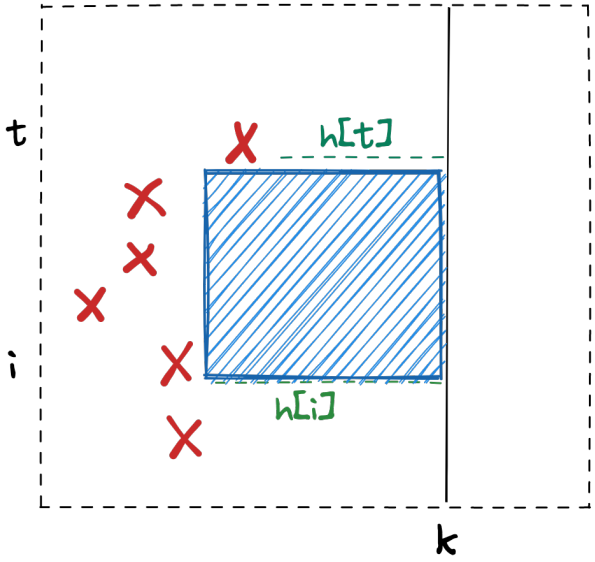


Как определяются значения h_i.

Енді dp_i мәнін қалай есептеу керектігі анық болуы керек. Алдымен i-ші жолдан барлық h_i ұяшықтарын алудан бастаймыз. Содан кейін, біз $c_j \leq c_{j+1}$ және $c_j \leq h_j$ болатын барлық $j < i$ жолдары үшін c_j ұяшықтарын алуға тырысамыз. Біз i-дан 1-ға дейін және ағымдағы c_j мәнін сақтай отырып, бұл c мәндерін ашкөздікпен есептей аламыз. Бірақ бұл $O(n^2 m)$ шешіміне әкеледі, бұл әлі де өте баяу.

Оның орнына dp_j алдыңғы мәндерін қайта пайдаланып көрейік. $h_t < h_i$ болатын ең үлкен t табайық. $[t+1, i]$ ауқымындағы барлық жолдар үшін c_j барлық мәндері h_i мәніне тең болады. Және бұл диапазон дәл тіктөртбұрышты құрайды, сондықтан оның мәнін $O(1)$ түрінде 2D префикс қосындыларымен қабылдай аламыз.

$[1, t]$ диапазоны dp_t-дағымен бірдей болады, сондықтан $dp_i = dp_t + sum(t, i)$ деген қорытынды жасауға болады, мұнда sum жоғарыда айтылған тіктөртбұрыш қосындысы.



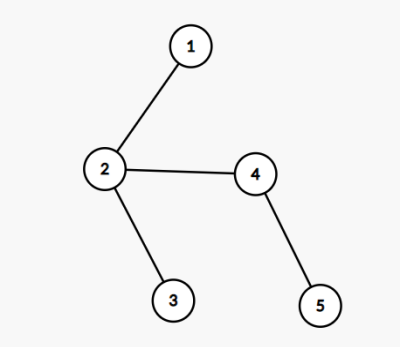
Calculating dp_i.

Біз мұндай t-ды барлық i үшін көптеген жолдармен таба аламыз. Ең таза әдіс - монотонды стекті сақтау. Басқа жолдар екілік іздеуді немесе оның орнына жиынды сақтауды қамтиды, бірақ бұл күрделілікке қосымша $O(\log n)$ факторын қосады.

Қорытындылай келе, біз $O(nm)$ уақыт пен жадтың күрделілігінде өте жақсы шешім аламыз.

Problem B. Екі ағаш

- **1-ші ішкі есеп**. DFS арқылы ішкі дарақтардың өлшемдерін тауып, жауапты есептеуге болады.
- **2-ші ішкі есеп**. 1 төбесінің көршісі x төбесі үшін жауап қатты өзгермейтінін байқайық, себебі, ағашты x төбесіне ілген соң, тек 1 және x төбелерінің ішкі дарақтарының өлшемдері өзгереді. Жауапты 1 төбесі үшіе есептеп, әр ағаштан сол төбеге көрсеткіш қоюға болады. Біз әрбір v төбесі үшін сұраулар жасай аламыз. Жауап екі көрсеткішті де v төбесіне жылжытқаннан кейін және өзгертілген төбелерді салыстырғаннан кейін табылады. Оны оңтайлы ету үшін біз MO алгоритмін қолдана аламыз.
- **4-ші ішкі есеп**. v төбесінің жауабы u төбесінің жауабына тең болады, егер олардың жолындағы шыңдарды санамасақ. Бұл ішкі есептегі ағаш толық екілік ағаш болғандықтан, екі төбенің арақашықтығы аз, сондықтан сіз жолда жүріп, жауапты есептей аласыз.
- **3-ші ішкі есеп**, x төбесі қандай есеп шешейік – қандай v төбелерінде $sub_1(x) > sub_2(x)$ шарты орындалады. $sub_1(x)$ $deg_1(x)+1$ түрлі мәнге, ал $sub_2(x)$ $deg_2(x)+1$ түрлі мәнге ие бола алады, мұнда $deg_1(x)$ — бірінші ағаштағы көршілер саны, $deg_2(x)$ — екінші ағаштағы көршілер саны.



Келесі ағашты түбірі 1 төбесі болатындей етіп қарастырайық:

Ішкі дарақтардың өлшемдері: $sub_1 = 5$, $sub_2 = 4$, $sub_3 = 1$, $sub_4 = 2$, $sub_5 = 1$. $2, 4$ қырын қарастырайық (демек v түбірі 4-ші немесе 5-ші төбе болады) және 2-ші ішкі дарақтың өлшемі 3 болады, яғни $n - sub_4$. Дәл осылай біз 2-ші ішкі дарақтың өлшемін қалған перзенттер үшін таба аламыз. Дегенмен, бізде тағы 1 әкеден қыр бар, бұл жағдайда біздің компонентіміздің өлшемі $sub - 2$-ге тең, ал басқа компоненттің өлшемі $n - sub_2$.

Егер x ағаштың түбірі болса, онда $sub_1(x)$ мен $sub_2(x)$ n-ға тең. Бұл жағдайда $sub_1(x) > sub_2(x)$ шарты ешқашан орындалмайды.

Тағы да біздің бірінші ағаштағы әкеміз a және екінші ағаштағы әкеміз b болған жағдайды қарастырайық. Сонда біз a, x қырын бірінші ағаштан жойған кезде, sub_1 мәні x төбесінің компонентінің өлшеміне тең болады. Дәл солай, b, x қырын бірінші ағаштан жойған кезде, sub_2 мәні x төбесінің компонентінің өлшеміне тең болады. Егер $sub_1(x) > sub_2(x)$, демек ол a-мен (a, x қырын бірінші ағаштан жойған соң) де, b-мен (b, x қырын бірінші ағаштан жойған соң) де бір компонентте жатқан әрбір v төбесі үшін орындалады.

v төбесі компонент ішінде екенін тексеру үшін Эйлер айналма алгоритмын қолдануға болады. Назар аударыңыз, әрбір компонент Эйлер айналма жолындағы ішкі кесінділер болып табылады.

v төбесін $2D$ кеңістігіндегі нүкте ретінде қарастырайық — бірінші ағаштағы tin_v x осі ретінде және екінші ағаштағы tin_v y осі ретінде. Әрбір көршілер жұбын сұрыптап, ішкі тіктөртбұрыш үшін қосу сұраулары істесек болады. Бұл есепті $n \cdot log(n)$ уақытында шешуге боады. Бірақ бізде $deg_1(v) \cdot deg_2(v)$ төбелер жұбы бар, сондықтан бұл шешім тек 1, 3 және 4-ші ішкі есептерді өтеді.

- **Толық шешім.** Есептің бұл бөлігін оңтайландыру үшін екі көрсеткіш техникасын қолдансақ болады. Сонымен, әрбір төбенің көршілері ішкі дарақтардың өлшемдері бойынша екі ағашта бөлек сұрыпталған болсын, және 1-ші төбе түбір болсын. Бізге әрбір көршілер жұбін қарастыру қажет емес, оның орнына a көршісін алайық, және біздің компоненттің өлшемі $n - sub_1(a)$ болады, және бұл мәндер көршілердің $subtree$ өскеніндей кемиді. Дәл солай екінші ағаш үшін, мұнда $n - sub_2(b)$. Біз $n - sub_1(a) > n - sub_2(b)$ болатындай a төбесі үшін ең бірінші b төбесін табу үшін екі көрсеткішті қолдана аламыз, сонда b толық ішкі дарағы үшін және көршілершің ішкі дарақтары үшін бұл шарт орындалады. x төбесінің әкелерін және x-тін перзенттерін екі ағаш үшін бөлек қарастыру керектігіне назар аударыңыз.

Problem C. Гольф

$v_a = 1$, $v_b = 2$ және $v_c =$ s сәйкесінше бірінші, екінші терминал шыңы және түбір шыңын анықтаймыз.

1. $n = 100$ $a + b = 4$. 10 ұпай.

   Қағазда шешіңіз.

2. $n = 100$ $a + b = 32$. 10 ұпай.

   32 жапырағы бар толық бинарлық ағаш құрастырамыз. Бірінші a жапырақтан v_a және қалған b жапыратқтан v_b қырларын жүргіземіз.

   Назар аударыңыз, барлық жапырақтардың ықтималдығы бірдей.

3. $n = 50$ $a + b = 2^{30}$. 10 ұпай.

   (a) Егер $a = b$. v_c мен v_a және v_b аралығындағы нүкте шеттері.

   (b) a және b жұп болса. a және b екіге бөліңіз. Ықтималдық өзгермейді.

   (c) $a \le b$, a және b тақ болса.
   Жаңа u түйінін және v_c бастап v_b және u нүктелерінің шеттерін жасаңыз. Енді u бастап a, $\frac{b-a}{2}$ болатын графикті рекурсивті түрде құра аламыз.
   $a + \frac{b-a}{2} = \frac{a+b}{2}$ әлі екінің дәрежесін ескеріңіз.

(d) $a \geq b$, a және b тақ болса. Алдыңғы жағдайға ұқсас.

4. $n = 33$ $a, b \leq 15$. 10 ұпай.

   Дәл екінші бөлімдегідей жасаймыз. Қалған жапырақтардан түбірге қыр қосып қоямыз.

5. $n = 64$ 10 ұпай

   Ең кіші k ст. табыңыз. $2^k \geq a + b$.

   $2^k \geq a + b$ жапырақтары бар толық екілік ағашты құрастырыңыз.

   Бірінші a жапырақтарын v_a және келесі b v_b және қалғанын v_c жағына көрсетіңіз.

   Егер түйіннің екі ұшы да бір түйінді көрсетсе, біз бұл түйінді жойып, барлық кіріс жиектерін келесі түйінге бағыттай аламыз.

   Үздіксіз интервалда барлық жапырақтар v_a белгісін көрсететіндіктен, біз оларды $2 * k$-ға дейін қыса аламыз. Сегмент ағашына ұқсас.

   v_b және v_c үшін бірдей. Соңында біз $2 * k$ түйіндерін қолданамыз

   Егер интервалдың кейбір соңы барлық ағаштың соңына тең болса, ол жаңа түйіндерді жасамайтынын ескеріңіз. Ал v_a оң жақ шеті мен сол жақ шеті v_b бірдей және түйіндерді қайта пайдаланады. v_b және үшін бірдей v_c.

6. $n = 50$, 10 ұпай

   Сегменттерді оңтайлы түрде құруға тырысайық.

   $a + b + c = 2^k$. Мұнда түбірді көрсететін c түйіндер саны.

   d тереңдіктегі түйін 2^{k-d} жапырақтарына сәйкес келеді.

   Біз 1 тереңдігінен түбірден бастаймыз.

   Егер $a \geq 2^{k-1}$ бірінші жиегін v_a нүктесіне көрсетіп, екінші баладан $a - 2^{k-1}, b, c$ және $k - 1$ арқылы шешіңіз.

   $2^{k-2} \leq a, b \leq 2^{k-1}$ бірінші баланы v_a және v_b көрсетсе және екінші баланы $a - 2^{k-2}, b - 2^{k-2}, c$ және $k - 1$.

   Басқа жағдайлар ұқсас немесе.

   Біз жаңа түйіндерді тек екінші жағдайда ғана жасаймыз. Сан $\frac{2}{3}k$-нан аз болатынын дәлелдеуге болады.

   Соңында біз $\frac{5}{3} * k$ түйіндерін қолданамыз.

7. $n = 36$, 10 ұпай

   Екінші жағдайда жасалған түйіндер v_a, v_b, v_c екі түйінді көрсететінін ескеріңіз. Тек 3 таңдау бар. Біз оларды бұрын жасап, қайта пайдалана аламыз.

   Соңында біз $k + 3$ түйіндерін қолданамыз.

8. $n = 35$, 10 ұпай

   Соңғы түйін v_a, v_b, v_c екі түйінді де көрсететінін ескеріңіз. Біз оны қайта пайдалана аламыз.

   Соңында біз $k + 2$ түйіндерін қолданамыз.

9. $n = 34$, 10 ұпай

   Бұл қосалқы тапсырманың пайдасыз екенін түсіну оңай.

   Соңында біз $k + 1$ түйіндерін қолданамыз.

10. $n = 33$, 10 ұпай

    Бізде a, b, c үшін осы есепті шешетін граф бар делік, мұнда $a + b + c = 2^k$.

    Келесі операцияны қарастырайық:

(a) Жаңа u түйінін жасаңыз.

(b) v_a бастап v_b және u аралығындағы екі жиекті қосыңыз. u бірінші терминал ретінде белгілеңіз.

Жаңа графикте $a, a + 2 * b, 2 * c$ аламыз, мұнда $a + b + c = 2^{k+1}$.

Бұл операцияның кері $a, b, c \to a, (b - a)/2, c/2$ сияқты көрінеді.

Операцияны қолдану үшін бізге $b \geq a$, b және a бірдей паритет, тіпті c қажет.

Бізде мұндай операцияның 6 нұсқасы бар. $a + b + c$ жұп болғандықтан, олардың кем дегенде біреуін қолдануға болады.

Соңында біз k түйіндерін қолданамыз.

Problem D. Атомдар

$[Da]$ атомдарын *қызыл нүктелер*, $[Bs]$ атомдарын *ақ нүктелер* және $[Km]$ атомдарын *ақ нүктелер* деп қарастырайық.

- $n = 3$. 9 **ұпай.**

  Кез-келген іріктеу шешімі жұмыс істеуі қажет. Сіз тіпті барлық жағдайларды қолмен жазуға тырысуыңызға болады.

- $d_1 = d_2 = \cdots = d_n$. 8 **ұпай.**

  Барлық қызыл нүктелердің координаттары бірдей. Қызыл нүктелердің сол жағында жатқан ақ нүктелердің санын тауып, ол санда x деп белгілейік. Сол сияқты, қызыл нүктелердің оң жағында жатқан қара нүктелердің санын тауып, оны y деп белгілейік. Біз ең көп дегенде $k = min(x, y)$ үштік құра аламыз, сондықтан энергиясы ең көп k қызыл нүктені алсақ жеткілікті.

- $k_i \geq b_j$ **және** $k_i \geq d_j$**, мұнда** $1 \leq i, j \leq n$. 11 **ұпай.**

  Барлық *қара нүктелер* қалған нүктелердің оң жағында орналасқан. Бұл біз кез-келген үштік үшін кез-келген қара нүктені алсақ болатынын білдіреді, демек біз ол нүктелерге қарамай, тек қызыл және ақ нүктелерге назар аударсақ болады.

  Мұнда келесі ашкөз алгоритм жұмыс істейді. Біз координатаның кему ретімен барлық ақ нүктелер бойынша итерация жасаймыз. Біз қазір x координатасындағы ақ нүктені қарастырып жатырмыз делік, x-тің оң жағындағы барлық қызыл нүктелердің ішінен біз энергиясы ең көп нүктені таңдап, оны жойып тастаймыз.

  Неліктен бұл дұрыс? Біз қызыл нүктелер жиынтығын сақтаймыз. Оңнан солға қарай жүргенде, жиынтыққа бірнеше жаңа қызыл нүктелер қосылады немесе біз жаңа жұпты қалыптастыру үшін қолдануға болатын ақ нүктені кездестіреміз. Жиынтыққа қосылған кез-келген қызыл нүкте әрдайым кейінірек қол жетімді болатындықтан, локалды оптимумды таңдау жұмыс істейтіні анық.

- $c_1 = c_2 = \cdots = c_n = 1$. 11 **ұпай.**

  Барлық қызыл нүктелердің шығыс энергиясы 1-ге тең, демек біз тек олардың ең көп мүмкін мөлшерін таңдауымыз керек. Бұл жағдайда біз басқа ашкөз алгоритмін қолдана аламыз. Солдан оңға қарай барлық қызыл нүктелермен жүрейік. Біз ағымдағы қызыл нүктені қамтитын үштікті құруға тырысамыз.

  Егер біздің нүкте бірнеше ақ нүктелермен жұп құрай алатын болса, біз олардың кез-келгенін таңдай аламыз, өйткені ол ештеңеге әсер етпейді. Егер біздің нүкте бірнеше қара нүктелер жұп құрай алатын болса, біз олардың ең жақынын таңдауымыз керек, өйткені бұл қара нүкте кейін қол жетімді емес болатын ең бірінші нүкте. Егер біз екі жақтан да жұп таңдай алатын болсақ, онда үштік құрып, барлық үш нүктені жоямыз.

- $n \leq 300$. 12 **ұпай.**

  Оптималды жауапта біз k қызыл нүкте қолданамыз деп болжамдайық. Әрбір қызыл нүктенің сол жағынан ақ және оң жағына қара нүкте қажет. Сондықтан әрқашанда тек ең сол жақта және ең оң жақта орналасқан нүктелерді таңдау оптималды.

  Сонымен, біз бірнеше ақ ($L_1 \leq \ldots \leq L_k$) және бірнеше қара ($R_1 \leq \ldots \leq R_k$) нүктелерді таңдадық. Енді біз шығыс энергияларының қосындысы ең көп k қызыл $(M_1, \ldots, M_k)$ нүкте таңдауымыз қажет. Кейбір қызыл нүктелер жұбы $M_i \leq M_j$ үшін $L_i \leq L_j$ және $R_i \leq R_j$ шарттары орындалуы қажеттігіне назар аударыңыз.

  Бұл дегеніміз, біз (L_i, R_i) жұптарын құра аламыз және k қызыл нүктелерін таңдау мәселесін әр нүкте сәйкес сегменттің ішінде болатындай және олардың жалпы шығыс энергиясы максималды мүмкін болатындай етіп шеше аламыз.

  Бұл динамикалық бағдарламалау арқылы шығарылады. $dp_{i,j}$ — біз алғашты j сегментті қалыптастырып және алғашқы i қызыл нүктелерді қарастырғандағы жауапқа тең болсын. Отпелер оңай — біз i-ші нүктені j-ші сегментке қоямыз немесе қоймаймыз. Бұл жағдайда біз уақыт күрделіліге $O(n^3)$ болатын шешім аламыз.

- $n \leq 2000$. 12 **points.**

  Біз тіркелген k үшін $O(n^2)$-дан жақсырақ шешім таба алатынымыз түсінікті, солай ма? Шынымен де, керемет ашкөз шешім бар.

  Бұл жолы біз координаттардың өсу ретімен барлық қара нүктелерден өтеміз. Біздің қазіргі қара нүктеміз x нүктесінде орналасқан делік. x-тің сол жағындағы барлық қызыл нүктелердің ішінен біз оның сол жағында ақ нүктесі бар және шығыс энергиясы ең үлкен нүктені таңдағымыз келеді. Егер мұндай қызыл нүкте болса, біз оны аламыз. Сонымен қатар, оның сол жағындағы барлық ақ нүктелерден ең жақынын таңдаймыз.

  Алдымен, оны қалай тезірек іске асыруға болады? Біз қара нүктелерді x-тің өсу ретімен қарастыратындықтан, біз келесі жаңартуларды шешуіміз қажет:

  1. **Координатасы x және шығыс энергиясы y-ке тең қызыл нүкте қосу.** Біз барлық қызыл нүктелерді жиынтықта y-тің кему ретімен сақтаймыз.
  2. **Координатасы x-ке тең ақ нүкте қосу.** Біз барлық ақ нүктелерді жиынтықта x-тің өсу ретімен сақтаймыз.
  3. **Үштік құрауға болатындығын тексеру.** y мәні ең үлкен болатын қызыл нүктені алайық. Егер ол ең алғашқы ақ нүктенің сол жағында орналасқан болса, біз оны ендігәрі ешқандай үштікке қоса алмаймыз. Сондықтан біз оны жиынтықтан өшіріп, келесі қызыл нүктені қарастырамыз. Олай болмаса, біз оптималды кызыл нүктені таптық. Енді біз оның сол жағындағы ең жақын ақ нүктені тауып, жоямыз.

  Әрбір операция $O(\log n)$ уақыт алатындықтан және біз $O(n)$ операция жасайтындықтан, уақыт күрделілігі тіркелген k үшін $O(n \log n)$ болатын шешім аламыз. Жалпы айтқанда, $O(n^2 \log n)$ болады.

  Енді дәлелдеуге көшейік. Тағы да, біз x үлкейту ретімен итерацияларды орындайтындықтан, егер біздің жиынтыққа кез келген уақытта жаңа қызыл нүкте қосылса, ол әрқашан кейінірек қол жетімді болады. Сонымен, *3-ішкі тапсырма* сияқты, біздің қазіргі қызыл нүкте таңдауымыз болашақта ештеңені бұзбайды, сондықтан біз олардың ең жақсысын ашкөздікпен таңдауымыз керек. Біз қызыл нүктені тапқаннан кейін, оны ең жақын ақ нүктеге неге қосу керек екені белгілі болады.

- **Қосымша шектеулер жоқ.** 37 **ұпай.**

  Енді $f(k)$ тіркелген k үшін ашкөздіктің нәтижесін білдірсін делік. Кейбір $k > c$ үшін $f(k)$ белгісіз болатындығы түсінікті. Егер солай болса, біз $f(k)$ $-\infty$-ке тең деп санаймыз.

  Біз $f(k)$-ның максималды мәнін тапқымыз келеді. Енді біз өте қызықты бөлікке көшеміз.

Лемма. $f(k) - f(k-1)$ мәні өспейді. Яғни, $f(k)$ қандай да бір t нүктесіне дейін өседі, сосын кемиді.

Дәлел. Формалды дәлел *Min-Cost-Max-Flow* алгоритмына сүйенеді. Біз өз мәселемізді бірінші қабатта барлық ақ нүктелер, екінші қабатта барлық қызыл нүктелер және үшінші қабатта барлық қара нүктелер бар желі ретінде елестете аламыз. Барлық қырлардың бірлік сыйымдылығы болады. Әрбір қызыл нүкте тек бір рет алынатындағына кепілдік беру үшін, біз оны екі u және v төбелеріне қайталаймыз, сосын u-ды барлық ақ нүктелермен, ал v-ны барлық қара нүктелермен байланыстырамыз, және u және v төбелерін сыйымдылығы бірлік және бағасы $-c_i$ болатын жалғыз қырмен байланыстырамыз, мұнда c_i қазіргі қызыл нүктенің шығыс энергиясы.

Біз іздейтін жауап осы құрастырылған жүйедегі теріс таңбамен алынған *MCMF*-дын мәніне тең болатыны түсінікті. Сөйтіп егер біз *MCMF* қалай жұмыс істейтіндігін қарастырсақ, біз оның $k = 0$ ағымынан бастайды, және қаныққан болуы мүмкін желідегі ең қысқа жолды табу арқылы $k + 1$ ағымына дейін кеңеюге тырысатынын байқаймыз. *Min-Cost-Max-Flow* вариациясы k-ді бұдан былай мүмкін болмайынша көбейтуді жалғастырады, сондықтан бізді *Min-Cost-Flow* вариациясы қызықтырады, мұнда біз k-дің нақты мәніне назар аудармаймыз, бағаны барынша кемитуге тырысамыз. Бұл нұсқада алгоритм $f(k)$ максимизациялайтын нақты k мәніне тікелей тоқтайды. Сондықтан біз оны дәлелденген деп санаймыз.

Бұл лемманы біле отырып, біз енді тернарлық іздеу арқылы оңтайлы k-ді таба аламыз. Сіз мұны $f(m)$ және $f(m+1)$ мәндерін салыстыру арқылы бинарлық іздеу стилінде жүзеге асыра аласыз, бірақ дәстүрлі тернарлық іздеу де өтеді.

Нәжтижесінде, біз уақыт күрделілігі $O(n \log^2 n)$ немесе $O(n \log n \log_{1.5} n)$ болатын шешім аламыз.

Problem E. Ағаштағы ойын

- $u_i = 1, v_i = i + 1$, $r_i = b_i = 1$. 9 **ұпай.**

  Ағаш барлық шыңдары 1-ға қосылған жұлдызды құрайды. Бірінші ойыншы 1 бастаса, ұпай $(n-1) : 1$ болады. Әйтпесе, бірінші ойыншы басқа ойыншының шыңын талап етіп, 1-ға ауысады, сондықтан есеп $n : 0$ болады.

- $u_i = i, v_i = i + 1$, $r_i = b_i = 1$. 13 **ұпай.**

  Ағаш $1 - 2 - \ldots - n$ тізбегін құрайды. a бірінші ойыншының шыңы және b екінші ойыншының шыңы болсын. WLOG $a \le b$.

  Егер олардың арасындағы қашықтық 1 болса, әр ойыншының ұпайы ағаштың жағындағы түйіндер саны болады $(a : n - b + 1)$.

  Егер олардың арасындағы қашықтық 2 болса, бірінші ойыншы ортаға шығып, $n : 0$ есебімен жеңеді.

  Егер қашықтық 2-дан асса, іс қызық болады. Олардың екеуі де арақашықтық 3-ға дейін төмендегенше бір-біріне қарай жылжитынын көрсетуге болады. Осы сәттен бастап олардың жақындау ешқашан оңтайлы емес, өйткені қашықтық 2 болады және басқа ойыншы ортасында жеңеді. Екеуі де басқа қозғалыстарды олардың біреуінің қимылы қалмайынша жасайды. Ол ойыншы ортаға шығуға мәжбүр болады және есеп сол жерден оңай есептелетін болады.

  Қашықтықтың теңдігін де, екі ойыншының да қосымша қозғалыстарының санын ескере отырып, бұл біраз жұмысты қажет етеді.

- $n, q \le 10$. 11 **ұпай.**

  Мұнда кез келген дұрыс енгізілген bruteforce симуляциясы жұмыс істеуі керек.

- $q = 1$, $r_i = b_i = 1$. 14 **ұпай.**

  Тағы да, 2 қосалқы тапсырмасындағыдай, ойыншылардың шыңдары арасындағы қашықтықты қарастырайық. Стратегия дәл сол күйінде қалады, тек бұл жолы ағашта істеу керек. $q = 1$ болғандықтан, сіз мұны $O(n)$ ішінде аңғалдықпен жүзеге асыра аласыз.

- $r_i = b_i = 1$. 20 **ұпай.**

  Жоғарыдағы қосалқы тапсырма сияқты, бірақ келесі нәрселерді жылдам есептеу керек болады:

  1. a және b екі шыңы арасындағы қашықтық.
  2. a және b арасындағы жолдағы k-ші шыңы.
  3. v ішкі ағашындағы шыңдар саны, егер ағаштың түбірі u болса.

  Бұлардың барлығы ағаштардағы тапсырмаларда жиі кездесетін өте классикалық заттар және әрқайсысын $O(\log n)$ немесе $O(1)$ ішінде іске асыруға болады.

- $q = 1$. 17 **ұпай.**

  Мұнда бізге ағымдағы ойын беріледі, сондықтан бізде қарастыратын істер көп.

  Егер оның екі шеткі нүктесі де қарама-қарсы түстерге боялған кем дегенде бір e жиегі болса, бұл болашақта ешбір қозғалыс кез келген шыңды қайта боямайтынын білдіреді. Сондықтан біз бұл жиекті жойып, қалған ағаштарды бөлек шеше аламыз. Қалғандардың әрқайсысында бізде ең көп дегенде бір түс болатынын және бұл түс e сәйкес соңғы нүктесі болатынын жылдам, жылы дәлелдеуге болады. Сондықтан жауапты қарапайым DFS көмегімен есептеуге болады.

  Егер мұндай жиек болмаса, бұл R және B шыңдарының екі қосылған құрамдас бөлігі бар дегенді білдіреді. (a, b) төбелерінің жұбын табайық, сонда $a \in R$, $b \in B$ және $dist(a, b)$ мүмкін емес. Ол жұпты төмендегідей таба аламыз. Кез келген $a' \in R$ шыңын алыңыз, оған ең жақын $b \in B$ төбесін табыңыз. Сол сияқты кез келген $b' \in B$ төбесін алыңыз, оған ең жақын $a \in B$ төбесін табыңыз. Қажетті жұп - бұл процесте тапқан (a, b).

  (a, b) жұбын тапқаннан кейін, бұл ойын күйін (a, b) шыңдары бар **start** ойыншылар үшін алдын ала боялған кейбір басқа шыңдары бар күйге тең етіп көруге болады. . Сонымен, бұдан былай шешім жоғарыдағы қосалқы тапсырмадағыдай болады.

- **Басқа шектеулер жоқ.** 16 **ұпай.**

  Тағы да сол шешімді тиімді түрде жүзеге асыруымыз керек. e жиегін табу $O(|R| + |B|)$ ішінде келесідей орындалуы мүмкін. Ағаш еркін түйінде тамырланған делік. Содан кейін әрбір e шеті кейбір v шыңын өзінің негізгі p_v нүктесімен қосады. Сонымен, бұл жиектерді табу үшін $(a \in A, p_a)$ және $(b \in B, p_b)$ жұптарын қарастыру жеткілікті.

  Егер e осындай жиектерді тапсақ, енді ішкі ағаштардың өлшемдерін есептеуге тура келеді. Біз мұны жылдам шеше аламыз, себебі ол мына тапсырмадан алынған: *Ағаштың түбірі u* болса, v ішкі ағашындағы шыңдар саны. Сонымен, мұндағы барлық нәрсе $O(|R| + |B|)$ немесе $O((|R| + |B|) \log n)$ тілінде жұмыс істейді.

  Егер мұндай e жиектері болмаса, жоғарыда сипатталған жолмен (a, b) табуымыз керек. Аңғал енгізу $O((|R| + |B|) \log n)$ уақыт күрделілігінде бұрыннан жұмыс істейді. Осыдан кейін шешім $r_i = b_i = 1$ жағдайына келетіндіктен, барлық мәселе шешілді.

  Жалпы алғанда, LCA және басқа алгоритмдердің орындалуына байланысты кейбір шамалы айырмашылықтары бар $O(n \log n + (|R| + |B|) \log n)$ уақыт күрделілігін аламыз.

Problem F. Зерттеушілер

- $n, m, q \leq 100$, $d_i \leq 100$, $r_i \leq 100$. 5 **ұпай.**

  1-ден 100-ға дейінгі барлық жылдардағы графты құрастырыңыз және әрбір сұрақ үшін қарапайым DFS көмегімен жету мүмкіндігін тексеріңіз. Уақыт күрделілігі $O(max\{r_i\}(n+m)q)$.

- $n, m, q \leq 3000$, $d_i \leq 3000$, $r_i \leq 3000$. 7 **ұпай.**

  Сол сияқты, 1-ден 3000-ға дейінгі барлық жылдардағы графикті құрастырыңыз. Барлық компоненттерді DFS көмегімен бояңыз. c_v v шыңының түсі болсын. Енді барлық сұраларды тексерген кезде, $c_{x_i} = c_{y_i}$ екенін тексеру арқылы қолжетімділікті тексеруге болады.

  Уақыт күрделілігі $O(max\{r_i\}(n+m+q))$.

- $m = n-1$, $a_i = i$, $b_i = i+1$. 12 **ұпай.**

  График $1 - 2 - \ldots - n$ тізбегі болып табылады. $u - v$ жолы болуы үшін ($u < v$) олардың арасындағы барлық жиектер болуы керек. Осылайша $[u, v-1]$ диапазонындағы сегменттердің қиылысуын есептеуге дейін барады, оны сегмент ағашымен жасауға болады. Ауқымнан $max\{l_i\}$ және ауқымда $min\{r_i\}$ табу керек.

  Уақыт күрделілігі $O(n + q \log n)$.

- $d_i = 10^9$. 16 **ұпай.**

  $d_i = 10^9$ негізінен жиектер ешқашан жойылмайтынын білдіреді. Сонымен, барлық жиектерді қосу уақытының ретімен сұрыптайық. Енді әрбір (u, v) сұрағы үшін u және v шыңдары қосылған кезде t бірінші рет екілік іздеуге болады. Мұны істеудің ең оңай және ең танымал жолы - *Parallel binary search*.

  Уақыт күрделілігі $O((n+m+q)\log 10^9)$.

- $l_i = r_i$. 12 **ұпай.**

  Мәселені келесіге түрлендірейік. 3 көлеміндегі оқиғаларды хронологиялық тәртіпте өңдеу керек:

  1. (u, v) жиегін қосыңыз.
  2. (u, v) жиегін алып тастаңыз.
  3. u және v шыңдары қосылған туралы сұрағы.

  Бұл *Dynamic Connectivity* деп аталатын өте классикалық мәселе. Бұл нақты жағдайда біз оны кері қайтарулары бар DSU көмегімен *divide and conquer* арқылы офлайн режимінде жасай аламыз.

  $O((n+m+q)\log^2 n)$.

- $n, m, q \leq 40000$. 27 **ұпай.**

  Хронологиялық ретпен екі түрдегі 2 оқиғаларын жасайық:

  1. (u, v) жиегін қосыңыз.
  2. (u, v) жиегін алып тастаңыз.

  Біз *Оқиғалар бойынша квадрат түбірді ыдыратамыз.*

  Әрбір сұрақ (l, r) оқиғалар ауқымына сәйкес келеді. Оқиғалардың барлық блоктарын санап көрейік. Әрбір блок үшін оны толығымен қамтитын кейбір сұрақтар болады және оны тек ішінара қамтитын кейбір сұрақтар болады.

  Сондай-ақ k өлшемді блоктың ішінде әсер ететін ең көбі $2k$ сәйкес шыңдар мен жиектер болады. Сонымен, бүкіл графикті қысып, пайдасыз шыңдар мен шеттерден арылайық.

  Енді біз ішінара сұрақтарды оңай шеше аламыз. Дәл 2 қосалқы тапсырмасында орындағанымыз сияқты, оқиғаларды бір-бірден түзетіп, DFS көмегімен барлық құрамдастарды бояйық.

Содан кейін біз барлық жартылай сұрақтарды қарап шығамыз және егер сұрақ ағымдағы оқиғаны қамтыса, біз оған жауапты сәйкесінше жаңартамыз.

Бұл бөлік әр блок үшін $O(k^2)$ жұмыс істейді. Және сұрақ ең көбі екі блокта ішінара қарастырылады, сондықтан сұрақ жалпы уақыт күрделілігіне тек $O(k)$ қосады.

Енді блокты толығымен қамтитын сұрақтармен айналысайық. Әрбір блокта олардың көп болуы мүмкін. Бірақ бұл жерде маңызды нәрсе бар: графикті сығымдағаннан кейін бізде тек $O(k)$ шыңдары бар, сондықтан бізде $O(k^2)$ ғана төбелер бар. Сонымен, егер біз әрбір (u, v) жұбы үшін $ans_{u,v}$ екі өлшемді массивін сақтасақ және оны ішінара сұрақтардағыдай есептесек, бір блок үшін $O(k^3)$ уақыт күрделілігін аламыз.

Енді оңтайлы k туралы ойланайық. Бізде жалпы n/k блоктары болады, олардың әрқайсысында графикті қысу және сәйкес сұрақтарды іздеу үшін шамамен $O(n + m + q)$ істеуіміз керек. Блок ішінде біз $O(k^3)$ операцияларын орындаймыз. Сонымен, егер екі жағын тең етуге тырыссақ, оңтайлы k $(n + m + q)^{1/3}$ шамасында болуы керек.

Осылайша, біз алатын жалпы уақыт күрделілігі $O(n+m+q)^{5/3}$, бұл ішкі тапсырманы орындау үшін жеткілікті болуы керек.

- **Қосымша шектеулер жоқ.** 21 **ұпай.**

  Шындығында, алдыңғы ішкі тапсырманың шешімін тіпті осы қосалқы тапсырмадан өтетін дәрежеге дейін оңтайландыруға болады. Бірақ бұл жерде біз жақсырақ шешімді талқылаймыз.

  Біздің шешіміміздің тар жолы $O(k^3)$ бөлігі болды, онда біз блоктағы барлық (u, v) жұптары үшін $ans_{u,v}$ есептейміз. Біз мұны тезірек жасай аламыз ба? Жауап бақытымызға орай иә.

  Айта кетейік, бізде бірдей бастапқы мәселе бар, бірақ бұл жолы $n = O(k)$, $m = O(k)$ және $q = O(k^2)$ барлық сұрақтар бүкіл ауқымда болады. сондай-ақ.

  Сонымен, біз сол ішкі мәселеге бірдей шешімді қолдана аламыз ба? $t = \sqrt{k}$ блогын алып, t өлшемді блоктарды қайтадан шешіңіз бе? Теориялық тұрғыдан, біз мүмкін, бірақ ол жоғарыда талқыланған $O(k^3)$ қарағанда баяу аяқталуы мүмкін.

  Бірақ оның орнына біз мұны толығырақ талдап, осы ішкі мәселе үшін оңтайлы блок өлшемі t $n/2$ болатынын анықтай аламыз. Басқаша айтқанда, бұл ішкі мәселені бөлу және жеңу алгоритмі ретінде шешу оңтайлы. Барлық оқиғаларды дәл ортасына бөліңіз, барлық шыңдар мен жиектерді екі жағынан қысыңыз, рекурсивті түрде шешіңіз және жауаптарды біріктіріңіз.

  Уақыттың күрделілігін талдап көрейік. Бізде $T(n) = 2T(n/2) + O(n^2)$ бар. Кеңейтсек, $T(n) = O(n^2 + 2 \times (n/2)^2 + 4 \times (n/4)^2 + \ldots) = O(n^2 \times (1 + 1/2 + 1/4 + \ldots) = O(n^2)$. Сонымен $n = k$ үшін біз $O(k^2)$ есебіндегі мәселені шеше аламыз, бұл дәл біз іздеген нәрсе.

  k мәнін шамамен $O(\sqrt{n + m + q})$ деп алсақ, біз $O((n+m+q)\sqrt{n + m + q})$ шешімін аламыз, өте жоғары тұрақты болса да.

Задача A. Горка

- $n = 1$. 12 **баллов.**

  Переберите все возможные пары (L_1, R_1) и выберите лучшую. Это можно сделать за $O(m^2)$ или $O(m)$

- **Нет заблокированных клеток.** 7 **баллов.**

  Всегда оптимально выбирать горкой всю таблицу. Таким образом, ответ - это просто сумма всех значений.

- $n, m \leqslant 50$. 25 **баллов.**

  Пусть $dp_{i,l,r}$ — наибольшая сумму горки, которую мы можем получить, если он заканчивается в строке i отрезком (l, r).

  Чтобы его вычислить, перебирите все пары (l_p, r_p) такие, что $l \leqslant l_p \leqslant r_p \leqslant r$, и попробуйте расширить горку dp_{i-1,l_p,r_p}. Также нужно рассмотреть случай, когда новая горка начнется в текущем ряду.

  Общая асимптотика $O(nm^4)$ с небольшой константой.

- $n, m \leqslant 300$. 22 **баллов.**

  Точно такое же решение, как и выше, но на этот раз вместо перебора (l_p, r_p) мы можем заметить, что мы просто ищем максимальную горку по всем $l \leqslant l_p \leqslant r_p \leqslant r$. Это можно поддерживать с помощью другого DP, скажем, $best_{i,l,r}$. Для перехода будем рассматривать $dp_{i,l,r}$, $best_{i,l+1,r}$ и $best_{i,l,r-1}$.

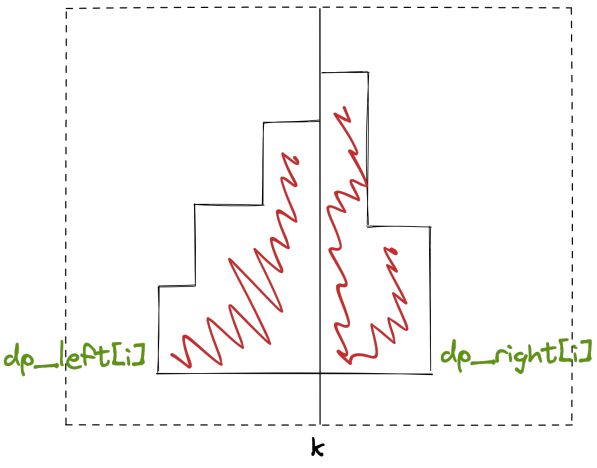
  Если реализовать аккуратно, асимптотика будет $O(nm^2)$.

- **Нет дополнительных ограничении.** 34 **баллов.**

  Горка в его нынешнем определении довольно сложно вычислить напрямую. Поэтому давайте попробуем сделать это по другому.

  Глядя на свойства горки, мы замечаем, что он будет увеличиваться до некоторого столбца, а затем уменьшаться. Пронумеруем этот столбец k и разделим горку на левую и правую части. Слева должно увеличиваться, а справа уменьшаться. Мы обнаруживаем, что они полностью симметричны, поэтому давайте сосредоточимся на расчете только левой части.

  Определим dp_i — наилучшую возможную сумму горки, если его **bottom** начинается со строки i (альтернативно, $e = i$), а нижний левый угол начинается где-то слева от столбца . Точно так же мы определим такой dp в правой части и попытаемся объединить результаты.

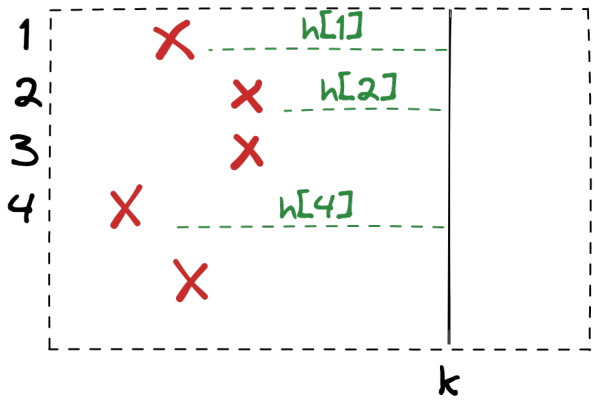


Горка объединенная в столбце k.

Понятно, что после того, как мы вычислили значения $dp_1, \ldots, dp_n$ как слева, так и справа, мы можем объединить их, перебрав e от 1 до n и взяв максимально возможную сумму $(dpLeft_e + dpRight_e)$.

Поэтому, если мы каким-то образом сможем вычислить dp за время $O(n \log n)$ или $O(n)$, задача будет решена.

Давайте сосредоточимся на вычислении dp_1. Мы должны попытаться взять как можно более длинный ряд, пока не наткнемся на заблокированную клетку. Пусть h_1 обозначает максимальное количество ячеек, которые мы можем взять. Точно так же мы определим $h_2, \ldots, h_n$. Каждую из них можно найти за $O(1)$, если предварительно вычислить ближайшую заблокированную клетку слева и справа для каждой ячейки (i, j).

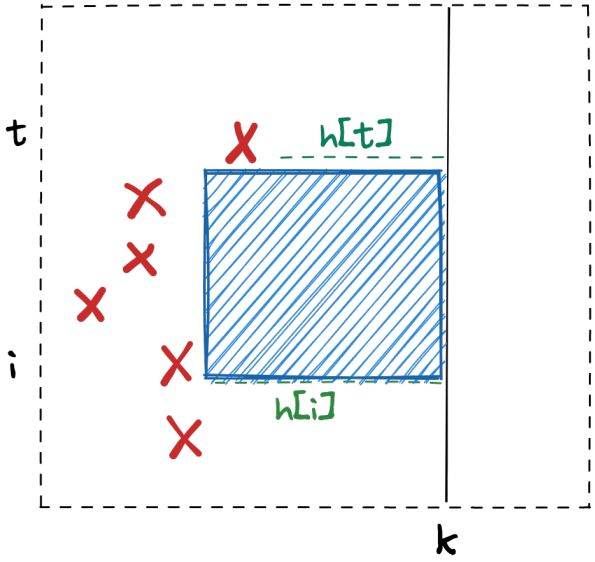


Как определяются значения h_i.

Теперь должно быть более понятно, как вычислить значение dp_i. Сначала мы возьмем все ячейки h_i из i-й строки. Затем попробуем взять c_j ячеек для всех строк $j < i$, где $c_j \leqslant c_{j+1}$ и $c_j \leqslant h_j$. Мы могли бы вычислить эти значения c жадным образом, перейдя от i к 1 и сохранив текущее значение c_j. Но это привело бы к решению $O(n^2 m)$, которое все еще довольно медленно.

Вместо этого попробуем повторно использовать предыдущие значения dp_j. Найдем наибольшее значение t, такое что $h_t < h_i$. Для всех строк в диапазоне $[t+1, i]$ все значения c_j будут равны h_i. И этот диапазон точно образует прямоугольник, поэтому мы можем взять его значение за $O(1)$ с 2D префиксных сумм.

Диапазон $[1, t]$ будет точно таким же, как и в dp_t, поэтому мы можем заключить, что $dp_i = dp_t + sum(t, i)$, где sum — упомянутая выше сумма прямоугольников.



Вычисление dp_i.

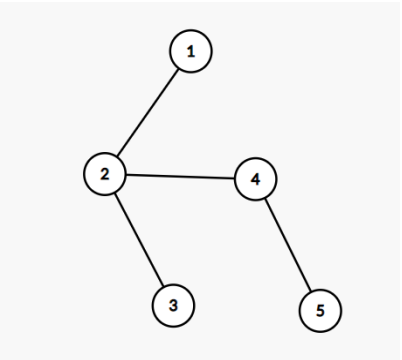
Мы можем найти такие t для всех i многими способами. Самый быстрый способ - поддерживать монотонный стек. Другие способы включают бинарный поиск или хранение seta вместо этого, но это добавит дополнительный фактор $O(\log n)$ к времени.

Подводя итог, мы получаем довольно хорошее решение со сложностью памяти и временем $O(nm)$.

Задача B. Два дерева

- **Для 1 подзадачи**, можно посчитать размеры поддеревьев через DFS, и посчитать ответ.
- **Для 2 подзадачи**, заметим, что ответ для вершины x, который сосед вершины 1, не будет сильно менять ответ, потому что после подвешивание дерева за x только размеры поддеревьев для вершин 1 и x изменятся. Можно посчитать ответ для вершины 1 и указать на эту вершину с каждого дерева. Мы можем создавать запросы для каждой вершины v, ответ будет найден после перемещения обоих указателей на вершину v и сравнения измененные вершины. Чтобы сделать его оптимальным, мы можем использовать алгоритм МО.
- **Для 4 подзадачи**, заметим, что ответ для вершины v равен ответу для вершины u, если не считать вершины на их пути. Поскольку дерево в этой подзадаче является полным бинарным деревом, расстояние между двумя вершинами невелико, поэтому вы можете пройтись по пути и вычислить ответ.
- **Для 3 подзадачи**, решим задачу для x – для каких v $sub_1(x) > sub_2(x)$ будет выполнено. $sub_1(x)$ может иметь $deg_1(x)+1$ разных значений и $sub_2(x)$ имеет $deg_2(x)+1$ разных значений, где $deg_1(x)$ — количество соседей в первом дерево, а $deg_2(x)$ — количество соседей во втором дереве.

Рассмотрим следующее дерево с вершиной 1 в качестве корня:



Размеры поддеревьев: $sub_1 = 5$, $sub_2 = 4$, $sub_3 = 1$, $sub_4 = 2$, $sub_5 = 1$. Рассмотрим ребро $2, 4$ (значит корень v будет вершиной либо 4, либо 5) и размер поддерева 2 будет 3, то есть $n - sub_4$. Точно так же мы можем вычислить размер поддерева 2 для каждого из сыновей. Тем не менее, у нас также есть ребро от родителя 1, в этом случае размер нашего компонента равен sub_2, а размер другого компонента равен $n - sub_2$.

Если x будет корнем дерева, тогда $sub_1(x)$ и $sub_2(x)$ равны n. В этом случае никогда не будет $sub_1(x) > sub_2(x)$.

Давайте также рассмотрим случай, когда наш родитель a в первом дереве и b во втором. Тогда, sub_1 будет равен размеру компоненты с вершиной x, когда мы удаляем ребро a, x из первого дерева. Аналогично, $sub_2(x)$ является равным размеру компоненты с вершиной x, когда удаляем ребро b, x из второго дерева. Если $sub_1(x) > sub_2(x)$, то оно выполняется для каждой вершины v, находящегося в одной компоненте как и с a (после удаления ребра a, x из первого дерева), так и с b (после удаления ребра b, x из второго дерева).

Вы можете использовать Эйлеров обход, чтобы проверить, что вершина v находится внутри компонента. Обратите внимание, что каждый компонент является подотрезков в Эйлеровым обходе.

Рассмотрим вершину v как точку на $2D$ пространстве — tin_v в первом дереве как ось x и tin_v во втором дереве как ось y. Можно перебирать каждую пару соседей (один из первого дерева, другой из второго дерева) и строить запросы на добавление в подпрямоугольнике. Эту задачу можно решать за время $n \cdot log(n)$. Но все же у нас есть $deg_1(v) \cdot deg_2(v)$ пар вершин, так что это решение проходит только на 1, 3 и 4 подзадачах.

- **Полное решение.** Чтобы оптимизировать эту часть мы будем использовать два указателя. Итак, пусть соседи для каждой вершины отсортированы по размеру поддерева в обоих деревьях отдельно, и пусть корнем будет вершина 1. Нам не нужно перебирать каждую пару соседей, вместо этого возьмем соседа a, и размер нашего компонента будет равен $n - sub_1(a)$, и эти значения уменьшаются как *subtree* соседей увеличиваются. То же самое со вторым деревом, где $n - sub_2(b)$. Мы можем использовать два указателя, чтобы найти для вершины a первый такой b, что $n - sub_1(a) > n - sub_2(b)$, и теперь для всего поддерева b и поддеревьев соседей после него это будет выполняться. Обратите внимание, что вам нужно рассматривать родителя вершины x в первом дереве и родителя во втором дереве отдельно друг от друга и с сыновьями вершины x в обоих деревьях.

Задача C. Гольф

Пусть $v_a = 1$, $v_b = 2$ и $v_c = $ s, вервая, вторая конечные вершины и корень соответсвенно.

1. $n = 100$ $a + b = 4$. 10 баллов.

   Решите в ручную.

2. $n = 100$ $a + b = 32$. 10 points.

   Постройте полное бинарное дерево с 32 листьями. Направьте первые a листьев в v_a и оставшиеся b в v_b.

   Заметьте, что все листья имеют ровную вероятность.

3. $n = 50$ $a + b = 2^{30}$. 10 баллов.

   (a) Если $a = b$. Направьте ребра из v_c в v_a и v_b.

   (b) Если a и b четны. Поделите a и b на два. Вероятность не изменится.

   (c) Если $a \leqslant b$, a и b нечетные.
   Создайте новую вершину u и направьте ребра из v_c в v_b и u.
   Now we can recursively build a graph with a, $\frac{b-a}{2}$ starting from u.
   Заметьте $a + \frac{b-a}{2} = \frac{a+b}{2}$ остается степенью двойки.

(d) Если $a \geqslant b$, a и b нечетные. Так же как и в предыдущем случае.

4. $n = 33$ $a, b \leqslant 15$. 10 баллов.

   Постройте случайный граф и проверьте вероятности симулируя случайные прохождения.

5. $n = 64$ 10 баллов.

   Найдите наименьшее k такое что $2^k \geqslant a + b$.

   Постройте полное бинарное дерево с $2^k \geqslant a + b$ листьями.

   Направьте первые a листьев в v_a и следующие b в v_b и остальные в v_c.

   Если оба конца из вершины ведут в одну и ту же вершину то мы можем удалить эту вершину и направить все входящие ребра в следующую вершину.

   Поскольку все листья ведущие в v_a образуют последовательный интервал, мы можем сжать их в $2 * k$. Так же как и в дереве отрезков.

   Так же для v_b и v_c. В конце, мы используем $2 * k$ вершин.

   Заметим что если какой то из концов интервала совпадает с концом всего дерева, оно не создаст новых вершин. Также, правый конец v_a и левый конец v_b совпадают и оба используют одну из вершин. Так же для v_b и v_c.

6. $n = 50$, 10 баллов.

   Давайте построим дерево отрезков более оптимально.

   $a + b + c = 2^k$. Здесь c количество вершин ведущих в корень.

   Вершина на глубине d относится к 2^{k-d} листьям.

   Мы начинаем с корня на глубине 1.

   Если $a \geqslant 2^{k-1}$ направьте первое ребро в v_a и решите со второго сына с $a - 2^{k-1}, b, c$ и $k - 1$.

   Если $2^{k-2} \leqslant a, b \leqslant 2^{k-1}$ направьте первого сына в v_a и v_b и решите для второго сына с $a - 2^{k-2}, b - 2^{k-2}, c$ и $k - 1$.

   Остальные случаи похожи.

   Мы создаем новые вершины только во втором случае. Можно доказать что число будет меньше чем $\frac{2}{3}k$.

   В конце, мы используем $\frac{5}{3} * k$ вершин.

7. $n = 36$, 10 баллов

   Заметьте что вершины созданные во втором случае направлены в две вершины из v_a, v_b, v_c. Есть только 3 способа выбора. Мы можем создать из заранее и использовать их заново.

   В конце, мы используем $k + 3$ вершин.

8. $n = 35$, 10 баллов

   Заметьте что последняя вершина так же ведет в две вершины из v_a, v_b, v_c.

   В конце, мы используем $k + 2$ вершин.

9. $n = 34$, 10 баллов

   Легко заметить что эта подзадача бесполезна.

   В конце, мы используем $k + 1$ вершин.

10. $n = 33$, 10 баллов

    Допустим у нас есть граф который решает эту задачу для a, b, c где $a + b + c = 2^k$.

    Рассмотрим следующие операцию:

(a) Создать новую вершину u.

(b) Добавить два ребра из v_a в v_b и u. Пометить u как первую конечную вершину.

В новом графе мы имеем $a, a + 2 * b, 2 * c$ где $a + b + c = 2^{k+1}$.

Разворот этих операций выглядит как $a, b, c \to a, (b - a)/2, c/2$.

Что бы использовать эту операцию нам нужно $b \geqslant a$, b и a одинаковой четности, четное c.

У нас есть 6 таких возможных операций. Поскольку $a+b+c$ четно, хотя бы одно из них может быть использовано.

В конце, мы воспользуемся k вершинами.

Задача D. Атомы

Обозначим атомы $[Da]$ как *красные точки*, атомы $[Bs]$ как *белые точки*, а атомы $[Km]$ как *черные точки*.

- $n = 3$. 9 **баллов.**

  Любое переборное решение должно работать. Можно даже попробовать прописать все варианты вручную.

- $d_1 = d_2 = \cdots = d_n$. 8 **баллов.**

  Все красные точки имеют одинаковые координаты. Найдите все белые точки, находящиеся слева от красных точек, и обозначьте их количество как x. Аналогичным образом найдите все черные точки справа и обозначьте их количество как y. Мы можем составить не более $k = min(x, y)$ троек, поэтому нам нужно просто взять k красных точек с наибольшим выходом энергии.

- $k_i \geqslant b_j$ **и** $k_i \geqslant d_j$ **длю любой** $1 \leqslant i, j \leqslant n$. 11 **баллов.**

  Все *черные точки* располагаются справа от всех остальных точек. Это означает, что мы можем выбрать любую из них для любой тройки, которую мы можем сформировать, поэтому мы можем просто игнорировать все черные точки и вместо этого сосредоточиться на парах белого и красного.

  Здесь будет работать следующий жадник. Мы будем перебирать все белые точки в порядке убывания координат. Предположим, мы сейчас рассматриваем белую точку с координатой x. Из всех красных точек справа от x мы выберем ту, которая имеет самый высокий выход энергии, и удалим ее.

  Почему это работает? Мы поддерживаем набор красных точек. По мере того, как мы идем справа налево, к набору добавляются новые красные точки или мы встречаем белую точку, которую можно использовать для формирования новой пары. Поскольку любая красная точка, которая была добавлена к набору, всегда будет доступна позже, ясно, что выбор локального оптимума работает.

- $c_1 = c_2 = \cdots = c_n = 1$. 11 **баллов.**

  Все красные точки имеют выход энергии 1, а это значит, что нам просто нужно выбрать наибольшее их количество. Здесь у нас другой вид жадника. Пройдемся по всем красным точкам слева направо. Мы попытаемся сформировать тройку, которая включает в себя текущую красную точку.

  Если наша точка может быть сопряжена с несколькими белыми точками, мы можем выбрать любую из них, так как это ни на что не повлияет. Если наша точка может быть соединена с несколькими черными точками, мы должны выбрать ближайшую, потому что эта черная точка позже станет первой недоступной. Если мы можем образовать пары с обеих сторон, то мы формируем тройку и стираем все три точки.

- $n \leqslant 300$**. 12 баллов.**

  Предположим, что в оптимальном ответе мы получим k красных точек. Каждой красной точке понадобится белая точка слева и черная точка справа. Поэтому всегда оптимально выбирать только **самые левые** белые точки и только **самые правые** черные точки.

  Таким образом, мы выбрали несколько белых точек ($L_1 \leqslant \ldots \leqslant L_k$) и несколько черных точек ($R_1 \leqslant \ldots \leqslant R_k$). Теперь нам нужно выбрать k красных точек ($M_1, \ldots, M_k$) с наибольшей суммарной выходной энергией. Заметим, что для некоторой пары красных точек $M_i \leqslant M_j$ также должны выполняться условия $L_i \leqslant L_j$ и $R_i \leqslant R_j$.

  Это означает, что мы можем составить пары (L_i, R_i) и решить задачу выбора k красных точек, таких, чтобы каждая точка находилась внутри соответствующего сегмента, а их суммарная выходная энергия была максимально возможной.

  Это можно сделать, сохраняя $dp_{i,j}$ — наибольший ответ, если мы сформировали первые j сегментов и рассмотрели первые i красных точек. Переходы тривиальны — либо мы помещаем i-ю точку в j-ый отрезок, либо нет. Таким образом, мы получаем решение с временной сложностью $O(n^3)$.

- $n \leqslant 2000$**. 12 баллов.**

  Ясно, что мы можем добиться большего, чем $O(n^2)$ при фиксированном k? Действительно, существует хорошее жадное решение.

  На этот раз мы пройдемся по всем черным точкам в порядке возрастания координат. Предположим, что наша текущая черная точка находится в точке x. Из всех красных точек слева от x мы хотим выбрать ту, которая имеет белую точку слева от нее и имеет наибольшую выходную мощность. Если есть такая красная точка, мы ее возьмем. Дополнительно из всех белых точек слева от него выберем ближайшую.

  Во-первых, как реализовать это быстро? Поскольку мы проходим все черные точки в порядке возрастания x, нам придется иметь дело со следующими обновлениями:

  1. **Добавим новую красную точку с координатой x и выходом энергии y.** Мы будем хранить все красные точки в наборе в порядке убывания y.
  2. **Добавить новую белую точку с координатой x.** Мы сохраним белую точку в другом наборе в порядке возрастания x.
  3. **Проверить, можем ли мы составить тройку.** Возьмем красную точку с наибольшим значением y. Если она расположена левее крайней левой белой точки, мы уже не можем взять ее в тройку и никогда не сможем. Так что мы можем удалить его из набора и повторить снова. В противном случае мы нашли оптимальную красную точку. Теперь находим ближайшую белую точку слева от нее и стираем ее.

  Поскольку каждая операция занимает $O(\log n)$ времени, а мы амортизируем $O(n)$ операций, мы получаем временную сложность $O(n \log n)$ для фиксированного k, что дает нам $O(n^2 \log n)$.

  Теперь приступим к доказательству. Опять же, поскольку мы итерируем в порядке возрастания x, если в любой момент к нашему набору добавится новая красная точка, она всегда будет доступна позже. Так что, как и в *подзадаче 3*, наш текущий выбор красной точки ничего не испортит в будущем, поэтому мы должны жадно выбрать лучшую. И как только мы нашли красную точку, становится очевидным, почему нам нужно соединить ее с ближайшей белой точкой.

- **Нет дополнительных ограничении.** 37 **баллов.**

  Теперь пусть $f(k)$ обозначает результат жадности выше для фиксированного k. Ясно, что $f(k)$ будет неопределенным для некоторого $k > c$. В этом случае мы будем считать $f(k)$ равным $-\infty$.

Мы хотели бы найти максимальное значение $f(k)$. Теперь мы подошли к очень интересной части.

Лемма. Значение $f(k) - f(k-1)$ не возрастает. То есть $f(k)$ сначала возрастает до некоторой точки t, а потом убывает.

Доказательство. Формальное доказательство следует из алгоритма *Min-Cost-Max-Flow*. Мы можем представить нашу задачу в виде сети, где первый слой содержит все белые точки, второй слой содержит все красные точки, а третий слой содержит все черные точки. Все ребра будут иметь единичные пропускные способности. Чтобы каждая красная точка была взята только один раз, мы продублируем ее в две вершины u и v, соединим u со всеми белыми точками, соединим v со всеми черными точками и соединим u и v с одним ребром единичной мощности и стоимостью $-c_i$, где c_i — выход энергии текущей красной точки.

Понятно, что искомым ответом будет значение *MCMF* в этой сети, взятое с обратным знаком. Итак, если мы рассмотрим, как на самом деле работает *MCMF*, мы заметим, что он начинается с потока $k = 0$ и пытается расшириться до потока $k + 1$, находя кратчайший путь в сети, который может быть насыщен. Вариант *Min-Cost-Max-Flow* продолжает увеличивать k до тех пор, пока это становится невозможным, поэтому нас на самом деле интересует вариант *Min-Cost-Flow*, где нас не волнует точное значение значение k, мы заботимся только о минимизации стоимости. И в этом варианте алгоритм останавливается прямо на точном значении k, которое максимизирует $f(k)$. Поэтому мы считаем его доказанным.

Зная эту лемму, теперь мы можем найти оптимальное k с помощью тернарного поиска. Вы можете реализовать его в стиле бинарного поиска, сравнивая значения $f(m)$ и $f(m + 1)$, но традиционный тернарный поиск также проходит.

В итоге получаем решение, имеющее асимптотику $O(n \log^2 n)$ или $O(n \log n \log_{1.5} n)$.

Задача E. Игра на дереве

- $u_i = 1, v_i = i + 1$, $r_i = b_i = 1$. 9 **баллов.**

  Дерево формирует звезду со всеми вершинами, соединенными с 1. Если первый игрок начинает с 1, счет будет $(n-1) : 1$. В противном случае первый игрок перейдет к 1, захватывая вершину другого игрока, и счет будет $n : 0$.

- $u_i = i, v_i = i + 1$, $r_i = b_i = 1$. 13 **баллов.**

  Дерево формирует цепь $1 - 2 - \ldots - n$. Пусть a это вершина первого игрока, а b - вершина второго игрока. WLOG $a \leqslant b$.

  Если расстояние между ними равно 1, то количество очков для каждого игрока будет равно количеству вершин на его стороне дерева $(a : n - b + 1)$.

  Если расстояние равно 2, то первый игрок сделает ход в центр и выиграет с результатом $n : 0$.

  Если расстояние более 2, то ситуация становится интереснее. Можно доказать, что они будут двигаться друг к другу, пока расстояние не уменьшится до 3. И с этого момента никогда не оптимально двигаться ближе, потому что расстояние становится 2 и другой игрок выигрывает в центре. Поэтому они будут делать другие ходы, пока один из них не закончит ходы. Такой игрок будет вынужден двинуться в центр, и счет будет легко рассчитываться оттуда.

  Это требует некоторого разбора случаев, учитывая как четность расстояния, так и количество дополнительных ходов, которые имеют оба игрока.

- $n, q \leqslant 10$. 11 **баллов.**

  Здесь любой правильно написанный перебор должен работать.

- $q = 1$, $r_i = b_i = 1$. 14 **баллов.**

  Опять, как в подзадаче 2, рассмотрим расстояние между вершинами игроков. Стратегия остается той же, за исключением того, что это нужно сделать на дереве. Так как $q = 1$, вы можете реализовать это наивным методом за $O(n)$.

- $r_i = b_i = 1$. 20 **баллов.**

  Так же, что и подзадаче выше, но в этот раз вам нужно будет быстро вычислять следующие вещи:

  1. Расстояние между двумя вершинами a и b.
  2. k-я вершина на пути между a и b.
  3. Количество вершин в поддереве v если корнем дерева будет u.

  Все это довольно классические вещи, которые часто встречаются в задачах на деревьях, и каждая из них может быть реализована за $O(\log n)$ или $O(1)$.

- $q = 1$. 17 **баллов.**

  Здесь нам дана игра, поэтому у нас есть больше случаев для рассмотрения.

  Если есть хотя бы один ребро e, у которого оба конечных вершины окрашены в противоположные цвета, это означает, что в будущем никакой ход не изменит цвет этих вершин. Поэтому мы можем удалить это ребро и решить отдельно для оставшихся деревьев. Можно доказать, что в каждом из оставшихся деревьев будет не более одного цвета, и этот цвет будет соответствующим конечным вершиной e. Ответ можно вычислить простым DFS.

  Если такого ребра нет, то у нас есть две связные компоненты вершин R и B. Найдем пару вершин (a, b), такую, что $a \in R$, $b \in B$ и $dist(a, b)$ минимально возможное. Мы можем найти эту пару следующим образом. Возьмем любую вершину $a' \in R$, найдем ближайшую вершину $b \in B$ к ней. Аналогично, возьмем любую вершину $b' \in B$, найдем ближайшую вершину $a \in R$ к ней. Желаемой парой будет (a, b), которую мы нашли в этом процессе.

  После того, как мы нашли пару (a, b), этот состояние игры может быть рассмотрено эквивалентным состоянию, где игроки **начинают** с вершинами (a, b), с тем лишь отличием, что некоторые другие вершины уже имеют предопределенные цвета. Таким образом, решение далее будет точно таким же что выше.

- **Без дополнительных ограничений.** 16 **баллов.**

  Вновь, нам требуется реализовать такое же решение оптимальный способом. Нахождение ребра e может бы сделано за $O(|R| + |B|)$ следующим образом. Предположим дерево подвешено за некоторую вершину. Тогда каждое ребро e соединяет какую-то вершину v с предком p_v. Следовательно, достаточно рассматривать пары $(a \in A, p_a)$ и $(b \in B, p_b)$ для нахождения нужных ребер.

  После того как мы нашли ребра e, нам нужно посчитать размеры поддеревьев. Мы можем решить это быстрым способом как и в подзадаче выше: *Количество вершин в поддереве v если корнем дерева будет u*. Решение работает за $O(|R| + |B|)$ или $O((|R| + |B|) \log n)$.

  Если нет такого ребра e, нам нужно найти пару вершин (a, b) как и в предыдущей подзадаче. Наивное решение работает за $O((|R| + |B|) \log n)$.

  Общая сложность решения $O(n \log n + (|R| + |B|) \log n)$ с небольшими изменями зависящями от реализации LCA и других алгоритмов.

Задача F. Исследователи

- $n, m, q \leqslant 100$, $d_i \leqslant 100$, $r_i \leqslant 100$. **5 баллов.**

  Постройте граф для всех лет от 1 до 100, и для каждого запроса проверьте доступность с помощью простого DFS. Временная сложность $O(maxr_i(n+m)q)$.

- $n, m, q \leqslant 3000$, $d_i \leqslant 3000$, $r_i \leqslant 3000$. **7 баллов.**

  Аналогично, постройте граф для всех лет от 1 до 3000. Окрасьте все компоненты с помощью DFS. Давайте c_v будет цветом вершины v. Теперь, когда вы проходите через все запросы, вы можете проверить доступность, просто проверяя, равен ли $c_{x_i} = c_{y_i}$.

  Временная сложность $O(maxr_i(n+m+q))$.

- $m = n-1$, $a_i = i$, $b_i = i+1$. **12 баллов.**

  Граф представляет собой цепь $1-2-\ldots-n$. Для того, чтобы путь $u-v$ существовал ($u < v$), все ребра между ними должны присутствовать. Так что это сводится к вычислению пересечения отрезков в диапазоне $[u, v-1]$, что может быть сделано с помощью дерева отрезков. Вам нужно найти $maxl_i$ на диапазоне и $minr_i$ на диапазоне.

  Сложность времени: $O(n + q \log n)$.

  $d_i = 10^9$ в основном означает, что ребра никогда не будут удалены. Так что давайте отсортируем все ребра в порядке их времени добавления. Теперь для каждого запроса (u, v) мы можем попытаться двоичным поиском найти первое время t, когда вершины u и v будут соединены. Самым простым и популярным способом это сделать является *Parallel binary search*.

  Сложность времени: $O((n+m+q)\log 10^9)$.

- $l_i = r_i$. **12 баллов.**

  Давайте преобразуем проблему следующим образом. Вы должны обрабатывать 3 вида событий в хронологическом порядке:

  1. Добавить ребро (u, v).
  2. Удалить ребро (u, v).
  3. Запрос о том, связаны ли вершины u и v.

  Это очень классическая проблема, называемая *Dynamic Connectivity*. В этом конкретном случае мы можем сделать это в автономном режиме с помощью *разделяй и властвуй* с DSU с откатами.

  $O((n+m+q)\log^2 n)$.

- $n, m, q \leqslant 40000$. **27 баллов.**

  Давайте создадим $2m$ событий двух типов в хронологическом порядке::

  1. Добавить ребро (u, v).
  2. Удалить ребро (u, v).

  Будем делать *Sqrt-декомпозицию по событиям*.

  Каждому запросу соответствует некоторый диапазон (l, r) событий. Пройдемся по всем блокам событий. Для каждого блока будут запросы, которые охватывают его полностью, и будут запросы, которые охватывают его только частично.

  Кроме того, внутри блока размером k будет затронуто не более $2k$ соответствующих вершин и ребер. Итак, давайте сожмем весь граф и избавимся от бесполезных вершин и ребер.

  Теперь мы можем легко работать с частичными запросами. Точно так же, как мы делали это в подзадаче 2, давайте разберемся с событиями прямолинейно одно за другим и раскрасим все компоненты с помощью DFS. Затем мы пройдемся по всем частичным запросам, и если запрос охватывает текущее событие, мы соответствующим образом обновим ответ на него.

Эта часть будет работать за $O(k^2)$ за блок. И запрос будет считаться частичным не более чем в двух блоках, поэтому вклад запроса в общую временную сложность составит всего $O(k)$.

Теперь займемся запросами, полностью покрывающими блок. Потенциально их может быть много на каждый блок. Но вот что важно: после того, как мы сжали граф, у нас осталось только $O(k)$ вершин, поэтому нас интересует только $O(k^2)$ пар вершин. Таким образом, если мы будем поддерживать двумерный массив $ans_{u,v}$ для каждой пары (u, v) и вычислять его так же, как мы это делали с частичными запросами, мы получим $O(k^3)$ временная сложность на блок.

Теперь давайте подумаем об оптимальном k. Всего у нас будет n/k блоков, в каждом из которых нам нужно будет сделать около $O(n + m + q)$, чтобы сжать граф и найти релевантные запросы. Внутри блока мы будем выполнять $O(k^3)$ операций. Итак, если мы попытаемся сделать две стороны равными, мы получим, что оптимальное значение k должно быть около $(n + m + q)^{1/3}$.

Таким образом, общая временная сложность, которую мы получаем, составляет $O(n+m+q)^{5/3}$, что должно быть достаточно для выполнения этой подзадачи.

- **Нет дополнительных ограничении.** 21 **баллов.**

  На самом деле, решение предыдущей подзадачи может быть оптимизировано до такой степени, что оно пройдет даже эту подзадачу. Но здесь мы обсудим лучшее решение.

  Узким местом нашего решения была часть $O(k^3)$, где мы вычисляем $ans_{u,v}$ для всех пар (u, v) в блоке. Можем ли мы сделать это быстрее? К счастью, да.

  Заметим, что у нас, по сути, та же исходная задача, но на этот раз $n = O(k)$, $m = O(k)$ и $q = O(k^2)$ со всеми запросами во всем диапазоне также.

  Так можем ли мы применить то же самое решение к этой подзадаче? Возьмем блок $t = \sqrt{k}$ и снова найдем блоки размером t? Теоретически мы можем, но, вероятно, это будет медленнее, чем $O(k^3)$, о котором говорилось выше.

  Но вместо этого мы можем проанализировать это и выяснить, что для этой подзадачи оптимальный размер блока t будет равен $n/2$. Другими словами, эту подзадачу оптимально решать по алгоритму «разделяй и властвуй». Разделить все события прямо посередине, сжать все вершины и ребра с обеих сторон, решить рекурсивно и объединить ответы.

  Проанализируем временную сложность. Имеем $T(n) = 2T(n/2) + O(n^2)$. Если расширить, то получим $T(n) = O(n^2 + 2 \times (n/2)^2 + 4 \times (n/4)^2 + \ldots) = O(n^2 \times (1 + 1/2 + 1/4 + \ldots) = O(n^2)$, поэтому при $n = k$ мы сможем решить задачу за $O(k^2)$, что именно то, что мы искали.

  Взяв за k примерно $O(\sqrt{n + m + q})$, мы получим решение $O((n + m + q)\sqrt{n + m + q})$, хотя и с очень высокой константой.