Editorial

Tasks, test data and solutions were prepared by: Gabrijel Jabrošić, Marin Kišić, Pavel Kliska, Adrian Satja Kurdija, Ivan Paljak, Stjepan Požgaj, Daniel Paleka and Paula Vidas. Implementation examples are given in attached source code files.

Task Patkice

Prepared by: Pavel Kliska
Necessary skills: traversing a cell matrix, simulation

We save the input as a matrix mat. Let the starting island have coordinates (ox, oy), so $math[ox][oy] =$ `'o'`. We will try to send the ducks to each of the four directions, yielding four paths. FOr each of these paths, we count through how many cells it passes before reaching `'x'`, or say the distance is infinite if the path ends in `'.'` or `'o'`.

Implementation: we run the following algorithm four times, starting from $(ox + 1, oy)$, $(ox - 1, oy)$, $(ox, oy + 1)$ and $(ox, oy - 1)$. We maintain the current duck position as (px, py), and the length of the path in a variable d initialized to 0. If $mat[px][py]$ is in `'<>^v'`, increment d and move (px, py) in the corresponding direction. (We can use `dict` or `std::map`, mapping the ordered pairs which describe the direction to characters, e.g. `{'>': (0, 1)}`). If $mat[px][py]$ equals `'o'` or `'.'`, set $d = \infty$ and stop processing this path. We are done if $mat[px][py]$ equals `'x'`.

After the previous part, we have the distances d_N, d_E, d_W i d_S. If all $d_i = \infty$, there is no solution and we print `:(`.

Else, we print `:)` and the letter i for which d_i is minimal, breaking ties alphabetically.

Task Bajka

Prepared by: Gabrijel Jambrošić
Necessary skills: dynamic programming

Let us denote the scary word by s, and the favourite word by f.

We use dynamic programming. State $dp[i][j]$ means that we are on the i-th position in the scary word, and we want to write down the j-th letter of the favourite word. The transition is given by $dp[i][j] = \min(dp[k][j + 1] + c)$, where c is the time cost for the transition. The transition can be done for each pair of positions (i, k) such that $s[i] = f[j]$, $s[k] = f[j + 1]$, and either $s[k - 1]$ or $s[k + 1]$ exists and is equal to $f[j]$. We first move from position i to either position $k - 1$ or position $k + 1$, whichever is equal to $f[j]$, using the second kind of move, and then to position k using the first kind of move (so we will write down $f[j + 1]$). The cost c is the sum of costs of the two moves. We need to be careful when both positions $k - 1$ and $k + 1$ contain $f[j]$ and take the one that is closer.

The time complexity is $\mathcal{O}(n^2 m)$. The problem can be solved in $\mathcal{O}(nm)$ complexity, but that is left as an exercise to the reader.

Task 3D Histogram

Prepared by: Marin Kišić i Stjepan Požgaj
Necessary skills: divide-and-conquer, two pointers, segment tree, convex hull

We will solve the problem in complexity $\mathcal{O}(n \log^2 n)$.

Let $f(i, j) = \min(a_i, ..., a_j)$ and $g(i, j) = \min(b_i, ..., b_j)$.

We want to find the interval $[i, j]$ which maximises $f(i, j) \cdot g(i, j) \cdot (j - i + 1)$. Solution is based on *divide-and-conquer* approach: function $solve(lo, hi)$ will find the best interval contained in $[lo, hi]$ that

includes the point mid. It will recursively call $solve(lo, mid-1)$ and $solve(mid+1, hi)$.

Let $[l, r]$ be an interval such that $l \in [lo, mid]$ and $r \in [mid, hi]$. We consider four cases:

1. $f(l,r)$ and $g(l,r)$ are determined by the left half, i.e. $f(l,r) = f(l, mid)$ and $g(l,r) = g(l, mid)$.
2. $f(l,r)$ and $g(l,r)$ are determined by the right half, i.e. $f(l,r) = f(mid, r)$ and $g(l,r) = g(mid, r)$.
3. $f(l,r)$ is determined by the left, and $g(l,r)$ by the right half.
4. $f(l,r)$ is determined by the right, and $g(l,r)$ by the left half.

First and second cases are analogous, and so are third and fourth, so we will describe the solution for the first and the third case.

Case 1: $f(l,r)$ i $g(l,r)$ are determined by the left half.

Note that this implies $f(mid, r) \geq f(l, mid)$ i $g(mid, r) \geq g(l, mid)$.

This case can easily be solved using the *two pointers* method in complexity $\mathcal{O}(hi - lo)$, i.e. $\mathcal{O}(n \log n)$ in total.

Case 3: $f(l,r)$ is determined by the left, and $g(l,r)$ by the right half.

Note that this implies $f(mid, r) \geq f(l, mid)$ i $g(mid, r) \leq g(l, mid)$.

This case is much more complex, and it can also be solved in $\mathcal{O}(n \log^2 n)$ total complexity. We will first describe a $\mathcal{O}(n \log^3 n)$ solution, and then optimise it.

For some l, let's denote the interval of possible r-s by $[a(l), b(l)]$. Borders $a(l)$ and $b(l)$ can again be found using the two pointers method.

We have

$$\begin{aligned} & f(l, mid) \cdot g(mid, r) \cdot (r - l + 1) = \\ &= f(l, mid) \cdot (-l + 1) \cdot g(mid, r) + f(l, mid) \cdot g(mid, r) \cdot r = \\ &= \langle (f(l, mid) \cdot (-l + 1), f(l, mid)), (g(mid, r), g(mid, r) \cdot r) \rangle, \end{aligned}$$

where $\langle (a, b), (c, d) \rangle$ represents the dot product of vectors (a, b) and (c, d). Note that the first vector $((l, mid) \cdot (-l + 1), f(l, mid))$ depends only on l, and the second vector $(g(mid, r), g(mid, r) \cdot r)$ depends only on r.

For a fixed l, we want to find the $r \in [a(l), b(r)]$ that maximises the dot product. If r was not limited to that interval, we could calculate the convex hull of the set of second points, and use ternary search to find the optimal r for each l. But now, we can sort the points by (for example) the first coordinate and build a segment tree that has the convex hull of the corresponding points in each node. So, for each l we will look at $\mathcal{O}(\log n)$ nodes in the segment tree, and do a ternary search on the hull in each node in $\mathcal{O}(\log n)$ complexity. This gives us the total complexity of $\mathcal{O}(n \log^3 n)$, because we also need to count in the divide-and-conquer.

Now we want to get rid of the binary search. Notice that points $f(l, mid) \cdot (-l+1, 1)$ move counterclockwise as l increases, and so the optimal point on the hulll will also move counterclockwise! For each node in the segment tree, we can answer its queries in amortised linear complexity, by keeping a pointer to the optimal point for the last query. This gives us total complexity of $\mathcal{O}(n \log^2 n)$,

Task Papričice

Prepared by: Paula Vidas and Daniel Paleka
Necessary skills: depth-first search, `std::set`

The first subtask can be solved with a simple *brute force* algorithm. For every possible choice of strings, we can calculate the size of the components resulting from their cutting, using breadth-first search or depth-first search. The complexity of this solution is $\mathcal{O}(n^3)$.

The second subtask can be solved by a smarter implementation of the previously described algorithm. Fix the first rope we will cut. We got two components. Let's look at one of them and root it in an arbitrary pepper. Using the depth-first search algorithm, we can efficiently calculate the size of subtree of each pepper, i.e. the sizes of the component in which the pepper will be located if we cut the string between it and its parent in the tree. The complexity of this solution is $\mathcal{O}(n^2)$.

We will also describe a different solution for the second subtask, which we will then optimize for the third subtask. Root the given tree in any pepper. Choosing two strings to cut can be Choosing the two strings we are going to cut can be viewed as choosing two peppers for which we will cut the strings connecting them to their parents.

For a pepper x, mark the size of its subtree by $S(x)$. We have two cases:

1. The pepper y is an ancestor of pepper x, i.e. pepper y is on the path from x to the root. The sizes of the components are then $S(x)$, $S(y) - S(x)$ and $n - S(y)$. (The case there x is an ancestor of y is similar.)
2. The pepper x is not an ancestor of y, nor y is an ancestor of x. The sizes of the components are then $S(x)$, $S(y)$ i $n - S(x) - S(y)$.

This approach can be implemented directly in the complexity of $\mathcal{O}(n^2)$.

To solve the third subtask, we will optimize this approach. During depth-first search, we will maintain two sets of already processed peppers: peppers that are ancestors of current peppers (set A) and those that are not (set B). When we enter some pepper, we add it to the set A. When we go out, we throw out the current pepper from the set A, and add it to the set B.

Consider the first case for the current pepper x. We are looking for y from the set A for which the numbers $S(x)$, $S(y) - S(x)$ and $n - S(y)$ are "as equal as possible". That is achieved for y for which the value $\left|S(y) - \frac{n+S(x)}{2}\right|$ is minimal (proof left as exercise). Subtree sizes can be kept in a C++ `std::set`, and we can use `set::lower_bound()` to find the values that are closest to $\frac{n+S(x)}{2}$ on both sides. The second case is solved analogously, using the set B.

The complexity of this solution is $\mathcal{O}(n \log n)$. For implementation details, see the official solution.

Task Tenis

Prepared by: Adrian Satja Kurdija
Necessary skills: binary search, ad hoc

We have two types of pairs of players: pairs in which the victory is strict (the best position of the winner (w.r.t. courts) is strictly higher than the best position of the loser), and pairs in which victory is not strict, which means the best positions of the players with respect to courts are equal.

There are at most $\mathcal{O}(n)$ pairs of the latter type, and thus it is enough to check, for each position from 1 to n, whether some such match happens between some of the (at most three) players to whom that position is the best w.r.t. courts. The rest of this description refers to the first set of matches (strict victories).

Each player can be assigned to a basket (bitmask) which indicates the court on which his position is the best. For example, the player whose best court is the third court would go to the basket 001, and the player whose best cours (with an equal rank) are the first and the second court would go to the basket 110.

We look for a solution by fixing one player (let's say his best position is p) and choosing another player's basket. We calculate how many players in that the basket beats him strictly. These are the players from this basket whose best position is better than p. This can be calculated by binary search on a sorted array of the best player positions in the basket. The court of those matches we can determine from the basket itself, i.e. from the position of the 1 in the bitmask. If the bitmask has two or three 1's (equal court winners), we compare positions of a fixed player (the loser) on those courts.