

Algorithms description

Tasks, test cases and solutions prepared by: Frane Kurtović, Gustav Matula, Ivan Katanić and Ante Đerek. The examples of implemented solutions are given in the attached source codes and do not necessarily match every detail of the algorithms described here.

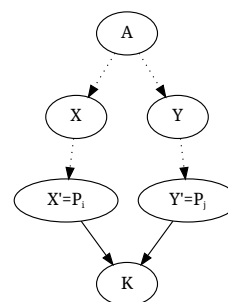
Task: Dijamant

Suggested by: Ante Đerek

Necessary skills: graph, dynamic programming, map data structure

The first step in solving this task is choosing which structure to use to store class names, in other words, to use to assign class names to ordinal numbers. One option is using the `map<string, int>` data structure in C++ programming language. When we recognize a new valid declaration, we assign the next ordinal number to the class name. The first two subtasks come down to checking whether the classes are declared, so this is easily solved now. As an alternative, for example when using the C programming language which doesn't have this data structure, we can first read, save and sort the names of all declared classes. The index of individual classes can then be efficiently found using binary search. In any case, the complexity of one query to the structure is $O(\log n)$.

For the other two subtasks, we need to consider the paths in the directed graph that models the class hierarchy. Let's assume that we are considering a new declaration " $K : P_1 P_2 \dots P_k ;$ " and first try to determine when a diamond A, B, X, Y is formed when adding it. Firstly, evidently it must hold $K = B$. Furthermore, there must exist a path from X to K and from Y to K so let X' and Y' be classes on these paths immediately before K . Since they are the last ones on the paths, classes X' and Y' need to be one of the classes P_1, P_2, \dots, P_k which K inherits. It is crucial to notice that classes X' and Y' cannot be derived one from another – in fact, when X' would, for example, be derived from Y' then A, X, Y, X' would be a diamond that already exists before adding the new declaration.



Therefore, when adding a new declaration, it is sufficient to check the following: Do two classes P_i and P_j exist among classes that K inherits and which are not derived from each other, and that are both derived from some class A ? Since we are processing declarations, for each valid declaration of class K , we will calculate R_K – the set of all ordinal numbers of classes which K is derived from. A new declaration is *not valid* if there exist two classes P_i and P_j among classes that K inherits so that it holds $P_i \notin R_{P_j}$ (P_j is not derived from P_i), $P_j \notin R_{P_i}$ (P_i is not derived from P_j) i $R_{P_i} \cap R_{P_j} \neq \emptyset$ (P_i and P_j are both derived from some class A). This logic can be directly implemented by considering a pair of classes P_i, P_j , which leads to an algorithm of the complexity ($O(n^3)$), but can be optimized enough to score 100 points (for example, by using bitmasks).

The targeted solution efficiently checks whether classes P_i and P_j exist that satisfy the aforementioned conditions. Among other things, it is necessary to notice that if P_a is derived from P_b , then P_b doesn't even need to be considered. For each declaration, we do the following:

1. Sort the classes P_1, P_2, \dots, P_k from the later declarations to the earlier ones.
2. We maintain the set R that corresponds to the union of all R_{P_i} for classes P_i that we have considered thus far. The set R can be stored as an array of n booleans.
3. For each class P_i .
 - (a) If P_i is already in R , ignore it.
 - (b) Otherwise, add all elements from R_{P_i} to R . If we encounter an element already in R while adding, then we found a diamond and the declaration is dismissed.

For each declaration, the number of processing steps is proportional to the total number of elements that are added to R , so it is bound by n . The total complexity is $O(n^2 \log n)$.

Task: Palinilap

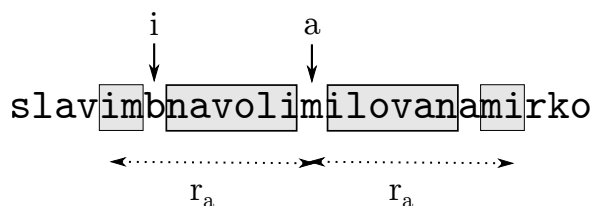
Suggested by: Ivan Katanić

Necessary skills: strings, hashing, binary search, sweep line algorithm

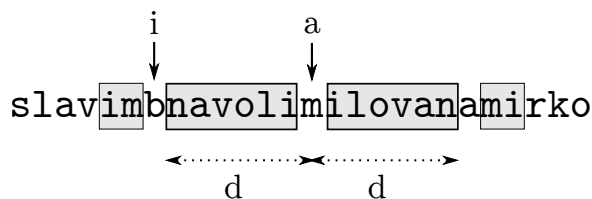
For the sake of simplicity, let's assume that we are only dealing with palindromes of odd length, and that the described solution is easily expanded to the general case. The *center* of a palindrome is the character in its middle, or the position of that character in the original sample. The first step of the solution is, for each position a ($1 \leq a \leq n$), to determine r_a – the half length of the longest palindrome with its center at a (the longest palindrome with its center in a is $w_{a-r_a, a+r_a}$). Notice that the weight of the original sample is equal to the sum of all the numbers $r_a + 1$.

There are multiple efficient ways to calculate all numbers r_a . One option is, for each position a , to determine the value r_a with binary search. In order for this approach to be efficient enough, we need to have a method that can quickly examine whether two arbitrary substrings of sample w , for example $w_{x,y}$ and $w_{u,v}$, are equal. This can be done using the so-called *rolling hash* function (http://wiki.xfer.hr/hashing_stringova/).

When we have calculated all the values r_a , we know the current weight of the sample. In the second step of the solution, we examine how the weight changes if the character at position i is converted to c . After conversion, some palindromes disappear, and new ones appear. First, we calculate how many existing palindromes disappear.



Let's now consider only palindromes with its center at a . Some of them disappear only if the converted character's position is from the interval $[a - r_a, a + r_a]$, and if, for example, the position of conversion i is from the interval $[a - r_a, a]$, then exactly $i - (a - r_a) + 1$ palindromes with centers in a disappear. With the help of this fact, we can implement a sweep line algorithm that calculates in a single pass, for each position, the total number of palindromes that disappear if the character is converted at that position.



We still need to find the number of newly created palindromes for each possible conversion. Let's assume that, after the conversion at position i into character c , a new palindrome appears with its center in a , where $i < a$. Let $d = a - i$. Therefore, the sample $w_{a-d, a+d}$ was not a palindrome before conversion, and now is. Since $w_{a-d, a+d}$ is now a palindrome, so is $w_{a-d+1, a+d-1}$, but this sample hasn't been changed, so it was a palindrome before conversion. Hence, $w_{a-d+1, a+d-1}$ was the longest palindrome with its center in a before conversion, so d is equal to $r_a + 1$. We conclude that each new palindrome with its center in a can appear so that either the character at position $a - r_a - 1$ is converted to the character at position $a + r_a + 1$ or vice versa. Therefore, similarly to how we calculated r_a (binary search with hash function to compare strings), we can calculate the number of new palindromes that appear in each of the two cases.



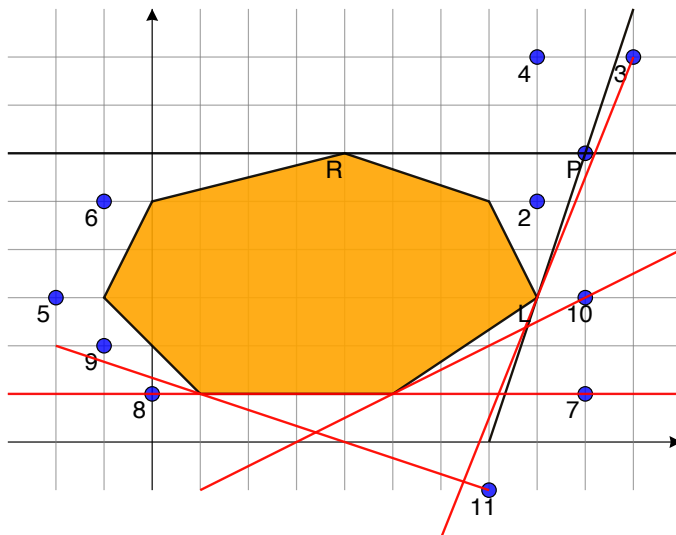
Now we have calculated everything we need in order to know how many palindromes appear for each position i and each new character c , and how many disappear if the character at position i is converted to c , so the task is solved.

Task: Relay

Suggested by: Frane Kurtović

Necessary skills: 2d geometry

Let's denote the position of the first ship with P . The first part of the task is to find the set of points S that are visible from point P – the receiver of the Mayday message. The second part of the problem is to find the set of points S' that are visible from at least one points from set S – the receivers of the Relay message. In the rest of this text, the expression to the left or right of line P is often used and means that, if the observer is at point P and is looking towards points L , something is to the right or to the left of that line. This expression directly corresponds to the geometric primitive ccw that is most often implemented using the vector product. The polygon edges are given counterclockwise, so we will use the expressions *before* or *after* point T , and will be denoted respectively with B_t and A_t .



We determine the points that are visible from P using the left and right tangent, PL and PR . All points to the left of PL and to the right of PR are directly visible. Additionally, points inside of the triangle PRL are directly visible. The right tangent from point P is determined in a way that we find the edge R for which it holds that point P is to the right of side B_rR and to the left of side RA_r . The left tangent is determined in a similar way.

We find points from set S' so that we first find the points that receive the Relay signal from points to the left of PL , then in the same way determine those that receive the Relay signal from points to the right of PR . We will only describe the process for the left tangent. Let K denote the set of points to the left of PL . In the picture, these points are P_3 , P_7 , P_{10} and P_{11} , and the red lines are their left tangents, in the rest of the text just tangents.

Let's observe which area is covered by point T from set K , without it being covered by the starting point P . Let TT' be the tangent that touches the polygon in point T' and let C be the intersection of tangents TT' and PL . This area consists of all points to the left of TT' and to the right of PL , and of all points inside of the triangle CLT' .

Notice that the point which has the tangent that forms a smaller angle with the line PL has an area that is a subset of the area of the point that has the tangent that forms a larger angle. This means that we only need to find the point that has the tangent that forms the largest angle and count the points that are visible from it. The only other problem is how to calculate the tangent to the polygon from every point. If we denote edge L with index 1 and denote other edges clockwise with ascending indices, then the edge in which the tangent from the required point touches the polygon will have the largest index (among points from set K). This is why we will maintain the largest index thus far and increment it with every new point until it becomes the edge in which the tangent from the current point touches the polygon. When we test whether we need to increment the index from point T , we only need to check if the next edge in the polygon is to the left of the length that connects T to the current edge. If it is, then we increment the index.

Regarding implementation, we need to pay attention to the collinearity of points and make sure that the edge points between areas are not counted multiple times. The total complexity of the algorithm is $O(N)$.



For the curious reader: Solve the version of the task where in each step we give two points a and b and ask whether b will receive one of the signals if a transmits the Mayday signal. You are given 100 000 queries, all other limitations stay the same.

Task: Torrent

Suggested by: Frane Kurtović

Necessary skills: dynamic programming, binary search

Let's first solve the simpler problem where there is only one source. Let's denote with $f(i)$ the minimal time needed for a file to spread over the subtree of node i (if we already have the file in node i , but not in the other nodes of the subtree).

Let's denote the children of node i with c_1, c_2, \dots, c_m . If the file is spread over node c_1 first, then c_2 and so on, the total time will be $\max_{j=1}^m \{f(c_j) + j\}$. Therefore, we need to choose the optimal order of sending the file to the children. It's intuitively clear that it is best to first send the file to the node in which the further sending requires the most amount of time, so node c_j that maximizes $f(c_j)$. This is easy to see, if we assume that we have a pair of children that are not in sorted order, in other words, $f(c_j) < f(c_k)$ if $j < k$. Then they potentially contribute to the maximum with $f(c_k) + k$, which is strictly larger than $\max\{f(c_j) + k, f(c_k) + j\}$, so the substitution of these two nodes cannot worsen the total time. Therefore, it is truly optimal to sort the children.

We have the relation: $f(i) = \max_{j=1}^m \{f(c_j) + j\}$, where c_j are sorted descendingly by $f(c_j)$. The direct implementation is of the complexity $O(N \log N)$.

Let's now return to the problem with two sources, a and b . Let's observe the path from a to b . It is clear that, in the optimal spreading of the file on it, there will exist an edge (c, d) such that the file will come to node c from a and to d from b . This is why the optimal spreading would carry out identically even if we erased the edge (c, d) and split the tree into two components. If we knew which edge we need to erase, we would just do it and solve the problem for a single source on both given components. The maximum of these two times would be optimal.

One solution is to try edge after edge on the path from a to b and take the best. The complexity of this algorithm is $O(N^2 \log N)$, because we use the $O(N \log N)$ algorithm to solve the task with a single source of complexity $O(N)$ times.

Let's denote the edge from a to b with numbers from 1 to L . Let's denote $f_a(i)$ as the time required to spread over the component of a if we erased the edge i . Analogously, let's use $f_b(i)$ for the component of b . The solution is then $\min_{i=1}^L \{\max\{f_a(i), f_b(i)\}\}$.

A crucial observation is that, as we traverse from a to b , the component in which node a is in (after erasing an edge) becomes larger, so the total time to spread the file over the component of a can't be reduced. For component of b the analogous observation applies, the time cannot increase.

Therefore, sequence $f_a(i)$ is ascending, and $f_b(i)$ is descending. So, if we have i for which $f_a(i) < f_b(i)$, the lesser maximum can be found only for a larger i . The similar applies for $f_a(i) > f_b(i)$. Therefore, the optimal edge can be found using binary search. The total complexity is then $O(N \log^2 N)$.

For the curious reader: Try to reduce the complexity of the algorithm for a single source. Solve the task if, initially, the file is located on three computers (using the same limitations).