CROATIAN OLYMPIAD IN INFORMATICS 2015
March 28th, 2015
solutions

Task ČVENK	Author: Ante Đerek

In this task, we have to determine the minimum total number of steps necessary in order for all the tourists to meet in the same field of the labyrinth.

According to the image, it seems that the labyrinth is shaped as a tree, and if we set field (0,0) as the root, field (x,y) has a depth of x + y. This isn't difficult to prove:
The case where one of the coordinates is 0 is trivial (father of (x,0) is (x - 1, 0), simmetrical for y).
Let's denote the lowest active bit of a positive integer x with *lobit*(x). Notice that, because x & y == 0, *lobit*(x) != *lobit*(y). Let's assume that *lobit*(x) < *lobit*(y) (the other case is symmetrical).

We can visualize it this way:
x: ???????????0000...010...000
y: ???????????1000...000...000

Let's observe numbers x - 1 and y - 1:
x - 1: ???????????0000...001...111
y - 1: ???????????0111...111...111

Now it is clear that (x - 1) & y == 0 and x & (y - 1) != 0, so the father of field (x, y) in the tree is field (x - 1, y), which matches our expectations. If *lobit*(x) > *lobit*(y), the father is (x, y - 1). Therefore, it really is a tree.
It needs to be noted that something stronger holds. Specifically, for *lobit*(x) < *lobit*(y), (x - z) & y == 0 for z from [0, *lobit*(x)], the argument is similar as for x - 1, and this will be useful later for quick calculation of the kth ancestor.

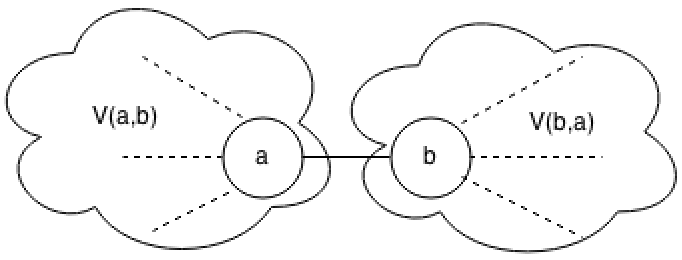
The typical next step is to implement the function *lca*(x1, y1, x2, y2) that finds the lowest common ancestor of fields (x1, y1) and (x2, y2). First, we will implement an auxiliary function *kth_ancestor*(x, y, k) that finds the kth in line ancestor of field (x, y). We can implement it recursively:

```
kth_ancestor(x, y, k):
        for k == 0: return (x, y)
        for x == 0: return (x, y - k)
        for y == 0: return (x - k, y)
        for lobit(x) < lobit(y):
                return kth_ancestor(x - min(lobit(x), k), y, k - min(lobit(x), k))
        for lobit(x) > lobit(y):
                return kth_ancestor(x, y - min(lobit(y), k), k - min(lobit(y), k))
```

Using this function, we can implement the function *lca* using the classical algorithm of jumping on powers of 2, in other words, binary search. For details, consult the official source code.

From now on, we will denote the fields with letters, instead of coordinates.

Notice that when we have the function *lca* we can easily calculate the distance between two nodes *a* and *b* in the tree as *depth*(*a*) + *depth*(*b*) - 2 * *depth*(*lca*(*a*, *b*)).



We are left with finding a node where the tourists will meet in. This is a typical tree problem and generally the idea is this: let's assume that we are located in node *a*. Let node *b* be the neighbour of *a*, let's observe the edge *a* - *b*. Let V(*a*, *b*) be the number of nodes in the subtree seen from the side of node *a* (when we are looking at the edge between *a* and *b*, image!), and V(*b*, *a*) the same for node *b*. Then *b* is a better choice for the meeting if and only if V(*b*, *a*) > V(*a*, *b*). Since V(*a*, *b*) + V(*b*, *a*) = N, it follows V(*b*, *a*) > N / 2. It is clear that node *a* is optimal only if for each of its neighbours *b* it holds V(*b*, *a*) <= N / 2. The reader is left with making sure that this condition is sufficient too.

Let's get back to the tree from the task (remember, it's rooted). We will denote the number of tourists in the subtree of node *a* in our tree with *tourists*(*a*). Now we are looking for node *a* such that *tourists*(*a*) >= N / 2 (so that in the part of the tree above *a* there is no more than N / 2 tourists) and *tourists*(*b*) <= N / 2 for each child *b* of *a*. If we take an initial node, we can easily use binary search and the function *kth_ancestor* to find the first ancestor *a* such that *tourists*(*a*) >= N / 2. If all of its children have less than or equal to N / 2 tourists in their subtrees, we have found the solution. Checking of the number of tourists in a subtree can be done in O(N * log(C)) by examining for each node with a tourist if it's a descendant of the node we're considering (by calling the function *kth_ancestor* with the depth difference). Given the fact that in the subtree of the optimal node there are at least N / 2 tourists, we can randomly choose one of N nodes with tourists as the initial node for binary search. The expected number of initial nodes we will need to check is 2.

After we find the optimal node, we are left with summing up the distances of tourists to our node.

The complexity of function *kth_ancestor* is O(log(C)), where C is the maximal value of coordinates of a field, so the complexity of finding the optimal node is O(N * log(C)^2), log(C) for binary search and N * log(C) to check the node (and the expected number of tries is 2).

The complexity of function *lca* is O(log(C)^2), log(C) for binary search, log(C) to call function *kth_ancestor*. We have to call it N times in order to calculate all distances, so the complexity of this part, as well as the total algorithm complexity, is O(N * log(C)^2).

Task KOVANICE	Author: Ante Đerek

Let’s first imagine there isn’t any weighings in which both sides are equal. We will construct a graph where each weighing a < b creates an edge a -> b. The constructed graph is a directed acyclic graph (DAG), because if we had cycles, that would mean that the test data isn’t consistent.

It is clear that this graph cannot have a chain longer than N because that would mean that there are more than N types of coins. The coins located on the chain of length N have their weight determined unambiguously and it is known that the first coin in the chain is K1, the second K2 and so on until the N^{th} coin which is KN. The coins not located in any chain of length N cannot have their weight unambiguously determined because that means there are more types of coins than the number of coins in the chain, and therefore there are more possible layouts satisfying the conditions from the chain.

Now it is also possible to process the weighings in which both sides are equal. All coins of equal weight can be combined in one node and run the previously described algorithm on this graph. It is necessary to remember which coins are in which node so we can output the result for each of them.

Task OGLEDALA	**Author: Ivan Katanić**

The first subtask can be solved so that for all **M** fields we calculate when they will be occupied and then easily answer queries. This can be simply done in time complexity of O(**N** + **M** lg **M** + **Q**) and space complexity O(**M**).

The second subtask has too many washbasins to apply this approach. However, the number $\mathbf{B_Q}$ is small so we can simulate the arrival of all the ladies up to lady $\mathbf{B_Q}$. When there are **X** ladies occupying the washbasins, there are at most **X**+1 segments of continuous unoccupied washbasins. These segments can be represented as ordered pairs (beginning, length) and keep them in a priority queue appropriately sorted so that the peak is always the leftmost longest segment. If that segment is represented by pair (**P**, **D**), the next lady coming will occupy washbasin **P**+(**D**-1)/2 (integer division) and new segments (**P**, (**D**-1)/2) and (**P**+(**D**-1)/2 + 1, **D**/2) will form. For the described procedure, it takes **N** + 2 * $\mathbf{B_Q}$ pushing into the priority queue and $\mathbf{B_Q}$ popping so the time complexity of the algorithm is O(**N** + $\mathbf{B_Q}$ lg (**N**+$\mathbf{B_Q}$)), and memory O(**N** + $\mathbf{B_Q}$).

The third subtask requires a different approach because we can’t simulate the arrivals of all ladies. Can we at least determine the size of the block which the Xth lady is coming to, for an arbitrary X? Of course we can!

To begin with, we need to calculate how many blocks of which size will form during the arrival of all the ladies to the restroom. When we know that, it’s easy to find the answer to our question because the ladies choose the blocks in descending order.

How to calculate how many blocks will form of which size? Let’s observe the state of the washbasins after the arrival of the first N ladies:

```
.X.....X..
```

In the upper example we have segments of empty washbasins of length 1, 5 and 2. Let’s call these segments **main segments**. As the ladies arrive, they will be split into smaller and smaller segments. It is crucial to notice that the main segment of size **D** will split into at most 2 lg **D** segments of different lengths. For example, for D = 8, we have 4 different lengths:

```
      8
    /   \
   3     4
 / \    / \
1   1  1   2
            \
             1
```

For each main segment, we can easily calculate all the different segment lengths that will appear when it's splitting and how many of them there are. When we calculate this for each main segment, we can construct an array of triplets (length, quantity, of which main segment) that describe all the segment lengths that will ever appear.

If we sort this array descending by the length of segment and from left to right by the main segment, we get even more than we asked. Now we can also determine the length of segment L which the Xth lady will "hop into", but also the main segment which that segment is going to belong to.

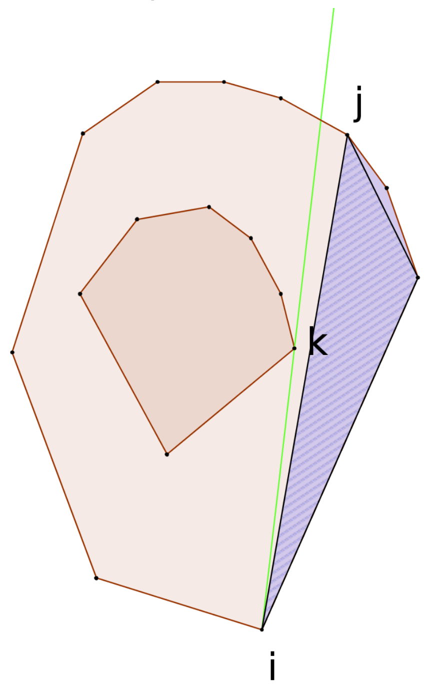
We are left with determining where in the main segment the required segment is going to be located. From the large sorted array we can find out the ordinal number of the required segment (by the time of occurrence) among all the segments of the same length from that main segment. Let's denote its ordinal number with K. Now we can forget about all the other main segments and focus on the division of one main segment.

Let's imagine the first division of the main segment (the arrival of the first lady in it), and it splits into two parts. All segments of length L that will ever appear in the left part will be chosen before all segments of length L from the right part. Therefore, if there are at least K segments of length L in the left part, we discard the right part and repeat the same procedure. If there are less than K, we denote that number with P, discard the left part and look for the (K-P)th segment of length L in the right part. We end the procedure when the segment we're splitting up becomes of length L.

The counting of segments of length L that will form from a segment can be done in the complexity of O(log M) dividing it in the same way as the main segments before we inserted them in the large array. Since in each step we split the segment into two equal parts and discard one of them, the algorithm will complete in O(log M) so the total complexity of this solution is O(N log M log N + Q log M log M).

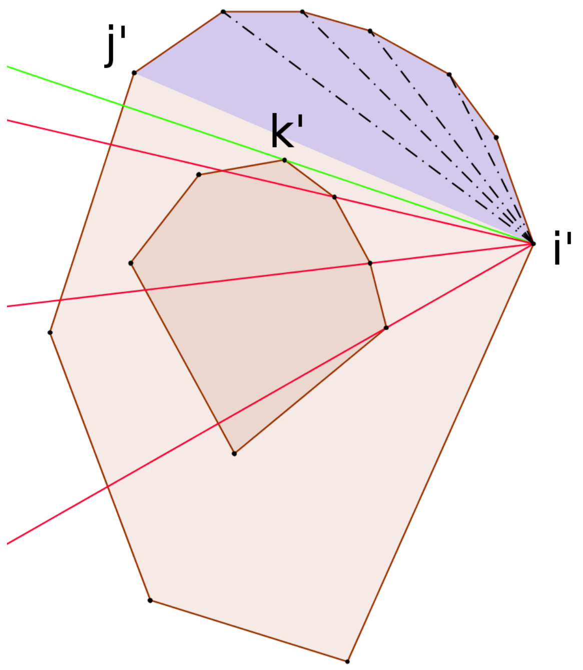
Task SIR	**Author: Gustav Matula**

To begin with, let’s assume that in the interior of the polygon there are more than two points and not all of them are on the same line. We are only interested in the points on the convex hull from the inner set. For a polygon vertex *i* we have the situation as depicted:



The green color marks the tangent in *ccw* (*counter-clockwise*) direction to the inner hull, index *k* marks the belonging hull vertex, and index *j* marks the polygon vertex to which (in *ccw* direction) we can make the best cut from *i*. Notice that this will always be the last vertex (starting from *i*) from the right side of the tangent (looking from *i* facing *k*). The area of the polygon from vertex *i* to *j* is marked with purple and let’s assume it’s P(*i*, *j*). If we take the maximum of these areas for each *i* (vertex *j* is determined with vertex *i*), we will get the solution.

Let’s denote with *i*’ the vertex adjacent to vertex *i* in *ccw* direction. We are interested in vertex *k*’ of the inner hull through which the tangent from *i*’ is going to pass, and vertex *j*’, the optimal vertex for *i*’. Let’s observe the following image:



Vertex *k′* can be found so that we move in *ccw* direction on the inner hull as long as the next vertex is on the right side of the line from *i′* to the current vertex (if we're looking from *i′* facing the current vertex). In the image, the vertices for which the next vertex is better are marked with red.

We can find vertex *j′* so that we move in *ccw* direction on the polygon as long as the next vertex is on the right side of the line (*i′, k′*), from *i′* facing *k′*.

Notice that in this procedure we can keep track of the current area. When we move from *i* to *i′*, we subtract the area of the triangle (*i*, *i′*, *j*) (marked in the first image). When we look index *j′*, we add the area of the corresponding triangle in each move (those triangles are separated using dotted lines in the second image).

The complexity of finding the convex hull of an inner set of points is O(M log M). After that, finding the optimal cut is O(N + H), where H is the number of points on the hull, or O(N + M) in worst case scenario. Index *i* will go through all the points of the outer polygon. Finding the indices *j* and *k* can be complex in one transition, but in total it's amortised because both indices can visit a point at most once (*j* on the polygon, *k* on the inner hull).