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.

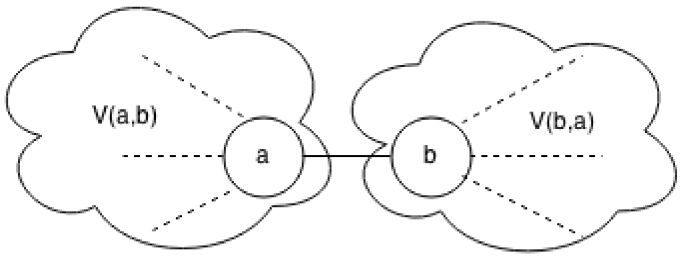
The typical next step is to implement the function *lca*(x1, y1, x2, y2) that finds the lowest common ancestor of fields (x1, y1) and (x2, y2). First, we will implement an auxiliary function *kth_ancestor*(x, y, k) that finds the kth in line ancestor of field (x, y). We can implement it recursively:

```
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).