Analysis of problem `locmin`

Tags: interactive problem, directed acyclic graph, binary search

For brevity, from now on we will say only *minimum point* instead of *local minimum point*.

Initial thoughts - 10 points

For 100 points, we must, with very few queries (under 2024 compared to a total of 250 000 cells), find a *minimum point*, which means that there must be a good way to understand that somewhere there must be a cell with this property. Moreover, in fact, the constraint that it is guaranteed that there is at least one *minimum point* is unnecessary and it can be proven that it follows from the definition and the fact that there are no neighboring cells in the table with equal values. Intuitively, this means that if we consider a given cell and it is not a *minimum point*, then there must be a neighboring cell with a smaller value and we can make the same reasoning for this neighboring cell and so little by little we will indeed reach a *minimum point*. Formally, if we represent the cells of the table as vertices in a graph and we have edges between neighboring cells from the cell with the larger value to the cell with the smaller value, then this graph will be a *DAG* (directed acyclic graph) and as such it will certainly have vertices with no outgoing edges and these are exactly the *minimum points*. This also gives us some non-trivial solution to the problem - we choose an arbitrary cell and follow neighboring cells with smaller value until we find a *minimum point*. Unfortunately, such a solution in the worst case is not better than the trivial checking of all cells in the table and therefore receives only the guaranteed minimum number of points for a correct solution - 10 points. Later we will show a better version of this solution.

One implementation detail that is used in all solutions is related to not repeating queries. For this purpose it is most convenient to make our own function which calls `value`, but before that checks whether we already know the value of the respective cell, by keeping a table of already known values.

Achieved number of queries (in the worst case): NM

Solution for the second subtask - 21 points

We use the above reasoning, as well as the fact that the table is actually just an array. Standardly in interactive problems, binary search is often applied and this is no exception. Especially this subtask practically ”screams”to use such an approach - we have an unknown array and we can ask queries about the value of some cell. Let us first ask queries about the cell in the middle and its two neighbors to determine whether it is a *minimum point* or not. If it is a *minimum point* we are done. If it is not a *minimum point*, then it means we have a neighbor with a smaller value. From the reasoning earlier, this means that in this half of the array we must have a *minimum point*. Thus we will be able to consider only the left or only the right part of the array and apply the same reasoning. Since each time we reduce the size of the array by half, then after roughly $\log_2 M$ queries we will reach 1 cell, which must be a *minimum point*.

Achieved number of queries (in the worst case): $3.\lceil \log_2 M \rceil$

Randomized solution for the third subtask - about 20 points

We can improve the initial approach, where we start from a single cell chosen arbitrarily, by trying to start from several cells so that some of them are quite close to a *minimum point*. This is a good idea, but how do we know which cell is really close to a *minimum point*? We can use the value as a guide, i.e. at the beginning we examine many random cells in the table and then we start from the one with the minimum value to search for a *minimum point*. Obviously such a point must exist, because we will eventually reach a vertex with no outgoing edges (in the graph of the table). This idea was proposed and implemented by *Boris Mihov*.

It turns out that it is quite difficult to create tests against such a solution, because it is guaranteed that it will hit some cell which is close to the actual *minimum point* (in most tests there is only one *minimum point*, to make them stronger). It can be computed mathematically that, for example, if initially we query 1000 random cells, then the probability that we are not within radius 15 of the single *minimum point* is less than 3%, i.e. it is highly likely that if we try all cells within radius 15 of the one with minimum value, we will find a *minimum point*. Therefore the grader of the problem (the jury's program) is adaptive and tries in the end to place the *minimum point* as far as possible from all queries. This grader became very good and even helped discover a small bug in the author's solution.

The adaptive grader made it so that this randomized solution, even with a random seed and with additional tricks (for example, when there are several smaller neighbors, choosing one of them at random), does not score more than 20 points.

Partial solution for the third subtask - about 56 points

We can relatively easily adapt the idea from the second subtask from the one-dimensional case to the two-dimensional case in the general problem. Let us try to determine whether in the upper part of the table there is a *minimum point*, or there is a *minimum point* in the lower part. We fix the middle row and find all cells in it. Suppose that none of them is a *minimum point*. We want to be sure in some way that this row will serve as a boundary and that we must have a *minimum point* either above or below. But when might we need to cross this row while searching for a *minimum point*? Only if there is a cell in it with a smaller value than some other. Thus, if we consider the cell with minimum value in the row (which we know is not a *minimum point*), then above or below there must be a cell with a smaller value and we have the important property that if we descend to it to search for the *minimum point*, then we cannot cross this fixed row again later, because all cells in it will have larger values. Thus depending on where we have a cell with smaller value than the minimum of the row, we will be left with only the upper or only the lower half of the table, in which we are guaranteed to have a *minimum point*. Recursively we do the same and thus we again obtain a solution similar to binary search but over the interval of rows. It is also important that when we reach a single row we do not try to apply the approach from the second subtask, because there we only had neighbors to the left and right, while here some cell may not have a smaller neighbor to the left or right but may not be a *minimum point* because it has a smaller cell above or below.

If we apply this idea directly, the number of queries will be $3N \times \lceil \log_2 M \rceil$, which is too many queries. We can notice that it is not necessary to check for all cells in the middle row whether they are *minimum points* - the only important thing is to perform this check for the cell with minimum value in the row. Even if some other cell happens to be a *minimum point*, we will still certainly find a *minimum point*. Otherwise, we would waste too many queries checking the values of the upper and lower rows relative to the middle, to determine whether we accidentally have a *minimum point* in it (and in tests with a single *minimum point* this will obviously only add extra queries).

Achieved number of queries (in the worst case): $N \times \lceil \log_2 M \rceil$

Solution for the third subtask - 100 points

It remains only to further improve the search - to adapt the one-dimensional idea a bit better to the two-dimensional case. Something important that we saw in the previous solution is that if we divide the table into some regions, and then find the minimum cell along the boundary of the regions, then it will indicate exactly in which region there must be a *minimum point*. Previously we made the division simply by the middle row. Now let us divide into four regions - we make a cross consisting of the middle row and the middle column (so that the region in which we have a *minimum point* decreases both in number of rows and number of columns). It turns out that this is the most optimal partition (similar to the reasoning why binary search is optimal). Each time we find the minimum cell in the cross and, if it is not a *minimum point*, then we will know in which of the four quadrants to

go to search for a *minimum point* (the minimum cell cannot be in the center of the cross, because it has minimum value and is not a *minimum point*). Thus initially we will query the values in $N + M - 1$ (the number of cells in the cross)$+2$ (the remaining two neighbors of the minimum cell), or in total $N + M + 1$ cells. At the next step we will query the values in $N/2 + M/2 + 1$ cells, and so on. Overall the queries in the worst case (or rather if we are not very lucky) will be roughly: $(N + N/2 + N/4 + ... + 1) + (M + M/2 + M/4 + ... + 1) + min(\log_2 N, \log_2 M) \approx 2N + 2M + min(\log_2 N, \log_2 M)$ (it is well known that $N + N/2 + N/4 + ... + 1 = N \times (1 + 1/2 + 1/4 + ...) \approx N.2$).

Achieved number of queries (in the worst case): roughly $2N + 2M + min(\log_2 N, \log_2 M)$ or in practice on the tests 1990.

Idea: Rumen Mihov
Implementation and solutions: Iliyan Yordanov