Editorial

Tasks, test data and solutions were prepared by: Vito Anić, Fran Babić, Toni Brajko, and Petar Sruk.

Implementation examples are given in attached source code files.

Task Šah

Prepared by: Fran Babić
Necessary skills: two-dimensional array

To achieve full points on the task, it was sufficient to maintain a set of attacked squares in a matrix, i.e., a two-dimensional array. Each type of piece was handled in a separate manner.

Rooks were handled by iterating over all cells on the board that share the same row as the rook and then over all cells that share the same column as the rook. This was sufficient for the first subtask.

For queens, we did the same as for rooks in terms of rows and columns, but we needed a different approach for diagonals. Notice that for a square (x, y), another square (x', y') is on the same diagonal if and only if $x + y = x' + y'$ or $x - y = x' - y'$. By iterating over all such squares, we marked all attacked squares on the diagonals. This solves the first two subtasks.

For knights, we iterated over all possible moves, i.e., positions that the knight can jump to. The simplest characterization of a knight's move is that the absolute value of the product of the row and column offsets must be equal to 2. The only thing to watch for here are the boundaries of the board, i.e., the matrix.

Details of the implementation can be seen in the provided solution.

Task Benzinska

Prepared by: Toni Brajko
Necessary skills: sorting, set

The key idea of the problem is that for a given restaurant, we do not need to decide immediately whether Mr. Malnar will dine there when we reach it; instead, we can make this decision later.

At the start, we are at $x = 0$ with D energy available. This means that we can reach all restaurants i for which $x_i \leq D$. If $D < X$, we must use at least one of these restaurants (otherwise, Mr. Malnar will pass out after $x = D$). It is easy to show that we will always want to choose the restaurant with the largest y_i. This leads us to the following algorithm:

1. Let $R = D$ be the current "reach", i.e., the furthest position that Mr. Malnar can reach with the current selection of restaurants, and let S be the set of all unused restaurants within the reach ($x_i \leq R$).
2. If $R \geq X$, Mr. Malnar can reach Čakovec with the current selection of restaurants, so we terminate the algorithm.
3. Otherwise, Mr. Malnar cannot reach Čakovec. From the set S of reachable restaurants, we select the one with the largest y_i and remove it from the set. The new reach is $R' = R + y_k$, where k is the selected restaurant.
4. Add to the set S all restaurants that become reachable with the new reach ($R < x_i \leq R'$).
5. Repeat step 2.

A naive implementation of this algorithm is sufficient to solve the second subtask. Full points can be achieved if we implement the algorithm using a `set` (or a priority queue) for fast insertion, removal,

and retrieval of the largest element. To handle the addition of new elements to the set, we first sort all restaurants by their x_i values. For implementation details, see the attached source code.

Task Xor

Prepared by: Fran Babić and Toni Brajko
Necessary skills: bitwise operations, two pointers

The first subtask can be solved with a naive implementation of the problem in $O(n^2)$.

For the second subtask, we iterate over every possible sum of two elements from the array. Let us fix a sum s; we want to know how many pairs (i, j) exist such that $a_i + a_j = s$. Instead of counting pairs of indices, let us fix all x and y such that $x + y = s$ and count how many ways we can sum $a_i = x$ and $a_j = y$ to obtain the sum s. If x and y are different, the number of ways is $f_x \cdot f_y$, where f_i is the number of elements with value i in the array a. If x and y are equal, the number of ways is $\frac{f_i \cdot (f_i+1)}{2}$. Note that the sum s contributes to the solution if and only if it can be obtained in an odd number of ways. The complexity of this approach is $O(m^2)$, where $m = \max_{1 \leq i \leq n} a_i$.

For each bit, we independently calculate whether it will be 0 or 1 in the solution. The following is a description of the algorithm for a fixed bit b.

For each element of the array a_i, all bits at positions greater than b are irrelevant for computing this bit, so we can remove them. The sum of two elements x and y will have 1 at bit b if:

- x and y have the same value at bit b and $x + y \geq 2^b$,
- x and y do not have the same value at bit b and $x + y < 2^b$.

The elements of the array can be divided into two sets: elements that have 0 at bit b and elements that have 1. Let these sets be A and B, respectively. Now, for a given set, we can calculate in $O(\text{size of the set})$ the number of pairs (x, y) from the set such that $x + y \geq 2^b$. This is done using the two-pointer method on the sorted set. Similarly, in $O(n)$, we can calculate the number of pairs (x, y) from different sets such that $x + y < 2^b$. Now we know how many sums have 1 at bit b, so, depending on the parity, we also know the value of bit b in the final solution.

The complexity of this approach is $O(n \cdot \log n \cdot \log m)$. Each bit is calculated separately, and for each, it is necessary to sort the sets A and B.

For all points, we can avoid the slow sorting of sets A and B. Specifically, if we have an array sorted for bit b (ignoring bits greater than b), then in $O(n)$, we can obtain a sorted array for $b + 1$. It is only necessary to place elements with 0 at bit $b + 1$ before elements with 1. The set A will then be some prefix, and the set B a suffix of the array. The final complexity is $O(n \cdot \log m)$. For implementation details, see the attached source code.

The first subtask can be solved with a naive implementation of the problem in $O(n^2)$.

For the second subtask, we iterate over every possible sum of two elements from the array. Let us fix a sum s; we want to know how many pairs (i, j) exist such that $a_i + a_j = s$. Instead of counting pairs of indices, let us fix all x and y such that $x + y = s$ and count how many ways we can sum $a_i = x$ and $a_j = y$ to obtain the sum s. If x and y are different, the number of ways is $f_x \cdot f_y$, where f_i is the number of elements with value i in the array a. If x and y are equal, the number of ways is $\frac{f_i \cdot (f_i+1)}{2}$. Note that the sum s contributes to the solution if and only if it can be obtained in an odd number of ways. The complexity of this approach is $O(m^2)$, where $m = \max_{1 \leq i \leq n} a_i$.

For each bit, we independently calculate whether it will be 0 or 1 in the solution. The following is a description of the algorithm for a fixed bit b.

For each element of the array a_i, all bits at positions greater than b are irrelevant for computing this bit,

so we can remove them. The sum of two elements x and y will have 1 at bit b if:

- x and y have the same value at bit b and $x + y \geq 2^b$,
- x and y do not have the same value at bit b and $x + y < 2^b$.

The elements of the array can be divided into two sets: elements that have 0 at bit b and elements that have 1. Let these sets be A and B, respectively. Now, for a given set, we can calculate in $O(\text{size of the set})$ the number of pairs (x, y) from the set such that $x + y \geq 2^b$. This is done using the two-pointer method on the sorted set. Similarly, in $O(n)$, we can calculate the number of pairs (x, y) from different sets such that $x + y < 2^b$. Now we know how many sums have 1 at bit b, so, depending on the parity, we also know the value of bit b in the final solution.

The complexity of this approach is $O(n \cdot \log n \cdot \log m)$. Each bit is calculated separately, and for each, it is necessary to sort the sets A and B.

For all points, we can avoid the slow sorting of sets A and B. Specifically, if we have an array sorted for bit b (ignoring bits greater than b), then in $O(n)$, we can obtain a sorted array for $b + 1$. It is only necessary to place elements with 0 at bit $b + 1$ before elements with 1. The set A will then be some prefix, and the set B a suffix of the array. The final complexity is $O(n \cdot \log m)$. For implementation details, see the attached source code.

Task Cipele

Prepared by: Vito Anić
Necessary skills: Greedy, Fenwick tree (or Segment tree alternatively)

Unfortunately, the official solution to this problem is incorrect. After the competition, a counterexample was found for which our solution does not output the minimal possible time.

The solutions for the two subtask that are correct described in this explanation.

The solution for the subtask $n = m$ is as follows: We observe that there is enough space in the hallway to put all shoes. Therefore, if Lana needs a pair of shoes more than once, the second time she will not waste time taking them out. When a pair of shoes is taken out of the closet, the extraction time for all shoes located *deeper* than the current ones will decrease by one. Thus, we need to decrease the extraction time of all shoes at positions greater than the currently extracted shoes by one. We will maintain the extraction times of the shoes using a Fenwick tree. When we want to extract a pair of shoes from the closet, we first check how much time is needed for that and then decrease the corresponding value in the Fenwick tree by 1.

The solution for the subtask $m = 0$ is as follows: Since there is no space in the hallway, Lana will have to return the shoes to the closet at the end of each day. Let us analyze what happens to the positions of the shoes in the closet. Assume that we have taken a pair of shoes x out of the closet. At the end of the day, we return them to the top of the closet. We notice that all shoes that were initially located between the start of the closet and the initial position of shoes x will be *deeper* in the closet by one position. All shoes that were initially *deeper* will remain in the same position. Similar to the first idea, we will use a Fenwick tree with an additional small trick. Again, we will keep track of how much time is needed to extract shoes from the closet, but instead of having only n positions in the Fenwick tree, we will have $n + q$ positions. Initially, the shoes will be placed at the last n positions. When we extract a pair of shoes from the closet, we decrease the value in the position where those shoes were located by one. Then we will assign the shoes a new position that no other shoes had at the start of the closet. Since we have $n + q$ positions in the closet, this position will always exist. At this position, we increase the value by 1, thereby obtaining the new state of the closet after that day.

On my behalf and on behalf of the entire scientific committee, I apologize for this mistake.

Task Tura Mačkica

Prepared by: Petar Sruk
Necessary skills: Euler cycle

The task is to choose some undirected edges and direct them in such a way that there is an Euler cycle together with already directed edges. A directed graph has an Euler cycle if and only if the difference between the indegree and outdegree of each vertex is 0 and the graph is connected.

Firstly, notice that if there is an undirected edge connecting a vertex to itself we will never use it in an Euler cycle so the undirected part of the graph is essentially a tree. Let's call the difference between the indegree and outdegree of a vertex a balance of that vertex. We want to ensure that the balance of each vertex is zero. Consider the deepest vertices (with respect to the undirected tree) with a balance of zero. If we were to add and direct their only undirected edge they would no longer have balance equal to zero, so we must not add these edges. The only undirected edge adjacent to the deepest vertices with non-zero balance must be directed in such a way that the balance after that is zero. This is only possible when the balance is 1 or -1. Now that we have uniquely determined which edges connected to the deepest vertices have to be directed and how, we can apply the same procedure on vertices one level higher until we get to the root.

If we have successfully applied the whole procedure we uniquely determined which edges we must direct and how, so to find out whether an Euler cycle exists it is enough to check whether the graph is connected with respect to directed edges, including the newly added ones.

In the case there is no undirected edge connecting a vertex to itself we have a connected graph with one cycle. Note that such a graph can be partitioned into a cycle and subtrees containing exactly one vertex from the cycle. Since the algorithm above considers only balances of vertices, not the directed edges, we can apply the same algorithm to each subtree of this graph. Note that in this case it is not necessary that roots, i.e. the vertices in a cycle, have a balance 0, after a successfully applied algorithm on each subtree.

In the case that vertices in a cycle do have a balance 0 we can check whether the graph is connected with respect to directed edges. If it is not we can add every edge from the cycle in the Euler cycle and direct them in an arbitrary circular direction. After this addition we have not changed the balance of any vertex, but we could have connected the graph with respect to directed edges, so we check the connectedness once again. Note that to keep the balance of each vertex 0 the added edges need to form a cycle so this was the only possibility to connect the graph.

If, however, after applying the balancing algorithm on each subtree the balances of vertices in a cycle are not 0 we have to balance them out too. We do this by choosing a vertex in a cycle with non-zero balance and directing one of its neighbouring edges in such a way that the balance of the vertex is closer to zero. After that we go to the other vertex that was adjacent to the newly added edge and we direct its other edge, if necessary, to make its balance closer to zero. After we decide for each edge in the cycle whether we direct it and how we check if the vertex of each cycle has balance 0 and whether the graph is connected with respect to directed edges. Note that a different choice between the starting two edges adjacent to the starting vertex can lead to a different result, but both choices uniquely determine every other newly directed edge, so we need to check both possibilities and choose the shorter one.

The total time complexity of the algorithm is $O(n+m)$.