



## Editorial

Tasks, test data and solutions were prepared by: Sofija Velkovska, Fran Škarica, Nikola Vujica and Jakov Celin.

Implementation examples are given in attached source code files.

### Task Zombie Apocalypse

Prepared by: Sofija Velkovska

Required knowledge: loop statement (for), decision statement (if), arrays

There are many ways to solve this task, but we will present only one of them. We calculate separately for each zombie whether it reaches the city, and keep track of how many zombies manage to do so. For a fixed zombie, the  $i$ -th zombie, we check for each bomb whether that bomb kills it.

The full range of the  $j$ -th bomb is  $[x_j - r_j, x_j + r_j]$ . However, keep in mind that the actual range of the bomb cannot extend beyond the path, either to the left or to the right. That is, the bomb only destroys zombies that are on the path, i.e. at positions between 1 and  $n$  inclusive. Therefore, we consider the interval  $[\max\{x_j - r_j, 1\}, \min\{x_j + r_j, n\}]$ . This ensures that we do not consider the bomb destroying zombies that have already reached the city or are still in the cave.

Knowing that bomb  $j$  explodes at time  $t_j$ , we can compute the position of the  $i$ -th zombie at that time using the formula  $t_j - i + 1$ . We then check whether the zombie is inside the bomb's effective range at that moment. If this happens for any of the  $k$  bombs, then the zombie does not reach the city.

### Task Tomahawk

Prepared by: Jakov Ceiln

Required knowledge: ad-hoc

By a straightforward simulation of roasting using for loops, we solve the first subtask. The second subtask is solved by using the claim that each roasting increases entire rows or entire columns, and by keeping track, for each row or column, by how much they need to be increased. In the end, we construct the matrix using these two arrays and determine the maximum and minimum temperature in the matrix. Depending on the implementation, using the previously described solution we can also solve the third subtask.

To solve the last two subtasks, it is necessary to observe that the maximum temperature in the matrix must be located in the bottom-left or bottom-right corner of the matrix. We reach a similar conclusion for the minimum temperature in the matrix depending on the parity of the matrix dimension  $n$ . If  $n$  is even, then the minimum temperature in the matrix is located among the two middle columns of the first row. If  $n$  is odd, the minimum temperature in the matrix is located among the three middle columns of the first row of the matrix.

The previously stated claims are easy to prove, and we leave their proofs to the reader as an exercise.

After determining all candidates for the maximum and minimum temperature in the matrix, it is necessary to determine the temperature of each candidate. By iterating through the list of roastings and adding temperature depending on the distance of the cell from a certain edge of the matrix, we can find the exact temperature of a candidate. The complexity of this approach is  $O(q)$ , since for  $O(1)$  candidates we iterate through the list of roastings.



## Task Magija

Prepared by: Jakov Celin

Required knowledge: union find, binary search, hashing, segment tree

It can be observed that if Ivor has learned to swap two disjoint intervals of equal length, then he has actually learned to swap position  $l$  with  $r$ ,  $l + 1$  with  $r + 1$ , and so on. Instead of swapping, we will connect all such pairs of positions. Now, the answer to the query about person  $x$  is simply the minimum and maximum of the component in which  $x$  belongs. A straightforward implementation of this algorithm using DFS or BFS solves the first two subtasks with complexity  $O(nq^2)$ .

If we connect pairs of positions only when they belong to different components, the task can be solved using a union find structure while maintaining the minimum and maximum of each component, with complexity  $O(nq)$ . This algorithm achieves 73 points on the task (or 56 depending on the implementation). The only drawback of this algorithm is iterating over  $O(nq)$  pairs of positions in order to perform a total of  $O(n)$  unions in the worst case.

The idea to achieve full points on the task is to find only the “important” pairs of positions, i.e. those that belong to different components of the graph, and connect only those. Let us check whether there exists an “important” pair by considering the intervals  $[l, l + x]$  and  $[r, r + x]$ . Let  $d_i$  denote the representative of the component of person  $i$  in the graph. If the values in the array  $d$  are identical over the two intervals, then there is no important pair between these two intervals. We compare whether the values in the intervals of array  $d$  are equal using hashing in  $O(1)$  time.

Now we can find the first “important” pair among the intervals using binary search, but merging graph nodes is slightly more demanding due to the need to maintain the array  $d$ . Note that when merging a smaller component into a larger one in union find, we can iterate over all nodes of the smaller component and update their representatives in the array  $d$ . Due to the need for hash queries on intervals of array  $d$  as well as updates to  $d$ , the data structure we need to use is a segment tree or a Fenwick tree. After merging the first important pair, we continue the process until there are no important pairs left among the intervals.

The total time complexity of the algorithm is  $O(n \log^2 n)$ , and the memory complexity is  $O(n)$ .

## Task Sladoled

Prepared by: Jakov Celin

Required knowledge: dynamic programming, bitset, optimizations

Let  $dp(i, j)$  be equal to 1 if it is possible to make a combination with value  $j$  at stand  $i$ , otherwise it is 0. Then adding ice cream with flavor  $k$  at stand  $i$  can be described by iterating over the combination value  $tren$  from  $k$  to the maximum value  $M$  (in the problem  $M = 50000$ ) and performing the transition  $dp(i, tren) = dp(i, tren - k)$ . To output the number of different combinations, we only need to count how many 1s are in the array  $dp(i)$ . This approach solves the first two subtasks and the time complexity of the algorithm is  $O(MQ)$ .

For the full solution, it is necessary to imagine the array  $dp(i)$  as a binary number. Then the transition can be written as  $dp(i) = (dp(i) \ll k)$  and this transition is performed until the array stops changing (this transition is equivalent to adding one scoop, and since we add them infinitely, the transition is performed until the array stops changing). The data structure that allows such transitions and viewing the array as a binary number is called a bitset.

Regardless of the existence of this structure, our code is still not fast enough to achieve full points on the problem, so it must be accelerated using various optimizations. Instead of repeatedly adding one scoop until the array stops changing, we add one scoop, two, four, eight, and so on. We can use this optimization because every number of scoops  $k$  can be represented using some powers of 2 multiplied by  $k$ , i.e., we have shown that the previous methods of adding scoops are equivalent. This solution is much faster than



the previous one, but it is still not enough to achieve full points on the problem.

The final optimization for achieving full points was to stop the process of adding scoops if we want to add a scoop  $x$  to stand  $i$ , and a combination with value  $x$  could already be constructed at that stand. Notice that if we could construct a combination with value  $x$ , then we can also construct all combinations with values that are multiples of  $x$ , i.e., adding a scoop  $x$  would not change the array.

The overall time complexity of the described algorithm is  $O\left(\frac{nM \log M}{64}\right)$ , and the memory complexity is  $O\left(\frac{nM}{64}\right)$ . For implementation details, see the attached official solution.

## Task Tjelesni

Prepared by: Nikola Vujica

Required knowledge: segment tree

For the first subtask, it was sufficient to simulate the process described in the problem by taking all numbers from the segment  $[l, r]$  and placing them into their appropriate positions after sorting. The time complexity of this simulation is  $O(nq \log n)$ .

In the second subtask, it was necessary to observe that after the operation with the largest right endpoint  $r_{\max}$ , none of the remaining operations will change the appearance of the array. Likewise, changes made before the operation with  $r_{\max}$  will not be visible in the final array. Therefore, we can simulate only the operation with  $r_{\max}$  and read off the position of height  $m$ . The time complexity of simulating a single operation is  $O(n \log n)$ .

From the third subtask onward, we no longer track the positions of all heights, but only the position of height  $m$ . Let us consider what happens to height  $m$  during the  $i$ -th operation in the case  $m = 1$ . If the current position of height 1 lies outside the segment  $[l_i, r_i]$ , its position remains unchanged. If height 1 lies inside the segment  $[l_i, r_i]$ , then after the operation it will be placed at position  $l_i$ , since it is certainly the smallest element in the segment. In the case  $m = n$ , the procedure is analogous, except that height  $n$  will be placed at position  $l_i + 1$ , since it is certainly the largest element in the segment. The only exception is the case  $l_i = r_i$ . The time complexity of this solution is  $O(n + q)$ .

For the remaining subtasks, observe that for each number in the array it is only important whether it is smaller or larger than  $m$ . We label all numbers smaller than  $m$  with 0, and all numbers larger than  $m$  with 1. Knowing the current arrangement of zeros and ones in the array is sufficient to determine their arrangement after the operation: the first number in the segment will be 0 (provided that before the operation there exists at least one zero in the segment), the second number will be 1, the third 0, and so on. After a certain position, depending on the number of zeros and ones in the segment before the operation, all values will be the same.

It remains to decide how to treat the number  $m$ . One option is to label it with the value 2. However, to avoid introducing too many special cases, an elegant approach is to label all numbers **greater than or equal to**  $m$  with 1, simulate the process as described, then relabel the numbers so that all values greater than or equal to  $m + 1$  are marked with 1, perform the same simulation once more, and determine the position at which the two resulting arrays differ. This position will be unique and will correspond to the position of the number  $m$ . The time complexity of this approach is  $O(nq)$ .

To obtain an efficient solution for the full problem, this procedure must be optimized using a segment tree. The data structure needs to support range sum queries and range assignment operations. There are several possible implementations; here we describe one that uses separate segment trees for even and odd positions. Initially, zeros and ones are placed in the leaves of the trees. For each operation, we first compute the sum over the segment  $[l_i, r_i]$ , which corresponds to the number of ones in the segment, and then update the values in the segment in both trees. Several cases arise depending on the parity of position  $l_i$  and on the number of ones relative to the length of the segment. For implementation details, see the attached official solution. The overall time complexity of the solution is  $O(n + q \log n)$ .