



## Editorial

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

Implementation examples are given in attached source code files.

### Task Kist

Prepared by: Jakov Celin

Required knowledge: matrices, distance computation in a grid

For the complete solution, we maintain two variables  $x$  and  $y$  representing the row and column of the matrix where we are currently located. Initially, we are in the cell at row and column  $(n + 1)/2$ . If we encounter a letter that moves us in one of the directions, we move in that direction **if** possible, i.e., if after the move we do not leave the matrix. If we encounter a letter that colors some cells, we iterate over the entire matrix and compute the distance of each cell from the cell at row  $x$  and column  $y$ .

The only thing left is to compute the distance between two arbitrary cells of the matrix. Note that the distance between two cells of the matrix is equal to the sum of the row distance and the column distance of those two cells. The reason is that in one step, in four directions, we can get closer to the other cell by exactly 1 row or 1 column. The distance between rows  $a$  and  $b$  is equal to  $|a - b|$ , i.e., the absolute difference of rows  $a$  and  $b$ . Another way to compute row distance is subtracting the smaller row from the larger one. We use the same approach to compute the distance between two columns. For implementation details, see the attached official solution.

### Task Festival

Prepared by: Sofija Velkovska

Necessary skills: dynamic programming

Let's label the candies from largest to smallest with the numbers from 1 to  $n$ , with candy number 1 being the largest and candy number  $n$  being the smallest. If  $k = 1$  all of the candies have to be in one box and candy number 1 has to be first. Any remaining candy can be placed in the second position in the box, so, we have  $n - 1$  options. Similarly, any of the remaining  $n - 2$  candies can be placed in the third position - that's  $n - 2$  options,  $n - 3$  options for the fourth position, ..., 1 option for the last position. In total there are  $1 \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdot \dots \cdot 1$  or  $(n - 1)!$  ways to arrange the candies in the box.

When  $n \leq 10$ , we can just generate all of the partitions of the set  $\{1, 2, \dots, n\}$  and if a partition is made up of exactly  $k$  subsets, we can then calculate the number of ways to order the candies in each of its subset using the same logic that was explained for the  $k = 1$  case. Multiplying the numbers for each subset, we get the answer for the whole partition.

Now, consider we have 2 boxes. We will process the candies one by one from candy number 1 to candy number  $n$ . Let's denote the number of ways to arrange the first  $m$  candies with  $dp(m)$ . Assume we have already calculated the number  $dp(m - 1)$  and we want to calculate  $dp(m)$ . For the  $m$ -th candy, we could have either already arranged the first  $m - 1$  candies into 2 boxes and then added the  $m$ -th one to one of those boxes, or we could have placed the  $m$ -th candy in a box by itself while all of the other ones are in the other box.

In the first case, we could have taken any arrangement of the first  $m - 1$  candies into two boxes and then added the  $m$ -th candy **after** any of the already placed candies. We can't place it before the first candy in both of the boxes because the  $m$ -th candy and any candies that might be possibly added to the box later are smaller than all of the candies currently in the box. This would go against the rule saying that the first candy has to be the largest, so, it isn't possible to place the new candy before the currently first candy in either box leaving us with a total of  $m - 1$  possible positions for it. This can be done for any



initial arrangement of the first  $m - 1$  candies - that's a total of  $(m - 1) \cdot dp(m - 1)$  ways.

In the second case, all of the other  $m - 1$  candies have to be in the other box - this can be done in  $(m - 1)!$  ways as seen in the case where  $k = 1$ . So,  $dp(m) = (m - 1) \cdot dp(m - 1) + (m - 1)!$  for  $m \geq 2$  and  $dp(1) = 0$  (we can't arrange one candy into two boxes).

We can expand this idea onto the case where  $k > 2$ . We will use  $dp(m, p)$  to denote the number of ways to arrange the first  $m$  candies into exactly  $p$  boxes. If we assumed the first  $m - 1$  candies were already arranged using  $p$  boxes and we want to add the new candy into one of these boxes, we could do that in  $(m - 1) \cdot dp(m - 1, p)$  ways. If we want the  $m$ -th candy in a box all by itself, we need to calculate the number of ways we could have placed the previous  $m - 1$  candies in  $p - 1$  boxes - that is exactly the number  $dp(m - 1, p - 1)$ . Therefore, the dynamic programming transition should be  $dp(m, p) = (m - 1) \cdot dp(m - 1, p) + dp(m - 1, p - 1)$ . For the base cases we have  $dp(1, 1) = 1$  (there's one way to arrange one candy in one box),  $dp(0, p) = 0$  for  $p > 0$  (the boxes mustn't be empty according to the rules) and  $dp(m, 0) = 0$  for  $m > 0$  (we can't arrange some positive number of candies in 0 boxes).

## Task Domjenak

Prepared by: Ivan Janjić

Required knowledge: graph theory, bipartite graphs

In graph theory terms, we need to find the longest alternating path with respect to a unique perfect matching in a bipartite graph. From each pair in the perfect matching, select one vertex such that the selected set is independent. Call this set the "left side", and call the vertices we did not select the "right side".

Suppose we have found a perfect matching and momentarily ignore the uniqueness condition. The symmetric difference with some other perfect matching is a union of alternating cycles and isolated vertices.

We can fix the direction in which we traverse alternating cycles. In other words, we decide that when traversing a matching edge, we always go, for example, from the left side to the right. This means that when traversing an edge that is not part of the matching, we go from the right side to the left.

The previous observations motivate the following transformation: direct all edges that are part of the perfect matching to point from the left side to the right, and direct edges that are not part of the matching from the right side to the left. We obtain a directed graph where directed paths/cycles correspond to alternating paths/cycles (with respect to the chosen matching) in the original graph.

Returning to the uniqueness condition, it is easy to observe that the existence of a directed cycle in the graph is equivalent to the existence of at least one additional perfect matching. Conclusion: this directed graph is acyclic.

Finding the longest path in such a graph is a known problem that can be solved in  $O(N + M)$  time. What remains is to find the perfect matching sufficiently fast. Note that in an acyclic directed graph, there exists a vertex with no outgoing edges. This implies the existence of a vertex (on the left side) in the original graph that has only one neighbor. Remove this vertex and its neighbor from the graph and mark them as matched. Repeat the process until all vertices are matched. This algorithm can be implemented in  $O(N + M)$  time. Finally, the total time complexity of the solution is  $O(N + M)$ . Additionally, the longest path can be computed in parallel with finding the perfect matching. For implementation details, see the official solution.



## Task Država

Prepared by: Jakov Celin

Required knowledge: dynamic programming, dp rerooting, complexity analysis

We solve the task by fixing the capital city  $P$  (rooting the tree at node  $P$ ) and counting, for each size, how many states of that size exist with capital  $P$ . Let  $dp(x, k)$  be the number of states of size  $k$  such that the secondary cities are located in the subtree of  $x$ , and the capital of the state is  $x$ . Note that  $P$  is the fixed capital of the states we are counting, but we still treat  $x$  as the capital in the dynamic programming state.

Now consider merging the subtree of child  $a$  with the remainder of the subtree of parent  $b$ . Let the size of the subtree of node  $a$  be  $s_a$ , and the size of the subtree of node  $b$  be  $s_b$ . Then the transitions can be written (ignoring modular arithmetic):

```
for (int i = s[b]; i >= 0; i--) {
    for (int j = s[a]; j >= 1; j--) {
        dp[b][i + j] += dp[b][i] * dp[a][j];
    }
}
```

We can imagine this state merging as choosing some of the  $s_a$  nodes from the subtree of  $a$  and combining them with all possible numbers of nodes from the subtree of  $b$ . Note that the time complexity of this transition is  $O(s_b \cdot s_a)$ .

Let us compute the time complexity of computing the dynamic programming states if we fix the capital  $P$ . Although it seems that the time complexity is  $O(n^3)$ , it is actually  $O(n^2)$ . For complexity proofs, see Section 7 of the [blog](#) and the solution to the problem [Džumbus](#). The total time complexity of this solution is  $O(n^3)$  because the complexity is  $O(n^2)$  per fixed capital. This solution was fast enough to solve the first 3 subtasks.

We solve the last subtask using the dp rerooting technique. Note that when "moving" the capital  $P$  to one of its children  $a$ , only the values of the states at nodes  $P$  and  $a$  change. Instead of merging the subtree of the child into the parent, we now need to cut it off and merge the parent into the child's subtree.

Let  $s_a$  be the size of the subtree of the child to which we move the capital  $P$ . Then  $s_b = n - s_a$ . The time complexity of the "new" merging and cutting operations can be analyzed similarly to the time complexity in the previous example. The merging procedure is identical to the one in the previously shown code section, and the cutting procedure is achieved by reversing the merging operations (see the code for implementation details). The complexities of both procedures are  $O(s_b \cdot s_a)$ , i.e.,  $O((n - s_a) \cdot s_a)$ .

Note that for the edge connecting nodes  $P$  and  $a$ , the value  $(n - s_a) \cdot s_a$  is equal to the number of paths in the tree that pass through that edge. Thus, the total complexity of the described solution is equal to the sum of distances of all ordered pairs of nodes in the tree. From the problem statement, we know that the distance between any two nodes is less than or equal to 36 (define this value as  $d$ ). It follows that the total time complexity of the solution is  $O(n^2 \cdot d)$ .



## Task Ravnalo

Prepared by: Jakov Celin

Required knowledge: working with fractions

We need to compute the minimum number of segments needed to draw the wall described in the task.

Note that the minimum number of segments is obtained using a simple strategy: extend all segments from the previous column that can be extended into the current column, and start drawing the remaining ones at the beginning of the current column.

Let  $a_i$  be the height of the  $i$ -th column, and  $b_i$  the number of parts it is divided into. Then the  $i$ -th column consists of  $b_i + 1$  segments, where the  $j$ -th ( $0 \leq j \leq b_i$ ) is located at height  $\frac{a_i}{b_i} \cdot j$ , i.e., the vertical distance between consecutive segments in the column is  $\frac{a_i}{b_i}$ . Define  $c_i = \frac{a_i}{b_i}$ .

Now we only need to find the number of segments that can be extended between two consecutive columns of the wall (call them  $i$  and  $i + 1$ ) using fractions  $c_i$  and  $c_{i+1}$ . One way to find the intersection of segments is to find the "least common multiple" of two fractions by taking the ratio of the least common multiple of the numerators and the greatest common divisor of the denominators. Another way is to set the fractions  $c_i$  and  $c_{i+1}$  in proportion, express the segments forming the intersection using that proportion, and compute their number.

Both methods compute the number of intersecting segments in  $O(\log M)$ , where  $M$  represents the maximum value in the arrays  $a$  and  $b$ . The total time complexity of the solution is  $O(n \log M)$ .