Editorial

Tasks, test data and solutions were prepared by: Fabijan Bošnjak, Nikola Dmitrović, Karlo Franić, Marin Kišić, Ivan Paljak i Stjepan Požgaj. Implementation examples are given in attached source code files.

Task: Datum

Suggested by: Karlo Franić
Necessary skills: palindrome check, ad-hoc

For the first subtask it was enough to take the first two characters of the date and increase that number by 1 until we reach a palindrome.

For the second subtask we can use the same approach as for the first one, but we need to take additional care when we enter a new month.

For the third subtask we can use the same approach as for the first two, but we need to take additional care when we enter a new year.

In order to score all points it was important to note that the number of palindromic dates in the given form is relatively small, 366 in total. You could simply find these dates and store them in an array. For each date in the input you can traverse through all palindromic dates and output the smallest one that comes after it.

The time complexity is $\mathcal{O}(NK)$, where K represents the number of palindromic dates. The task can also be solved in $\mathcal{O}(N \log K)$, but we will leave that solution as an exercise to the reader.

Task: Birmingham

Suggested by: Marin Kišić
Necessary skills: elementary graph theory, breadth first seach (BFS)

In order to score half of the points, we must first determine an array `dist[a][b]` which stores the shortest distances between pairs of nodes. We can do that by starting BFS BFS from each node towards all other nodes. After that, we can do a special BFS from starting nodes that expands in the following way: if we are currently in node a with distance d, then we add all nodes to the queue which are not yet visited and are distant from a by $\leq (d-1) \cdot K$. We can find these nodes by traversing `dist[a]`.

To score all points it was enough to conclude that, if we know the distance from node x to some of the starting nodes, then we can deduce the solution for that node (either using a formula or a simple simulation). Distance from each node to the set of the starting nodes can be determined using BFS with all starting nodes being initially emplaced in our queue.

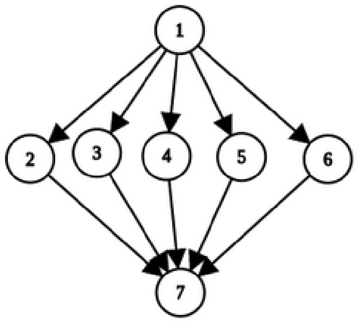
Task: Konstrukcija

Suggested by: Stjepan Požgaj
Necessary skills: ad-hoc

We are asked to construct a graph with N nodes and M edges such that $tns(1, N) = K$, where $tns(x, y) = \sum_{C \in S_{x,y}} sgn(C)$. Let $[a, b>$ be a set of nodes x such that there is a path from a to x, there is a path from x to b and x is different from b. We can remove the last element of each ordered array $C \in S_{x,y}$ and obtain another ordered array C' that begins with x, ends in some node from $[x, y>$ and whose length is smaller by one than the length of C, so $sgn(C) = -sgn(C')$. Therefore $tns(x, y) = -\sum_{z \in [x,y>} tns(x, z)$.

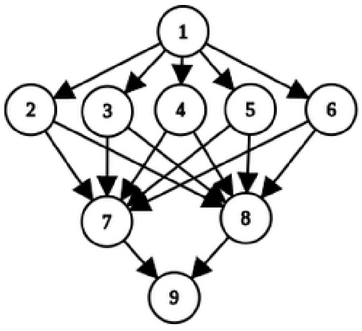
We will build our graph level-by-level. The first level will always contain a single node 1 and the last level will contain a single node N. In the first two subtasks, all nodes (except N) in a certain level will have directed edges towards all nodes in the next level. The general solution will be slightly modified.

The graph which solves the first subtask $1 \leq K < 500$ has three levels. The first level contains node 1, the second level contains $K + 1$ nodes, and the last level contains node $K + 3$. For example, for $K = 4$ the graph looks like the one depicted below.



In this example, using the recurrence relation for $tns(1, x)$, we can see that $tns(1, 1) = 1$, $tns(1, 2) = \cdots = tns(1, 6) = (-1) \cdot 1$, $tns(1, 7) = (-1) \cdot (1 + 5 \cdot (-1)) = 4$.

We will solve the second subtask with $-300 < K \leq -1$ in a similar manner. For example, for $K = -4$ the graph looks like the one depicted below.



By inspecting the solutions for the first two subtasks we can observe that, by adding M nodes in a new level, we are multiplying the sum of all $tns(1, x)$ values by $-(M - 1)$. It is also clear from the recurrence relation that $tns(1, N)$ is equal to the sum of all $tns(1, x)$ values, where $1 \leq x < N$, multiplied by -1. If K has the form $\pm 2^i$, by adding three nodes in a new level, the sum of $tns(1, x)$ is multiplied with -2 so we can obtain the value K up to its sign using $3 + (i - 1) \cdot 9$ nodes. If we get the wrong sign, we can simply add 2 nodes in the next level to multiply the solution with -1.

All that is now left do determine is how to increase the absolute value of the current value by 1. Then we can use multiplications by 2 and additions by 1 to perform a popular algorithm for transposing the number from binary to its decimal notation. If the current sum is negative, we can simply subtract 1 by adding a new node to the current level that is connected with node 1. Similarly, if the sum is positive, we can add two nodes to a new level that are connected to all from the previous level, thereby negating the sum and subtracting 1 as described.

This algorithm constructs a graph for given K in less than $16 \cdot \log_2(K)$ edges.

Task: Skandi

Suggested by: Fabijan Bošnjak and Ivan Paljak
Necessary skills: maximum bipartite matching, minimum vertex cover, König's theorem

In order to score points on the first subtask, it was enough to note that, due to small number of ones, the number of questions in our crossword puzzle is very small. It was enough to use brute force in order to traverse each subset of those questions and assume that was the subset of questions that needed to be answered. For each subset we checked whether the crossword puzzle was filled. The time complexity is $\mathcal{O}(2^Q NM)$, where Q denotes the total number of questions.

The constraints of the second subtask will immediately put an experienced contestant to the right track. One dimension of the matrix is very small and the problem immediately starts smelling like dynamic programming with bitmasks. And that intuition is correct, we will traverse the matrix row-by-row and store in our bitmask in which columns we have decided to answer vertical questions whose answers span through the current row. Therefore, the state can be denoted with `dp[row][mask]` and it tells us what is the minimal number of questions that need to be answered in order to fill an entire crossword puzzle if we have filled all rows until $row - 1$ and we are currently at row row with $mask$ storing active columns as described. Further analysis of the transitions and the reconstruction are left as an exercise to the reader. You can see the implementation of this solution in `skandi_dp.cpp`.

Sometimes it is useful to obtain a more formal description of the problem. In this case, that will naturally lead us to the full solution. Let's denote the *horizontal* questions with $a_1, a_2, \ldots, a_p$, and *vertical* questions with $b_1, b_2, \ldots, b_q$. Let a_i (or b_j) be equal to 1 if we choose to answer that particular question in our solution and 0 otherwise. Note that for each empty square x we need to choose to answer at least one of exactly two questions which have answers that contain x. Therefore, for each empty square there are exactly two questions a_i and b_j for which $a_i \vee b_j = 1$ must hold.

Let's model this with a bipartite graph in which on one side we have nodes that represent horizontal questions, and on other side we have nodes that represent vertical questions. Let's connect two nodes that represent questions a_i and b_j with an edge if there exists an empty square for which $a_i \vee b_j = 1$ must hold. Now, we want to determine the smallest subset of nodes such that each edge is incident to at least one node from that subset. This is a famous problem called minimum vertex cover. Since the graph is bipartite, we can use König's theorem and solve the problem in polynomial complexity.

Task: Trener

Suggested by: Karlo Franić
Necessary skills: graph theory, dynamic programming

For the first subtask you could have checked each possibility. The complexity is $\mathcal{O}(K^N)$.

For the next two subtasks it was necessary to build a directed acyclic graph (DAG) and count the number of paths from nodes on the first level to the nodes on the last level. This can be done using dynamic programming and we leave it as an exercise to the reader. Let's describe what kind of DAG will be built.

On each of the N levels we will have a node for each distinct surname with length equal to that level. The value of that node is equal to the number of surnames it represents. The edges are built naturally – from nodes on level i towards nodes on level $i+1$. Edge from node A to node B exists if surname in node A is different in exactly one letter from surname in node B. Depending on how efficient you have built this DAG, you could have scored only the second subtask or the full score. For additional details, check out the official implementation.