Editorial

Tasks, test data and solutions were prepared by: Vito Anić, Toni Brajko, Dorijan Lendvaj, Martina Licul, and Patrick Pavić.

Implementation examples are given in attached source code files.

Task Sudoku

Prepared by: Martina Licul
Necessary skills: strings

The input data for this problem is not very convenient for solving, but we can reshape it into a more manageable format. A useful representation for solving this problem is to express the Sudoku grid as a matrix of size 9×9, with cells to be filled with numbers.

Now, to determine if there is a mistake in the given 9×9 Sudoku grid, we need to apply the three rules of Sudoku:

Check rows: For each row in the grid, count the number of occurrences of each digit from 1 to 9. If any digit occurs more than once, there is a mistake.

Check columns: Similarly, for each column in the grid, count the number of occurrences of each digit from 1 to 9. If any digit occurs more than once, there is a mistake.

Check 3×3 subgrids: Divide the 9×9 grid into nine 3×3 subgrids. For each subgrid, repeat the process from steps 1 and 2, but instead of rows and columns, check the subgrid.

If none of the checks in steps 1, 2, and 3 found a mistake, that means the board is valid.

Task Labirint

Prepared by: Vito Anić
Necessary skills: dfs

Let us first solve a simpler problem. We have a similar labyrinth, but there are no colors. Some doors are locked, and some are unlocked. Teo and Gabriel can pass through an unlocked door, but they cannot pass through a locked door. How to determine if is it possible to reach the rest of the team? This can be solved with dfs that spreads from the starting point. If it reaches the required goal, then it is possible to go from the start to the end, and if it does not, then it is not possible.

Now we can solve the whole problem. There are only four door colors, and it is possible to go through every combination of colors and try to reach the end by using only the chosen colors. To clarify, all the doors with the colors we chose will be 'unlocked', and the rest will be locked. Now we have the simpler problem from the previous paragraph. For all the combinations of colors for which we can reach the end, we take the one that uses the least colors. Total time complexity is $\mathcal{O}(nmq \cdot 2^4)$

Task AN2DL

Prepared by: Martina Licul
Necessary skills: monotonic queue

To solve the first subtask, you only needed to determine the largest number in the table of size $n \times m$.

To solve the second subtask, you could iterate through all subtables of size $r \times s$ and find the largest number in each of them by looping through all the elements in that subtable. The overall complexity of this approach is $\mathcal{O}(nmrs)$.

To solve the third subtask, you break the problem into two phases: in the first phase, you determine the table you would get if you considered a frame of size $1 \times s$, and in the second phase, you create a new table using a frame of size $r \times 1$ on the table created in the first phase. The result is the table you are looking for. Let's describe the process for each phase.

In the first phase, you can solve it row by row. The first number you would write in the i-th row of the table is the largest number among the numbers $a_{i,1}, a_{i,2}, \dots, ai, s$. The second number you write is the largest number among the numbers $a_{i,2}, a_{i,3}, \dots, ai, s+1$. The set of numbers you consider for determining the first and second number differs only in the elements $a_{i,1}$ and $a_{i,s+1}$; all other elements are the same. Therefore, you can use a data structure like a *multiset*, where you can insert elements in $\mathcal{O}(\log n)$ and find the largest element in $\mathcal{O}(1)$. You insert the first s elements of the first row into the structure and write down the largest among them. Then, you remove $a_{1,1}$ from the structure and add $a_{1,s+1}$. Write down the largest among them as the second number in the first row of the new table. By repeating this process to the end of the row, for each row, you get the table for the first phase.

In the second phase, you repeat a similar process for the columns using a similar data structure. This process results in the final table. The overall complexity of this approach is $\mathcal{O}(nm \log n)$.

To solve the fourth subtask, you need to speed up the approach from the third subtask. Instead of inserting an element into a structure in $\mathcal{O}(\log n)$, you want to do it in $\mathcal{O}(1)$. You can use a data structure called a *monotonic queue*, which allows you to do this in $\mathcal{O}(1)$. You can learn more about it on this link. The overall complexity of this approach is $\mathcal{O}(nm)$.

Task Kocke

Prepared by: Toni Brajko
Necessary skills: dynamic programming

Let's observe the part of the sequence that is strictly positive. Notice that it will form a continuous segment, so we can disregard the zeroes before and after it. Denote ans_i as the number of ways to form a sequence of length i (without zeroes). To account in the zeroes, we see that we can place a sequence of length i on $k - i + 1$ ways $(i \leq k)$ in the initial sequence. Therefore, the solution is given with the following expression:

$$\sum_{i=1}^{min(n,k)} ans_i \cdot (k - i + 1)$$

It remains to calculate ans_i for each i.

We can generate every possible sequence of moves and use it to calculate the final sequence we get. To count the number of different sequences, we can, for example, sort them (and then if two sequences are equal they will necessarily be neighboring). The complexity of this algorithm is $O(n^2 \cdot n^2)$ or $O(n \cdot 2^n)$, depending on the implementation, and is sufficient to solve the first subtask.

Let's observe creating the sequence backwards. We have (n) as the initial sequence. For each number from $n - 1$ to 1 in decreasing order, we put number i next to $i + 1$. If we put a number on an already existing position, the sequence won't change (because the larger number will come after in the original process, and it will "overwrite" the lower one). If we want to put a number on a place where we haven't written any numbers yet (i.e. on a zero, which is not included in current sequence because we disregarded it), the sequence will extend by one spot either to the left or to the right. Notice that this process it equivalent to the original one.

We use dynamic programming to solve the rest of the task. Let $dp[i][j]$ be the number of achievable sequences of length i, such that j is the minimum of the sequence. Notice that the minimum of any such sequence will be one of its endpoints. This follows directly from the aforementioned description. WLOG, assume the smallest element in the sequence is the right endpoint. We have two possibilities:

- The second smallest element of the sequence is also on the right end, one place to the left from the smallest element. It can be any numer of the form $j - 1 - 2k$, for $k \in \mathbb{N}$.

- The second smallest element of the sequence is the left endpoint. It can be any number of the form $j+i-1+2k$, for $k \in \mathbb{N}^0$.

Detailed analysis of these expressions is left as an exercise for the reader. Now the following holds:

$$dp[i][j] = \sum_{k=1}^{\left\lfloor \frac{n-j+1}{2} \right\rfloor} dp[i-1][j-1+2k] + \sum_{k=0}^{\left\lfloor \frac{n-j-i+1}{2} \right\rfloor} dp[i-1][j+i-1+2k]$$

$$ans_i = \sum_{j=1}^{n} dp[i][j]$$

The complexity of each transition is $O(n)$, and there are a total of $O(n^2)$ states, so the final complexity of direct implementation of this formula is $O(n^3)$, and is sufficient to solve the second and the third subtask.

Finally, we can speed up each transition by using prefix sums by parity, since the independent term is always $2k$. Final complexity is $O(n^2)$. Notice that it does not depend on the sequence length k from the task statement at all. For details of implementation and base cases refer to the official solution.

Task Mostovi

Prepared by: Patrick Pavić
Necessary skills: dfs tree, segment tree

For solving the first and second subtasks, it was sufficient to attempt a depth-first search on the graph after removing each edge and check if the graph is connected.

To solve the remaining subtasks, similar to the classical algorithm for finding bridges, our algorithm will work with the DFS tree of the given graph. A specific property of the DFS tree is that all edges not in the tree connect a node to one of its descendants. We will refer to nodes of that kind as *back edges.* The edges that have the property sought in the task will be called *good edges.*

First, we will check for all edges in the DFS tree to see if they are good edges. This algorithm will not differ much from the algorithm for finding articulation points, and we leave the details to the reader as an exercise. This was sufficient to solve the fourth subtask, while checking the remaining $m-n+1$ edges using the same algorithm as in the first two subtasks.

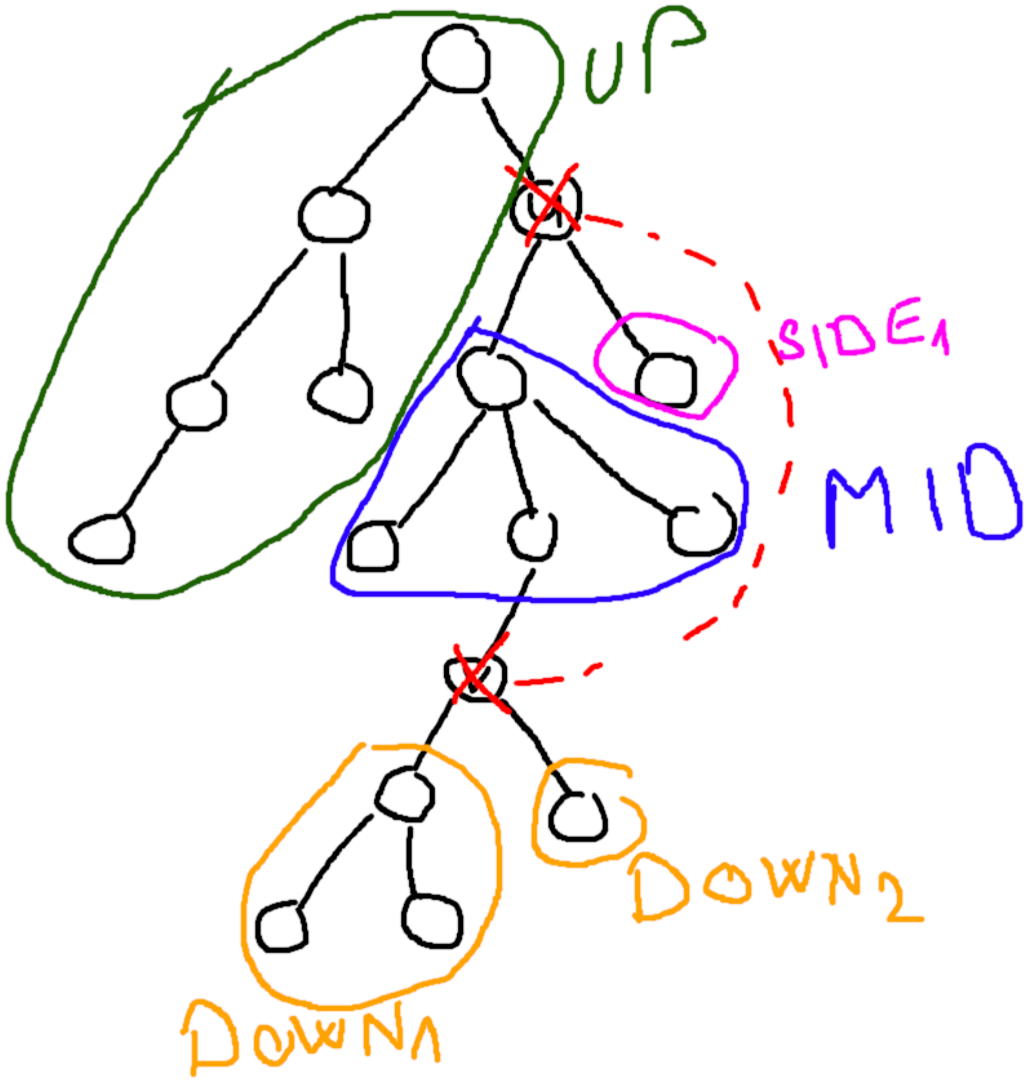
Now, let's consider a back edge (u, v) and the components formed by its removal. It is necessary to check if these components are connected to each other by back edges. If they are not, that edge is good. These components can be divided into four primary groups:

- UP - a component that contains the root of the DFS tree (the node from which we initiated the DFS)
- SIDE$_i$ - the components formed by removing node v in its subtree, except the one that contains node u
- MID - a component containing the path between nodes u and v (as the edge is not in the tree, this path will always exist)
- DOWN$_j$ - the components formed by removing node u in its subtree

Assuming that the graph remains connected, all components of the SIDE$_i$ group must contain a back edge within them that connects them to the UP component. Similarly, all components of the DOWN$_j$ form must have a back edge connecting them either to the MID component or to the UP component (note that they cannot be connected to the SIDE$_i$ component due to the condition of the DFS tree). Finally, to keep the graph connected, at least one of the following two statements must be true:

- There exists a DOWN$_j$ component that contains both a back edge to the UP component and to the MID component.
- There exists a back edge from the MID component to the UP component.

All of these conditions can be efficiently checked using various segment trees ordered by the preorder traversal of the DFS tree. One very useful thing is to compute, for each subtree, the highest and lowest back edge leaving that subtree (disregarding back edges completely in the subtree). This was possible to calculate using segment trees or the small to large algorithm. Depending on the implementation, it was achievable to reach a complexity of $O((m+n)\log n)$ or $O((m+n)\log^2 n)$, which was sufficient to solve the problem. Slower implementations solved the third subtask.



Sketch of an example