CROATIAN OPEN COMPETITION IN INFORMATICS 2012/2013

ROUND 5

SOLUTIONS

COCI 2012/2013	**Task LJESTVICA**
5th round, February 16th, 2013	**Author:** Adrian Satja Kurdija

The basic idea: We use a variable **Amin** to count accented tones that are main tones in A-minor, and a variable **Cmaj** to count accented tones that are main tones in C-major.

Implementation details: One traversal (using a for-loop) over the input string is used to find acented tones in the following way: a tone is accented if it is the first character in the string, or if the previous character was "|". For each accented tone, we can use a branching statement (such as if-then-else or switch-case) to check whether it is equal to "C", "F", or "G" (in which case we increment the variable **Cmaj**), or to "A", "D", or "E" (in which case we increment the variable **Amin**).

Finally, it is obvious what we need to output if **Amin** < **Cmaj** or if **Amin** > **Cmaj**. If, on the other hand, **Amin** = **Cmaj**, we simply need to check whether the last character of the input string is "A" or "C", as defined in the problem statement.

Necessary skills: loops, string manipulation

Category: ad-hoc

COCI 2012/2013	Task ARHIPELAG
5th round, February 16th, 2013	**Author:** Adrian Satja Kurdija

Let us create a new matrix representing the (untrimmed) future map of the archipelago. The cells of this matrix are filled in the following way: if the corresponding cell in the input matrix is a sea cell or a land cell surrounded by at least three sea cells, the result is a sea cell; otherwise, it is a land cell.

How can we find cells that surround a given cell? If the current cell has coordinates (**r**, **c**), we need to check cells (**r**, **c** + 1), (**r**, **c** - 1), (**r** + 1, **c**), and (**r** - 1, **c**). It can be done using a for-loop with an index **i** from 1 to 4, looking at cell (**r** + **u**[**i**], **c** + **v**[**i**]), where the helper arrays **u** and **v** are defined as **u**[] = {0, 0, 1, -1}, **v**[] = {1, -1, 0, 0}. However, before indexing a neighbouring cell, we first need to check whether the cell is outside of the matrix boundaries. If it is, we assume that it is a sea cell; otherwise, it is safe to read the cell value.

After determining the future map, we need to find its smallest rectangular part containing all land cells. It can be done by finding the leftmost, rightmost, uppermost, and lowermost land cells, using four variables (*min_column*, *max_column*, *min_row*, *max_row*), updated as we traverse the future map and reach a land cell. Finally, we simply need to output all rows from *min_row* to *max_row*, but only cells from *min_column* to *max_column*.

Necessary skills: character matrix manipulation

Category: ad-hoc

COCI 2012/2013	Task TOTEM
5th round, February 16th, 2013	**Authors:** Nikola Dmitrović, Antun Razum

Since the problem in this task is finding the shortest path, in number of steps, from the beginning to some position, we can obviously use the breadth-first search (BFS) algorithm. However, we first need to derive a graph corresponding to the problem that we can apply BFS to. In this graph, nodes will correspond to tiles, and edges will exist between some two nodes if it is possible to step directly from one corresponding tile to the other.

The graph can be constructed in the following way. The numbers chiseled into tiles can be read into a matrix with **N** rows and 2***N** columns, where squares not covered by a tile (the first and last cells of even-numbered rows, numbered from 1) can be set to zero. Another matrix of the same dimensions can be used to store the corresponding tile index for each square. Now we can, for each square, iterate over its four neighbouring squares and add an edge between the current square's and the neighbouring square's tile iff the cells have different tile indexes (so that we don't add an edge from a tile to itself) and the squares have equal numbers (the condition of stepping from one tile to the next). The complexity of this part of the solution is $O(\mathbf{N}^2)$ since we have processed each of the $2*\mathbf{N}^2$ squares once (as the current square) plus at most four times (as the neighbouring square).

Now that we have constructed the graph, we can apply breadth-first search starting from the first tile. For each tile, we need to keep track not only of the distance to the starting tile, but also of the previous tile (the tile that we stepped directly from when moving to the current tile), so that we can reconstruct the path later. The complexity of this part of the solution is also $O(\mathbf{N}^2)$, since BFS processes each of the $\mathbf{N}^2$ nodes once (as the current node) plus at most six times (since there are at most six neighbouring tiles).

Finally, we need to find the tile with the largest index that we have reached and, going backwards using the previous nodes stored during BFS, reconstruct the shortest path by storing the traversed tiles' indexes, from end to start, in an array. The final solution is the length of the obtained array and the array itself, output in reverse. The complexity of this part of

the solution is again $O(\mathbf{N}^2)$ because the shortest path can contain each of the $\mathbf{N}^2$ tiles at most once.

Necessary skills: breadth-first search

Category: shortest path reconstruction

COCI 2012/2013	**Task HIPERCIJEVI**
5th round, February 16th, 2013	**Author:** Bruno Rahle

The graph described in the problem statement is a hypergraph. When solving the shortest path problem on a hypergraph, we can convert it to a "normal" undirected graph by simply connecting all pairs of nodes (stations) on the same hyperedge (hypertube) with undirected edges. This results in a graph with a total of **N** nodes and **M*****K***(**K**-1)/2 edges. A simple breadth-first search applied to this graph is then a solution, ignoring time and memory constraints.

However, the constraints prevent such a solution from obtaining all points. The most elegant way to speed up the solution is adding a new node for each hypertube, connecting it with undirected edges to all stations connected to the corresponding hypertube. Such a graph has **N**+**M** nodes and **M*****K** edges. A breadth-first search applied to this graph yields the solution 2***X**-1, where **X** is the number of stations on the shortest path from station 1 to station **N**.

Necessary skills: breadth-first search

Category: graphs

COCI 2012/2013	Task ROTIRAJ
5th round, February 16th, 2013	**Author:** Matija Milišić

We will arrive at a solution of this problem in several steps. Let us first simplify the problem.

Notice that each operation has an inverse operation, which is of the same type, but with the opposite sign, that is, rotating in the opposite direction. This means that we can obtain the starting sequence by applying inverses of the input operations, in reverse order, to the final sequence.

Also, notice that any rotation to the left can be replaced with an equivalent rotation to the right. Rotation of the whole sequence by **X** to the left can be replaced by a rotation of the sequence by **N - X** to the right; analogously for section rotations.

The next simplification is replacing the numbers in the sequence with their positions, and then applying operations to the position sequence. It is easy to restore the original numbers from the positions after finishing with the rotations.

The problem has been reduced to starting with the sequence (0, 1, …, **N-**1) and applying 2 types of rotation-to-the-right operations to it. A simple simulation of each operation leads to the complexity O(**N** * **Q**), which is sufficient for 40% of total points.

Looking at numbers **A** and **A** + **K** (modulo **N**), observe that their distance (modulo **N**) remains equal to **K** after any rotation operation. Therefore, it is sufficient to track only the positions of numbers (0, 1, …, **K-**1) while applying rotations. In the end, the positions of the remaining numbers can be reconstructed as follows: pos[**A**] = (pos[**A** % **K**] + **A** / **K** * **K**) % **K**.

This improved simulation of each operation has the complexity O(**K** * **Q**), which is sufficient for 70% of total points.

For further optimization, we can track the positions of the numbers (0, 1, …, **K**-1) (modulo **K**). After each operation, their positions are some rotation of the general form: **B**, **B**+1, …, **K**-1, 0, 1, …, **B**-1. This rotation can be tracked by a single global variable representing the total number of rotations to the right needed to transform the starting sequence of positions to the current one. Both types of rotations to the right simply add **X** to this total rotation. Thus, the position of each of the first **K** numbers in its current section is easily obtainable from the total rotation.

To unambiguously determine the position of the first **K** numbers, and by extension all **N** numbers, we also need to track the current section for each of the **K** numbers.
An operation of type 1 doesn't change the contents of the sections. An operation of type 2 can be processed modulo **K**, with the whole part tracked by another global variable since it applies to all numbers equally. Looking at the first **K** numbers, an operation of type 2 moves a total of **X** numbers (where **X** is the remainder modulo **K**) to the following section. We also know exactly which **X** numbers are moved. From the general form of rotation, we can see that it applies to the last **X** numbers of the sequence: **B**, **B**+1, …, **K**-1, 0, 1, …, **B**-1. Since these numbers are always contiguous, we can increase the interval in time O(1). In the array tracking the sections for each of the **K** numbers, we add a 1 to the beginning and a -1 to the end of the interval.
The positions of the first **K** numbers are thus obtained from two parts. One defines the section that each number is in, and the other defines the position inside the section. After obtaining the positions of the first **K** numbers, the positions of the remaining **N** - **K** numbers are derived as described above.
Since each operation is now processed in time O(1), the total complexity is O(**N** + **Q**).

Necessary skills: -

Category: ad-hoc

COCI 2012/2013	Task MNOGOMET
5th round, February 16th, 2013	**Author:** Anton Grbin

Before understanding the solution of the problem itself, some basic probability theory is required.

Independent events. If two events are independent, the probability of both events being realized is equal to the product of their individual probabilities.

Total probability. A complete set of alternatives is a set of disjoint events which together cover the entire sample space. If the only methods available to commute to school are by tram or on foot, then those two events form a complete set of alternatives for commuting to school. Now, the probability of an event in the same space can be expressed as follows:

$$P(\mathbf{A}) = P(\mathbf{N1})\,P(\mathbf{A|N1}) + P(\mathbf{N2})\,P(\mathbf{A|N2}) + \ldots$$

For the commuting to school example, it can be applied as: the probability of arriving to school today (P(**A**)) is equal to the probability of going on foot (P(**N1**)) times the probability of arriving if going on foot (P(**A|N1**)) plus the probability of going by tram (P(**N2**)) times the probability of arriving if going by tram (P(**A|N2**)).

Input. From the input data it is possible, in a straightforward way, to find the probability of a player **A** coming into possession of the ball one second after a player **B** had the ball, as well as the probability of each team scoring if a player **C** has the ball.

Computation. We will solve the problem in two steps. In the first step, we find the probabilities of events of the following types:

Pg(**X**,**Y**,**T**) = P({the first player of team **X** has the ball, **T** seconds later team **Y** scores})
Pm(**X**,**T**) = P({the first player of team **X** has the ball, **T** seconds later no team has scored})

where **X** and **Y** are the labels of one of the teams, and **T** is a positive integer. The probabilities **Pm** and **Pg** can be computed using dynamic programming, with the state {entity, number of seconds, team that had the ball in the beginning}, and the value for each state being the **probability that the entity has the ball after the number of seconds** has passed, where an entity can be any player or the goal of one of the teams. The relation relies only on the states in the previous second. In this step we assume that once a goal is scored, the ball remains in the goal. The time complexity of the first step is $O(\mathbf{N}^2 * \mathbf{T})$.

In the second step, we can ignore the individual players and their probabilities; the state is modelled by {current result, team, number of seconds}, with the value being the **probability that after the number of seconds since game start, the team has just scored a goal, leading to the current result**. For each state, the previous result is uniquely determined. The required probability can be decomposed to a complete set of alternatives which describe the second and the team that scored a goal in that second leading to that previous result. The solution for the current state is obtained as the sum, over all the alternatives, of the products of the probability of the alternative and the probability of a goal just being scored to obtain the current result, which can be read from the array **Pg**. The time complexity of the second step is $O(\mathbf{R}^2 * \mathbf{T}^2)$.

Output. The probability of an outcome can also be decomposed to a complete set of alternatives, describing the second when the last goal was scored. There must have been no scored goals from that moment to the end of the game, which is included by multiplying the probability from the second step with the **Pm** value for the appropriate time period.

However, if we are processing a winning result (with **R** goals scored by one of the teams), we must not multiply in the **Pm** value, since the game has ended at that moment and the remaining time period is zero.

Necessary skills: basic probability theory, dynamic programming, mathematical problem modelling

Category: dynamic programming