EDITORIAL
JUNIOR BALKAN OLYMPIAD IN INFORMATICS, DAY 1

Problem: AB

Proposed by: prof. Ionel-Vasile Piț-Rada

We use the terms *AB-permutation* and *AB-transposition* for permutations and transpositions of the K elements which obey the AB matrix constraints. (A transposition is a permutation where exactly two elements are swapped, such as (12435).) We will show that if an AB-permutation exists, then an AB-transposition also exists.

Recall that any permutation consists of a composition of cycles. For example, the permutation (42613578) consists of the cycles $3 \to 5 \to 6 \to 3$ and $1 \to 4 \to 1$, as well as the single-element cycles 2, 7 and 8. For any AB-permutation P we will show how to find an AB-transposition between two elements on the same cycle. Once we find it, we simply put the other elements of P back on their original positions. This will not break any constraints, because the original matrix was an AB-matrix.

So let us consider one of the cycles in P, call it $x_0, x_1, \ldots, x_{c-1}, x_c = x_0$. Let $[a_i, b_i]$ be the range of values that can replace x_i in the matrix. Obviously, $a_i \le x_i \le b_i$ for all i. The values of a_i, b_i can be deduced from the values neighbouring x_i: a_i is the maximum of the values to the left and above x_i (if they exist), plus one, and b_i is the minimum of the values to the right and below x_i (if they exist) minus one.

Assume without loss of generality that x_0 is the minimum value (we can rotate the cycle if not). Then the values $x_0, x_1, \ldots$ will increase for a while, until at some point x_d there will necessarily exist a decrease, $x_d < x_{d-1}$ (d can be equal to c if the cycle consists of exactly one increasing streak). So there exists some $i \ge 1$ such that $x_{i-1} \le x_d < x_i$. We will show that the transposition (x_i, x_d) is an AB-transposition. Let us collect some useful inequalities.

(1) $a_i \le x_i \le b_i$, because x_i obviously obeys its own range.
(2) $a_d \le x_d \le b_d$, similarly.
(3) $x_{i-1} \le x_d < x_i \le x_{d-1}$, by our choices of d and i.
(4) $a_i \le x_{i-1} \le b_i$ because x_{i-1} fits in place of x_i in the cycle.
(5) $x_{d-1} \le b_d$ because x_{d-1} fits in place of x_d in the cycle.

From (1), (3) and (4) we get $a_i \le x_{i-1} \le x_d < x_i \le b_i$. Therefore, x_d obeys the constraints of x_i. Conversely, from (2), (3) and (5) we get $a_d \le x_d < x_i \le x_{d-1} \le b_d$. Therefore, x_i obeys the constraints of x_d. By definition, this means that (x_i, x_d) is an AB-transposition.

Now, we will explain how to determine whether there is any AB-transposition. We will iterate over the values X in increasing order. While iterating, we will keep a stack with all the previous positions in which the current position can still be placed. In other words, at a step of the iteration x_i, we will keep a stack of all previous positions x_j for which $x_i \le a_j$. Therefore x_i can be placed in place of all positions x_j from the stack. For an AB-transposition, we need to check whether there is any x_j that is large enough to fit in $[a_i, a_i]$. For this, it is sufficient to check the maximum value of the all x_j, which is the actually the top of the stack.

Now, regarding keeping the stack up-to-date. At each iteration step, we need to eliminate all x_j which have a $b_j < x_i$. It is not necessary to remove them from the middle of the stack, but instead we will remove them from the top of the stack in a lazy manner. As such, we will pop all tops of the stack that have $b_j < x_i$. Once a top exist for which $x_i \le b_j$, we check that top whether it generates a swap with x_i. If yes, then the solution is not unique. If not, then we continue to the next iteration.

Here is pseudo-code following this idea:

(1) Iterate over $x_1, \ldots, x_k$, keeping a stack S.

(2) Suppose we are considering x_i. While S is not empty:
   (a) Let x_t be the top element of S.
   (b) If $b_t < x_i$ then eliminate x_t and re-try this loop.
   (c) If $x_t < a_i$ then add x_i to S and exit this loop.
   (d) Otherwise, output the pair (x_t, x_i) as being AB-transposition.
(3) If no pair has been found then conclude that no AB-transpositions.

Solution proposed by Tamio-Vesa Nakajima. Consider some query. Suppose that the query deals with $x_1 < \ldots < x_k$. For each x_i, let $[l_i, r_i]$ be the interval of values that could replace x_i while maintaining the orderedness properties of the matrix. (Observe that l_i is the maximum of the values above and to the left of x_i plus one, and r_i is the minimum of the values to the right and above of x_i minus one.)

Theorem. *The elements $x_1, \ldots, x_k$ can be reordered while maintaining the orderedness properties of the matrix if and only if a pair (x_i, x_j) exists that can be swapped while maintaining these properties.*

Thus it is sufficient to look only for a pair of elements that can be swapped. We can then use the following stack-based algorithm:

(1) Iterate over $x_1, \ldots, x_k$, keeping a stack S.
(2) Suppose we are considering x_i. While S is not empty:
   (a) Let x_t be the top element of S.
   (b) If $r_t < x_i$ then eliminate x_t and re-try this loop.
   (c) If $x_t < l_i$ then add x_i to S and exit this loop.
   (d) Otherwise, output the pair (x_t, x_i) as being swappable.
(3) If no pair has been found then conclude that no swappable pair exists.

Why is this algorithm correct? We will show that the algorithm is sound (when it outputs a pair it always outputs a correct pair) and complete (if a swappable pair exists then the algorithm must output some pair).

Soundness: Observe that if we output the pair (x_t, x_i) then we know that $x_i \leq r_t$ and $l_i \leq x_t$ Furthermore we know, by the order in which we go through the elements, that $x_t \leq x_i$, and thus $l_t \leq x_t \leq x_i \leq r_i$. All of this information is sufficient to deduce that we can swap x_i with x_t.

Completeness: First observe that if some x_t is eliminated from the stack by some x_i, then $x_t < x_i$ and $r_t < x_i \leq r_i$. Furthermore x_i must eventually be added to the stack (unless we already output some pair). This means that after some x_a was added to the stack, some $x_{a'}$ must always exist in the stack, such that $x_a < x_{a'}$ and $r_a < r_{a'}$.

Now suppose we can swap some pair, and suppose that (x_a, x_b) is the swappable pair that minimizes the value of x_b, and if there are several such pairs. If we reach x_b in the iteration (if we haven't then we have already outputted some pair and there is nothing to prove) then at some point x_a was added to the stack; thus as shown above the stack contains some $x_{a'}$ where $x_a < x_{a'}$ and $r_a < r_{a'}$. Since we know already that $l_b \leq x_a$ and $x_b \leq r_a$ (since we can swap x_a and x_b) it follows that $l_b \leq x_{a'} < x_b \leq r_b$ and $l_{a'} \leq x_{a'} < x_b \leq r_a < r_{a'}$ — in other words some element $x_{a'}$ exists in the stack that can be swapped with x_b.

If this element is ever at the top of the stack then we will surely output the pair $(x_{a'}, x_b)$. Until the element reaches the top of the stack, if x_t is the element at the top of the stack, then it is not possible for $x_t < l_b$ (since $l_b \leq x_{a'} < x_t$) — thus we must either output some other pair, or pop an element of the stack. It follows that we must eventually output some pair, as required.

Problem: MPF

Proposed by: prof. Daniela Lica

Consider the following notations:

- $VMAX$ the maximum value that can be assigned to X, with respect to the problem's constraints;
- $Maxp[i]$ the largest prime divisor of the positive integer i, where $1 \leq i \leq VMAX$. This can be computed in $O(VMAX \log VMAX)$ time complexity using an algorithm similar to the Sieve of Eratosthenes. When marking elements as non-prime, we save the prime number that divides it. Because the primes are traversed in increasing order, the last saved divisor is also the largest. Thus, using operation 1 once, a positive integer X becomes $X/Maxp[X]$;
- $Level[i]$ the sum of the exponents from the prime factor decomposition of the positive integer i, where $1 \leq i \leq VMAX$ (this can be computed along with the $MaxP[]$ array, such that $Level[X] = 1 + Level[X/Maxp[X]]$). In fact, for an integer i, $Level[i]$ represents the minimum number of operations of type 1 that need to be applied successively on i for it to become equal to 1 and, at the same time, the minimum number of type 2 operations that need to be applied successively on 1 to obtain i;
- $Query(i, j)$ the minimum number of operations that need to be applied successively on i for it to become equal to j. Obviously $Query(i, j) = Query(j, i)$.

Subtask 1. We precompute the results for each possible pair of numbers, with approximately $O(VMAX^2 T)$ time complexity, where T is the number of prime numbers less than or equal to $\sqrt{VMAX}$. We use a $2D$ array $ans[i][j] = k$, where $Query(i, j) = Query(j, i) = k$. Every i-th line can be computed starting using a fill algorithm, starting from the number i with the help of a *queue* data structure (FIFO), processing each number from 1 to $VMAX$ exactly once. When processing a state X, we try to consider state $X/Maxp[X]$ (type 1 operation applied on X) and all states $j \cdot X$, where j is a prime number, $j \geq Maxp[X]$ (type 2 operation applied on X).

For each query, the output can be provided in $O(1)$ time. Final time complexity for answering all queries is $O(Q)$.

Subtask 2. For each query we will compute the prime factor decomposition of the two numbers (X, Y) in $O(\sqrt{VMAX})$ time complexity. Consider the number Z the maximum number obtained as an intermediary value in both transformations of X to 1 and Y to 1. The value of Z is in fact the product, in order, of the smallest common prime factors of X and Y. The answer for $Query(X, Y) = (Level[X] - Leve[Z]) + (Level[Y] - Level[Z])$. This is because it takes $Level[X] - Level[Z]$ operations of type 1 to transform X into Z, and then another $(Level[Y] - Level[Z])$ operations of type 2 to transform Z into Y. The final complexity for all queries is therefore $O(Q \cdot \sqrt{VMAX})$.

Subtask 3. Similarly to the previous approach, for each query pair (X, Y), we will determine the value of Z. The approach will follow the transformation of X and Y to the value 1, until a common value Z is found. While the two values X and Y are different (i.e. a Z value has not been found), we will pick the maximum of X and Y and apply an operation of type 1 to it, thus getting closer to the value of Z. The maximum complexity of one individual query is determined by the maximum number of prime factors that a value can have - $O(\log_2 VMAX)$. The final complexity is $O((Q + VMAX) \cdot \log_2 VMAX)$

Problem: Roboclean

Proposed by: prof. Zoltan Szabo

Solution proposed by Tamio-Vesa Nakajima. First, observe that the parity of the length of any path from some point in the grid to some other point in the grid is fixed. To see why, consider a checkerboard pattern overlaid on top of the grid. If the colour of the starting cell is equal to the colour of the ending cell, then any path between them has odd length; otherwise, it must have even length. Thus, observe that if we can always create a path of length $N \times M$ or $N \times M - 1$, then we will always create a path of optimal length — if we create one of length $N \times M$ then it is obviously the longest possible path, and if it is of length $N \times M - 1$ then a longer path of length $N \times M$ is impossible due to parity. We will now describe a recursive algorithm that always generates such a path.

First, assume without loss of generality that we want to create a path between $(1,1)$ and (i,j), and also that $i \geq j$, and if $i = j$ then $N \leq M$. By rotating and flipping the matrix it is always possible to reach this case. Note that since $(i,j) \neq (1,1)$ it follows that $i > 1$. Our algorithm will have two cases.

Case 1, $n > 2$**:** Observe that by our conditions it is impossible for $(i,j) = (2,M)$ or for $i = 1$. In this case we can therefore always end our path with the sequence $(2,M) \to (1,M) \to \ldots \to (1,1)$. Thus we can reduce to the case of finding a path from (i,j) to $(2,M)$ without using the first line — or equivalently finding a path from $(i-1,j)$ to $(1,M)$ in an $(N-1) \times M$ matrix.

Case 2, $N = 2$**:** In this case we can prove that either $(i,j) = (2,1)$ or $(i,j) = (2,2)$. In either case our path is $(i,j) \to \ldots \to (2,M) \to (1,M) \to \ldots \to (1,1)$.

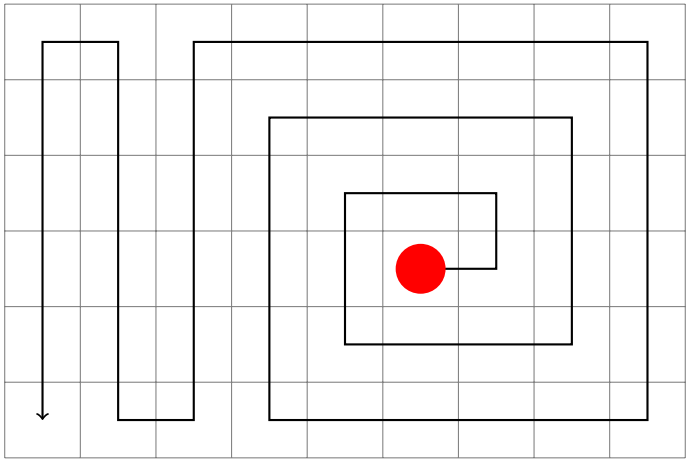
The algorithm has the following steps:

(1) the matrix is rotated until the exit cell lands on $(1,1)$, i.e. $L_2 = 1$ and $C_2 = 1$;
(2) the matrix is then transposed (i.e. cell (i,j) becomes (j,i)) in order to obtain $L_1 \geq C_1$, and if $L_1 = C_1$ we want to obtain $N \leq M$;
(3) we add to the path the sequence of cells $(1,1),(1,2),\ldots,(1,M),(2,M)$, and from here we recursively solve the smaller task of finding the best path from $(2,M)$ to (L_1,C_1), which is equivalent to finding a path from $(1,M)$ to (L_1-1,C_1) in a $(N-1) \times M$ matrix;
(4) the steps are repeated until we obtain a matrix with only 2 rows for which we construct the path using the method mentioned in the second case of the above demonstration.

A careful implementation is needed to remap the North, South, East and West directions when rotating or transposing the matrix.

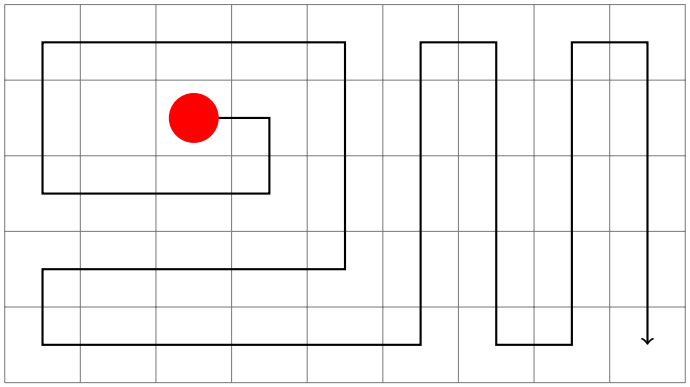
We will look at the path generated by the proposed approach on a couple of examples.

Example 1. Considering $N = 6$, $M = 9$, the starting cell at $(4,6)$, and the exit cell at $(6,1)$.



All cells are thus cleaned by the robot and the sequence of moves is the following:
ENWWSSEEENNNWWWWSSSSEEEEENNNNNWWWWWWSSSSWNNNNNWSSSSS

Example 2. Considering $N = 5$, $M = 9$, the starting cell at $(2, 3)$, and the exit cell at $(5, 9)$.



In this example a single cell is left uncleaned. The sequence of moves is as follows:
`ESWWWNNEEEESSSWWWWSEEEEENNNNESSSSENNNNESSSS`

Solution proposed by Cristian Frâncu. An alternative solution, to compute the path in reverse:

(1) At each step consider the (at most) four possible moves to the neighboring cells;
(2) Choose the cell that's farthest from the robot's start position, using the Euclidean distance;
(3) In case of equal distance choose the cell that has more possible following moves (i.e. more adjacent neighbors not yet in the path);
(4) Repeat until reaching the starting point.

This algorithm generates the same paths as the previous solution on the examples mentioned above.

Scientific Committee

The problems were prepared by:

- Gheorghe-Eugen Nodea (chair) - "Tudor Vladimirescu" National College, Târgu Jiu
- Adrian Panaete - "A.T. Laurian" National College, Botoșani
- Daniela Elena Lica - Centre of Excellence, Ploiești
- Ionel-Vasile Piț-Rada - "Traian" National College, Drobeta Turnu Severin
- Zoltan Szabo - County School Inspectorate, Mureș
- Radu Voroneanu - Google
- Vlad Gavrilă - University of Cambridge
- Emanuela Cerchez - "Emil Racoviță" National College, Iași
- Marinel Șerban - "Emil Racoviță" National College, Iași
- Mihai Bunget - "Tudor Vladimirescu" National College, Târgu Jiu
- Tamio-Vesa Nakajima - Oxford, Computer Science department, UK
- Bogdan Iordache - University of Bucharest
- Cristian Frâncu - Clubul Nerdvana București
- Cosmin Pițˇ Rada - Bolt
- Ciprian-Daniel Chește - "Grigore C. Moisil" Technological High School, Buzău
- Marius Nicoli - "Frații Buzești" National College, Craiova
- Dan Narcis Pracsiu - "Emil Racoviță" Theoretical High School, Vaslui
- Flavius Boian - "Spiru Haret" National College, Târgu Jiu
- Petru Simion Opriță - Liceul Regina Maria Dorohoi
- Andrei-Costin Constantinescu - ETH Zurich