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 $(i,j) \to \ldots \to (2,M) \to (1,M) \to \ldots \to (1,1)$.

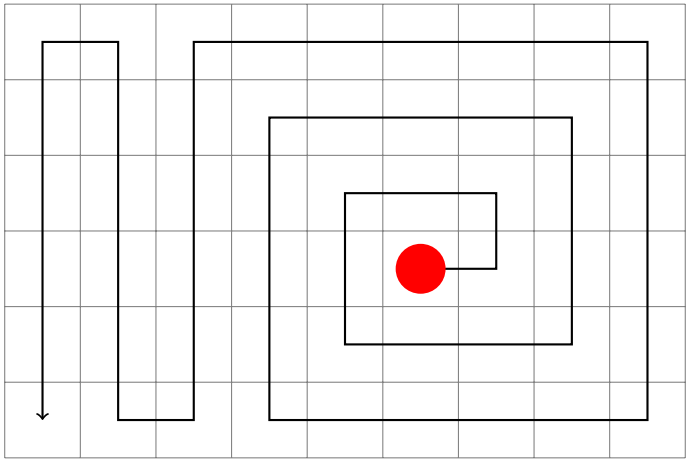
The algorithm has the following steps:

(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.

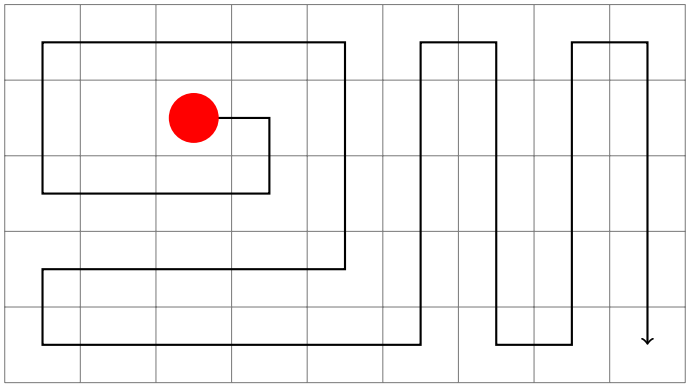
We will look at the path generated by the proposed approach on a couple of examples.

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

EDITORIAL
JUNIOR BALKAN OLYMPIAD IN INFORMATICS, DAY 2

Problem: Kaguya Wants to Receive Flowers

Proposed by: Vlad Gavrilă

Subtask 1.

For this subtask, there is only one flower parcel and we need to find the distance to it from all other parcels. Let's say the flower parcel is located at position (x, y). For a parcel (r, c), the distance from that parcel to the flower parcel will be $|x - r| + |y - c|$, so we print that.

Subtasks 2 and 5 — $O(N^2 F \log F)$.

For the second subtask, the limits are generally small, so any polynomial solution will be accepted. We present such a solution that also solves subtask 5.

First, we create an array P containing all (x, y) parcels that contain flowers. Then, for each parcel (r, c) for which we want to determine the answer, we will sort array P increasingly by the distance between the current parcel (r, c) and each parcels in P. The answer for cell (r, c) will be the sum of distances between itself and the first K elements of P.

Since sorting (and computing the distances) takes $\mathcal{O}(F \log F)$, and we need to do this $\mathcal{O}(N^2)$ times (once per cell), the total time compexity is $O(N^2 F \log F)$.

Subtask 3 — $\mathcal{O}(N^3)$.

Let (r, c) be a fixed point in the matrix. Note that all the flower parcels at distance d from (r, c) lie on a diamond-like edge. All the flower parcels closer than d lie inside the diamond. We denote this diamond (including both the edge and inside) as (r, c, d) and refer to d as the radius. We observe that, for any cell (r, c), the flower parcels closest to (r, c) will be included in the smallest radius diamond centered in (r, c).

The following algorithm follows for computing the answer for parcel (r, c): we start with a diamond of size 1, and increase it until it contains at least K flowers, adding the corresponding distances to the answer as we go. As we can increase this diamond at most $2N$ times, we must do each increase in $O(1)$ time in order to have our final $\mathcal{O}(N^3)$. Let's introduce the following concept:

Interval sums on an array.

Given an array A, we wish to quickly answer queries of the form $\text{sum}(i, j) = A_i + \cdots + A_j$. To do this, we precompute partial sums array of A, namely $P_i = A_1 + \cdots + A_i$. Then $\text{sum}(i, j) = P_j - P_{i-1}$.

Putting it all together.

Let's say we currently have diamond (r, c, d) that contains $f < K$ flowers. We therefore need to increase its size, by adding the edge of the $d + 1$ radius diamond to our current diamond. This edge is made up of four contiguous side pieces which are at a 45 degree angle relative to the sides of the garden, for which we want to know how many flowers (f') they contain.

We can determine this easily using the interval sums concept by considering as "arrays" all sets of parcels that form a 45 degree angle "line" relative to the sides of the garden, and selecting our edge pieces as intervals from the corresponding "arrays". Note that we need to be careful that our side pieces do not extend beyond the garden's limit (but we can easily fix that by cutting them to size).

Finding that our edge contains f' flower parcels, we can either still be in need of adding more flowers ($f + f' < K$), in which case we add $f' \times (d+1)$ to our answer, increment d and repeat the process above, or add $(K - f) \times (d+1)$ to our answer and terminate the computation for our current parcel.

Subtask 4.

In this particular subtask, we need to find the closest flower parcel to each parcel in the garden. We can do this by applying Lee's algorithm[1], having as starting points all the parcels with flowers. The final time complexity is $\mathcal{O}(N^2)$.

Subtask 6 (and possibly also subtasks 7 and 8) — $\mathcal{O}(N^2 \log N)$.

Assume we had an oracle that could answer the following questions in constant time:

(1) How many flowers are there inside the diamond (r, c, d)?
(2) What is the sum of the distances of those flowers from (r, c)?

Then an $\mathcal{O}(N^2 \log N)$ algorithm follows easily: for every point (r, c), run a binary search to find the smallest radius d such that (r, c, d) contains at least k flowers. If the diamond contains more than K flowers, the surplus necessarily lies on the border (otherwise d would not be minimal).

How can we answer those questions in constant time? We introduce a set of tools to perform this task:

Rectangle sums on a matrix.

The above approach generalizes to multiple dimensions. Given a matrix A, we wish to quickly answer queries of the form:

$$\text{sum}(r_1, c_1, r_2, c_2) = \sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} A_{i,j}$$

Again, precompute the partial sums matrix, namely

$$P_{r,c} = \sum_{i=1}^{r} \sum_{j=1}^{c} A_{i,j}$$

It follows that $\text{sum}(r_1, c_1, r_2, c_2) = P_{r_2,c_2} - P_{r_1-1,c_2} - P_{r_2,c_1-1} + P_{r_1-1,c_1-1}$.

Weighted interval sums on an array.

Going back to the array A, consider *weighted* queries of the form

$$\text{wsum}(i, j, k) = A_i \cdot k + A_{i+1} \cdot (k+1) + \cdots + A_j \cdot (k + j - i)$$

Precompute the array Q of partial weighted sums, that is

[1] Lee's algorithm on Wikipedia

$$Q_i = A_1 \cdot 1 + \cdots + A_i \cdot i$$

It follows that

$$Q_j - Q_{i-1} = A_i \cdot i + A_{i+1} \cdot (i+1) + \cdots + A_j \cdot j$$

Each term in this quantity differs from the corresponding term in $\text{wsum}(i,j,k)$ by a factor of $k-i$. Therefore,

$$\begin{aligned}\text{wsum}(i,j,k) &= Q_j - Q_{i-1} + (A_i + A_{i+1} + + \cdots + A_j) \cdot (k-i) \\ &= Q_j - Q_{i-1} + (P_j - P_{i-1}) \cdot (k-i)\end{aligned}$$

Weighted rectangle sums on a matrix.

We can combine the the previous two sections combine to quickly answer queries like

$$\text{wsum}(r_1, c_1, r_2, c_2) = \sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} A_{i,j} \cdot (k+i+j)$$

We precompute

$$Q_{r,c} = \sum_{i=1}^{r} \sum_{j=1}^{c} A_{i,j} \cdot (i+j)$$

after which

$$\text{wsum}(r_1, c_1, r_2, c_2) = Q_{r_2,c_2} - Q_{r_1-1,c_2} - Q_{r_2,c_1-1} + Q_{r_1-1,c_1-1} - k \cdot \text{sum}(r_1, c_1, r_2, c_2)$$

Triangle sums on a matrix.

We can apply the partial sums approach to right-angled triangles. For every point (r,c), we precompute the sum of the triangle $(r,c)-(1,c)-(1,c+r-1)$. Now, given an arbitrary triangle, we can extend it upwards to the first row, using a rectangle and a second triangle. We can then compute the sum of the original triangle by taking the difference.

Putting it all together.

We can always decompose our diamond as a set of triangles and rectangles that completely lie within the garden. As each of the precomputations above is done to answer sum queries on triangles and rectangles in $O(1)$, we can therefore find the answer for each parcel in $O(\log N)$ (from the binary search on the diamond radius), for a total complexity of $O(N^2 \log N)$. This approach is designed to solve Subtask 6 at least, but various implementations can also solve Subtasks 7 and 8.

Subtask 8 — $\mathcal{O}(N^2)$, solution proposed by Radu Voroneanu.

Let's take two adjoining parcels (r,c) and (r',c'), and denote by d the minimum diamond that contains at least K flowers for parcel (r,c) and by d' the same for parcel (r',c'). We will prove that $|d'-d| \leq 1$ by contradiction.

Assume, without the loss of generality, that $d' > d$, and that $d'-d > 1$. Then, consider diamond $(r',c',d'-1)$. Since $d'-d>1$, then $d'-1-d>0$, therefore diamond (r,c,d) is completely included in diamond $(r',c',d'-1)$. But then, since (r,c,d) already contains at least K flowers (by definition), it entails that $(r',c',d'-1)$ also contains at least K flowers. But this contradicts that (r',c',d') is the smallest diamond containing at least K flowers.

By using this observation, once we find the answer for a parcel (r, c) to be given by a diamond of size d, we can only query diamonds $(r', c', d-1)$, (r', c', d) and $(r', c', d+1)$ to find the answer for the adjoining parcels. We can do these queries either by triangle and diamond sums as in the Subtask 6 solution, or by cleverly removing and adding diamond side pieces as seen in the Subtask 3 solution, which leads to a faster solution in practice. Both solutions successfully solve this subtask.

Problem: Lock

Proposed by: Radu Voroneanu

We will start by describing how to calculate the minimum number of incS operations required for any permutation of the numbers 1 through N, defined as $A[1..N]$. Let B be the lock's displayed code, initially filled with N values of 0. The optimal method of obtaining A from B is to apply incS in turns, first increasing all required values to 1, then all required values to 2 and so on. More formally, at a step X (iterating from 1 to N), we will increment all values $B[i]$ for which $X \leq A[i]$, thus increasing them from $(X-1)$ to X. To minimise the number of operations for each step X, one single incS operation will be used for each continuous sub-string of values larger or equal to X.

Let us take as an example the permutation $[2,4,7,1,5,3,6]$. As a first step, we increase all values of B to 1 using incS$(1,7)$. Then we increase all numbers in B for which their A equivalent is at least 2. This can be done using two operations incS$(1,3)$ and incS$(5,7)$ and B thus becomes $[2,2,2,1,2,2,2]$. After, we increase all required values to 3, using incS$(2,3)$ and incS$(5,7)$ and B thus becomes $[2,3,3,1,3,3,3]$. Then, at step 4, we use incS$(2,3)$, incS$(5,5)$ and incS$(7,7)$ to make $B = [2,4,4,1,4,3,4]$. At step 5, we use incS$(3,3)$, incS$(5,5)$ and incS$(7,7)$ to make $B = [2,4,5,1,5,3,5]$. At step 6 we use incS$(3,3)$ and incS$(7,7)$ to make $B = [2,4,6,1,5,3,6]$. Lastly, we use incS$(3,3)$ to make $A = B$. In total, the minimum number of operations is 14.

To ease the explanation, we will extend A with 0's on both side - i.e. $A[0] = A[N+1] = 0$. For any step X, the number of incS operations required will be equal to the number of continuous sub-string of values larger or equal to X. This number is equal to the total number of ends of such sub-strings divided by 2, as each sub-string is determined by two ends (one left and one right). One such end is defined as two neighbouring values in A, one smaller than X and the other larger or equal to X. Now, viewing it the other way, you get that two consecutive values $A[i]$ and $A[i+1]$ will be the edges of a sub-string in all steps from $\min(A[i], A[i+1])+1$ to $\max(A[i], A[i+1])$ - or, in other words, a total of $\left|A[i]-A[i+1]\right|$ times. Using this, we can compute the total number of incS operations as:

$$\#\text{incS} = \frac{1}{2}\sum_{i=0}^{N}\left|A[i]-A[i+1]\right| = \frac{1}{2}\sum_{i=0}^{N}\Big(A[i]+A[i+1]-2*\min(A[i],A[i+1])\Big)$$

Let CNT be an array in which $CNT[i]$ represents the number of neighbours that value i has in A which are larger than i. For example, for the permutation $[2,4,7,1,5,3,6]$, we have that $CNT = [2,1,2,1,0,0,0]$ showing the fact that 1 has 2 larger neighbours (7 and 5), 2 has one single larger neighbour (just 4), and so on.

Firstly, the sum of values in CNT has to be equal to $N-1$. With the help of CNT, we can open the brackets in the above sum and obtain:

$$\#\text{incS} = \sum_{i=1}^{N} A[i] - \sum_{i=1}^{N} CNT[i]*i$$

Now, going back to the original problem, which asks that the number of incS operations is exactly equal to M. We can rewrite the equation above as:

$$\sum_{i=1}^{N} CNT[i]*i = \frac{N*(N+1)}{2} - M \tag{1}$$

We will define this as the CNT-sum of the permutation. We will look at the minimum and maximum value that this sum can take. The minimum CNT-sum is obtained putting as many larger values towards the beginning of the array - i.e. either $CNT = [2, 2, \ldots, 2, 0, 0, \ldots, 0]$ or $CNT = [2, 2, \ldots, 2, 1, 0, 0, \ldots, 0]$ depending on whether N is odd or even. The maximum CNT-sum can be obtained from $CNT = [1, 1, ..., 1, 0]$. One important observation is that any intermediary sum, between the minimum and maximum, can be obtained from a valid permutation.

Let $S = \frac{N*(N+1)}{2} - M$ be the required CNT-sum. We will now try to construct the permutation from left to right, trying to put the minimum possible value at each position.

We will first try to set $A[1] = 1$. For this to work, we need to set $CNT[1] = 1$, because $A[1]$ has one single neighbour. The minimum CNT-sum can be obtained by setting $CNT = [1, 2, 2, \ldots, 2, (1), 0, 0, \ldots, 0]$ depending on whether N is odd or even, and the maximum can be obtained by setting $CNT = [1, 1, \ldots, 1, 0]$. We can set $CNT[1] = 1$ if and only if S is in the determined by the minimum and maximum. If $minimum \leq S \leq maximum$, then we can set $CNT[1] = 1$ and therefore we can set $A[1] = 1$. if not, it means that $CNT[1] = 2$ and we will continue by trying to set $A[1] = 2$. This in turn would require that $CNT[2] = 1$ and the process is repeated until a valid value is found for $A[1]$.

Also in a similar manner, we we continue to try out the value of $A[2]$, then $A[3]$ and so on. While building, once we fixed the value for a position $A[i]$, we have to be careful at what $CNT[A[i+1]]$ can take. For example, if $CNT[A[i]] = 0$ (i.e. zero neighbours larger), then $CNT[A[i+1]]$ has to be either 1 or 2. Similarly, if $CNT[A[i]] = 2$, then $CNT[A[i+1]]$ has to be 0 or 1.

Let us take an example where $N = 8$ and $M = 13$. Thus, $S = \frac{8*9}{2} - 13 = 23$. We try to set $A[1] = 1$ and would need $CNT[1] = 1$. The minimum CNT-sum will be obtained when $CNT = [1, 2, 2, 2, 0, 0, 0, 0]$ and will have a value of $1*1+2*2+3*2+4*2+5*0+6*0+7*0+8*0 = 19$. The maximum CNT-sum will be obtained from $CNT = [1, 1, 1, 1, 1, 1, 1, 0]$ and will have a value of 28. We see that $19 \leq S = 23 \leq 28$ and conclude that therefore conclude that we can set $CNT[1] = 1$ and $A[1] = 1$. Moving forward, we try to set $A[2] = 2$. Again, the minimum CNT-sum that can be obtained is 22, obtained from $[1, 1, 2, 2, 1, 0, 0, 0]$. The maximum CNT-sum is 28, obtained from $[1, 1, 1, 1, 1, 1, 1, 0]$. Again, $22 \leq S \leq 28$ so we can set $CNT[2] = 1$ and $A[2] = 2$. We continue to try to set $A[3] = 3$. The minimum CNT-sum is $CNT = [1, 1, 1, 2, 2, 0, 0, 0]$ with a value of 24. We now see that $24 \not\leq S$, so we cannot set $CNT[3] = 1$. We therefore set $CNT[3] = 2$ and continue to try to set $A[3] = 4$ and $CNT[4] = 1$. The minimum CNT-sum can be obtained from $CNT = [1, 1, 2, 1, 2, 0, 0, 0]$ with a value of 23 and the maximum CNT-sum is obtained from $CNT = [1, 1, 2, 1, 1, 1, 0, 0]$ with a value of 24. Since $23 \leq S \leq 24$, it means we can set $CNT[4] = 1$ and $A[3] = 4$. We then try to set $A[4] = 5$ and $CNT[5] = 1$. The minimum and maximum CNT-sum will then both become 24, obtained from $CNT = [1, 1, 2, 1, 1, 1, 0, 0]$. Since $24 \not\leq S$, we therefore have to have $CNT[5] = 2$. From this point on, the only valid CNT is $[1, 1, 2, 1, 2, 0, 0, 0]$. We will try to set $A[4] = 6$ as the minimum possible value, which works since $CNT[6] = 0$. The process continues setting $A[5] = 3$ as the smallest valid remaining value, $A[6] = 7$, $A[7] = 5$ and lastly $A[8] = 8$. The minimum lexicographic permutation is therefore $[1, 2, 4, 6, 7, 5, 8]$.

The final complexity is $O(N)$. This can be obtained by updating the minimum and maximum CNT-sum in $O(1)$ at each step, using, for example, the formula of sum of consecutive numbers $a + (a+1) + ... + b = \frac{(a+b)*(b-a+1)}{2}$

There are a few last observations that are not required for the solution, but would ease implementation:

- Assuming a solution exists (i.e. S is in the range determined by the minimum and maximum CNT-sum before setting any CNT), then it is sufficient to check S only against the minimum CNT-sum.
- When constructing A and CNT, we observe that the first values in A will have a CNT value of 1, and then the remaining ones will have an alternating CNT value of 2 and 0.
- When constructing CNT, the last values will always be 0.

The described solution is a bit technical in order to show the correctness of the approach. Alternative solutions exist, by generating the permutations of a fixed N and varied M and observing similar patterns.

Problem: Wall

Proposed by: prof. Gheorghe Eugen Nodea

The problem asks for determining the best fragment for which $H_{max} \cdot L - Sum = S$, where H_{max} is the maximum height from the selected fragment, L is the width of the fragment (i.e. the number of towers in the fragment) and Sum is the sum of the heights of the towers in the fragment.

Subtask 1. We can set every fragment by selecting its left and right ends, then we can iterate through the towers between these indices and compute the sum of their heights and their maximum. This solution runs in $O(N^3)$ time complexity.

Subtask 2. For this subtask we need to slightly optimise the previous method. After we set the left end of the fragment we can increasingly iterate over all possible right ends, and at each step in order to update the sum and the maximum we only need constant time. Thus, this solution runs in $O(N^2)$ complexity.

Subtask 3. We can optimise the previous solution further by doing a binary search on the right end, after fixing the left one in place. Then for determining the sum on the fragment we can use precomputed prefix sums, while for the maximum we can employ a Segment Tree. This solutions runs in $O(N \log^2 N)$ time. Note that because of the design of the Segment Tree we can perform binary search directly on its structure obtaining an $O(N \log N)$ time complexity. Alternatively, we can replace the Segment Tree with a precomputed Range Maximum Query table and obtain the same $O(N \log N)$ time complexity. These last approaches may score the points from the last subtask as well, of course depending on implementation.

Subtask 4. For the last subtask we need to use the "two pointers trick" in order to find the best fragment. The first pointer will point to the left end of the fragment, while the second pointer will point to the right end. Initially both pointers point to the first tower from the wall. If the number of blocks needed to fix the current fragment is greater than the number of available blocks (i.e. $H_{max} \cdot L - Sum > S$) we move the first pointer one position to the right. Otherwise, if the number of needed blocks is $\leq S$ we move the second pointer one position to the right, also if the number of needed blocks is exactly S we have found a good fragment that needs to be compared with previously found ones for optimality (considering the priority defined in the task statement).

Maintaining the sum of the fragment towers is fairly easy (when increasing the left or the right pointer we need to do a subtraction or an addition, respectively).

The final complexity is thus determined by the means of computing the maximum height from the fragment:

- using a double-ended queue (deque): this structure is commonly used for maintaining the maximum/minimum for a sliding window, it stores a subset of the heights from the fragment sorted decreasingly; when a new height is added to the right (i.e. the second pointer is moved) the heights at the end of the deque are removed as long as they are smaller than the new one; when the left pointer is moved we need to make sure that the height of the "deleted" tower is removed from the deque, but it can only be on the first position at the front of the deque so it would cost $O(1)$ time to remove it; thus this approach runs in $O(N)$.
- using a max-heap or a STL set data structure: these structures allow us to perform insertion, deletion and maximum query in at most $O(\log N)$ time, so we can maintain such a structure to which we add a new height when the right pointer is moved, or

remove a height when the left pointer is moved; thus the time complexity for this approach is $O(N \log N)$.

- using Range Maximum Query: precomputing the RMQ table costs $O(N \log N)$ time but enables us to find in constant time the maximum from any segment; this solution runs in $O(N \log N)$.

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