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 $\mathrm{wsum}(i,j,k)$ by a factor of $k-i$. Therefore,

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

Weighted rectangle sums on a matrix.

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

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

We precompute

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

after which

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

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 $|d'-d| \le 1$ 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:

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

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 $a + (a+1) + \ldots + b = \frac{(a+b)*(b-a+1)}{2}$

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 Francu - Clubul Nerdvana București
- Cosmin Piț Rada - Bolt
- Ciprian-Daniel Cheșcă - “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