Inflation

EGOI 2023

Problem author: Pak Hei Chan.

The first task of Day 1 of EGOI 2023 is relatively straightforward, utilizing commonly used techniques that are widely recognized. Hopefully, completing this task will provide advantageous assistance for you in achieving your dream of earning a medal in EGOI 2023.

Subtasks may not be ordered for ease of discussion.

1 The Problem

Given an array $p_1, p_2, \ldots, p_N$, support the following two types of operations (there are Q operations in total):

- `INFLATION x`: Add p_i by x, for all $1 \leq i \leq N$.
- `SET x y`: For all i ($1 \leq i \leq N$) such that $p_i = x$, set $p_i = y$.

After each operation output the sum of the array.

2 Subtask 2: Small N and Q

Here, $N, Q \leq 100$. To solve this subtask, one can simulate the process described in the problem statement. Each of the two types of operations can be handled in $O(N)$ time, by going through the array and changing the values appropiately. Therefore, the overall complexity is $O(NQ)$.

Since $N, Q, p_i, x, y \leq 100$, C++ contestants do not have to worry about integer overflow problems.

Expected score: 28.

3 Subtask 1: $N = 1$

In this subtask, a $O(NQ)$ solution works as well. Here difference from subtask 2 is that contestants are not required to have knowledge about arrays. Also, C++ contestants have to be careful about integer overflow problems in this subtask (and all future subtasks): one need to use 64-bit integers (long long) to solve this subtask.

Expected score: 14 for just subtask 1, $28 + 14 = 42$ for subtask 1 and 2.

4 Subtask 3: Only `INFLATION`

In this subtask there are only `INFLATION` events. Notice that an event of the form `INFLATION x` adds the sum of the array by $N \cdot x$. Therefore, one can simply compute the sum of the array at first, then for every event `INFLATION x`, add the sum of the array by $N \cdot x$. There is no need to add each element of the array by x, and compute the sum of the array again.

Overall complexity: $O(N + Q)$.

Expected score: 19.

5 Subtask 4: Only `SET`

In this subtask there are only `SET` events. The key idea here is to try make our computations constant time. Notice that the indices in the array do not matter, what matters is the values.

Therefore, we will maintain a frequency array $f_1, f_2, \ldots, f_{10^6}$, where f_i counts the number of items in the array p with value i.

Let the sum of the array before an operation be s. After a `SET x y` event, the following happens:

- $s := s + f_x \cdot (y - x)$, update the sum of the array.
- $f_y := f_y + f_x$, all indices that have value x now have value y instead.
- $f_x := 0$, there are no more indices that have value x now.

There is a **special case** where $x = y$. In this case nothing happens. If one simulates the above process it will not work. Careful handling is required to pass this subtask.

Overall complexity: $O(N + Q + C)$, where $C = 10^6$ is the maximum value in the array.

Expected score: 23.

6 Full Task

To get the final 16 points, we merge the ideas of subtask 3 and subtask 4.

Consider maintaining a `global` tag that keeps track of the total value added to each element of the array. Initially, `global` $= 0$.

That means, for each element, if its initial value is p_i, its current value must be equal to $p_i +$ `global`.

This idea is important for understanding the following parts of the solution.

From subtask 4, we will maintain two things:

- Sum of the array s, ignoring global updates.
- Frequency array f. However, here the values can exceed the range $[1, 10^6]$. Hence, we should use a `unordered_map` (in C++) or dictionary (in Python).

For an `INFLATION x` operation:

- Add the `global` tag by x. There is no need to update s.

For an `SET x y` operation:

- We want to set all elements that have value x to have value y instead.
- For an element that currently has value x, its initial value must have been $x -$ `global`.
- These elements will then have a current value of y, which corresponds to an initial value of $y -$ `global`.
  - Look back at the key idea above if it doesn't make sense. If an element has an initial value of $y -$ `global`, its current value equals y, which matches the objective of the operation.
- Based on the above two points, we subtract the inputs x and y by `global`.
- Perform the steps from subtask 4, if $x \neq y$:
  - $s := s + f_x \cdot (y - x)$
  - $f_y := f_y + f_x$
  - $f_x := 0$

Finally, the output after each operation is $s + N \cdot$ `global`, since s does not take into account of global updates.

Time complexity: $O(N + Q)$, or $O(N + Q \log(N + Q))$ if a `map` is used instead of `unordered_map` in C++.

Expected score: $28 + 14 + 19 + 23 + 16 = 100$.

7 Hard Version

The originally proposed version of this problem was more difficult (but solvable). For those who are interested, you may try thinking of the solution.

Given an array $p_1, p_2, \ldots, p_N$, support the following **four** types of operations/ queries (there are Q operations in total):

1. Given i, x, do $p_i := p_i + x$.
2. Given x, do $p_i := p_i + x$ for all $1 \leq i \leq N$. (`INFLATION`)
3. Given i, output p_i.
4. Given x and y, for all i $(1 \leq i \leq N)$ such that $p_i = x$, set $p_i = y$. (`SET`)

8 Solution to Hard Version

Consider grouping the initial array elements into equivalence classes, such that in each equivalence class, all elements have equal value. The idea is to use four maps/arrays:

1. Value $\to$ equivalence class
2. Equivalence class $\to$ value
3. Equivalence class $\to$ set of indices
4. Index in array $\to$ equivalence class

By maintaining these four maps/arrays for every type of operation, it can be easily seen that all types of operations can be solved. The tricky part is in the `SET` operations, the essential thing that has to be done is to merge two equivalence classes. We can naively insert all elements from the smaller set to the larger set, then clear the smaller set. This technique is known as **small-to-large merging**, and it ensures the total number of insert operations is $O((N + Q) \log N)$.

Padel Prize Pursuit

EGOI 2023

Problem authors: Pavle Martinović and Mladen Puzić

1 Two players – $N = 2$

For each medal we can determine who won more matches after that day and thus receives this medal in the end. If we do this starting with the last match, we can do this for all medals in $O(M)$ so the total complexity is $O(N + M)$.

2 Simulation – $N, M \leq 2000$

We can simulate the process by going through matches and maintaining a list of medals for each participant. For each pair of participant and medal we keep the number of nights that that participant held that medal. Then for each medal, we determine who held it the longest. Complexity: $O((N + M)M)$.

3 The winner of the ith match participates in the $(i + 1)$th match for all i

For each medal, we need to determine the participant with the most wins. Observe that as the winner always plays the next match, all medals will always stay together. Like in Subtask 1, we can now go through the days starting at the last day, maintaining the number of wins and keeping track of which participant has the most wins. Complexity: $O(N + M)$.

4 At every match, the winner has at least as many medals as the loser

From the fact that the match is always won by the participant with the larger amount of medals at the moment, we can deduce that each medal will switch hands at most $O(\log M)$ times, as if we look at a fixed medal, the number of medals the person who holds this medal has doubles for each change of hands. Thus, we can simulate the process, but this time, in contrast to Subtask 2, we can not keep days held for each medal and participant. However, there will be at most $O(\log M)$ medal holders per medal, so we can just remember all of them and find who held it the most nights in the end. Complexity: $O(M \log M + N)$.

5 Once a participant loses, they are never in a match again

Construct a tree where each match is a node. The parent of node x is the next match where the winner of match x appears. Add a fake root, so we turn the forest into a tree. The lifetime of each medal is a path from its first match to the root. All matches of a participant are consecutive on the path from the match where she first appeared to the root. This means that we can do DFS on the tree, starting from the root, and maintain who held each medal the longest from the root to the current node and also the current medal holder. Complexity $O(N + M)$.

6 Full Score

Construct the same tree as in the Subtask 5. Participants' matches are no longer a single path in the tree. Instead of keeping just the current and the longest medal holder, we need to keep the count for every participant and also maintain the longest medal holder. This can be done by keeping track of how many medals on a path a participant has one and reversing the changes when going up in the DFS tree. Complexity $O(N + M)$.

Find the Box

EGOI 2023

Problem author: Nils Gustafsson.

The Problem

There is an $H \times W$ grid. In some unknown cell there is a box. Your goal is to find the box.

Every night, a robot starts in the top left corner, and moves around the grid. You can decide how the robot should move by giving it instructions in the form of a string consisting of characters `<`, `>`, `^`, `v`.

The walls are solid, so if the robot attempts to move outside of the grid, nothing will happen. The box is also solid, so you cannot move into the cell containing the box.

At the end of each night, the robot will report to you its location, and go back to the top left corner. Your task is to find the location of the box in at most Q nights. In particular, you get full score if $Q \leq 2$.

Solution 1: $Q = 50$ (20 points)

Notice that if the robot is on $(r, 0)$, and the box is on (r, c), where $c > 0$, you may instruct the robot to keep going right and it will eventually get stuck on $(r, c - 1)$.

On the other hand, if the box is not on row r, the robot reaches $(r, W - 1)$.

Therefore, you can brute force each row. For row r, move r times downwards, then move $W - 1$ times rightwards and see if the box is stuck.

But be careful of the special case when the box is in column 0.

When $H = 50$, this solution uses $H = 50$ queries, which scores 20 points.

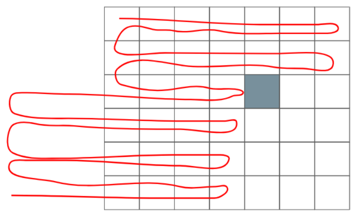
Solution 2: $Q = 3$ (82 points)

Using a similar idea to the previous solution, if we know which column the box is in, we can easily find its exact position by instructing the robot to go to the correct column, then moving down many times such that it gets stuck eventually.

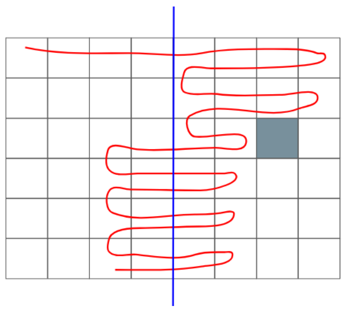
Therefore, we may consider focusing on finding the correct column of the box using 2 queries, then using 1 final query to find the position of the box. That adds up to 3 queries. Now, the question is how we can do it.

When encountering similar types of grid interactive problems, some common strategies contestants may consider include constructing some well-defined patterns (like spirals), or dividing the grid into parts, considering parities, randomized methods, etc. In fact, the first two methods are helpful with solving this task using $Q = 3$ queries!

Let's consider the following pattern: moving back and forth along each row (like in the figure below). When the robot encounters the box, we know that it gets stuck. As a result, the robot will be displaced to the left, revealing which column the box is in.

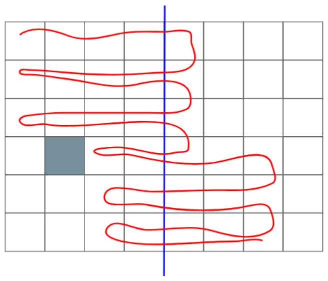


While this method seems promising, there is a small issue: the robot will run into the left wall and the information about the column will be lost. So instead, let's divide the grid into two halves and run this solution on both halves! The first query will look like this:



When the robot enters the second half of the grid, let's say it moves x units back and forth along each row. If the robot finally lands on $(H - 1, c)$ then it must have got stuck somewhere at column $c + x$, so the box would be at column $c + x + 1$.

Of course, the above solution assumes that the box is in the right half of the grid. If the box had been in the left half, we would need one more similar query to find the correct column:



There are also some special cases, like if the box is in the first row. But if you are careful enough, this will let you solve the problem in only three queries.

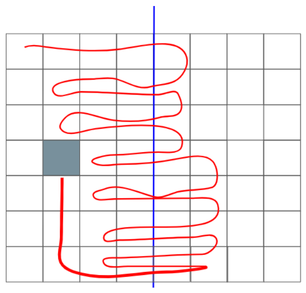
Solution 3: $Q = 2$ (100 points)

To get the number of queries down to 2, we will modify the $Q = 3$ solution.

The main idea is that we do not always need the final query to get the position given the correct column. Sometimes we can incorporate that into the query that finds the column.

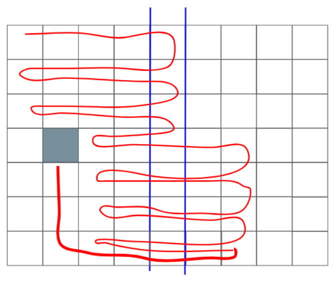
Let's start by making the same first query as in the $Q = 3$ solution. If the box was in the right half of the grid, we would find the column of the box using one query and find the row as well using another query. This takes 2 queries, so there is no issue.

The issue arises when the box was in the left half, we need to find it in only one more query. This is a bit tricky to do, and it is convenient to divide it up in two cases depending on whether W is even or odd. For even W, we can do this:



In other words, we find the column similar to how it was done in the previous solution, and then we also move back to the column and get the position of the box. This is possible thanks to the fact that we know that the box is not in the right half, so we can move freely there.

When W is odd, then this second query can look like this instead. Note that there are some special cases to be handled, such as when the box is in the middle column.



With careful handling, one may solve the problem using $Q = 2$ queries, which scores 100 points.

Aftermath

There are also many other ways to get full score on this problem. If you ask someone who solved it, then you will quite likely hear something different.

Can you prove that it is not possible to solve the problem with $Q = 1$?

Bikes vs Cars

EGOI 2023

Problem author: Nils Gustafsson.

Solution:

We can start by writing down what the numbers $C_{i,j}$ and $B_{i,j}$ mean in a more concise way:

$$C_{i,j} = \max_p(\min_b(W - b))$$

$$B_{i,j} = \max_p(\min_b(b))$$

Here, the `max` is taken over all paths from i to j, and the `min` is over all bike lane widths of edges on the path.

Note that if we let $A_{i,j} = W - C_{i,j}$, then we get rid of the parameter W, because

$$A_{i,j} = \min_p(\max_b(b))$$

So now we only focus on the bike lane width, and our goal is to construct a graph that satisfies all the `minmax`- and `maxmin`-constraints imposed by the numbers $A_{i,j}$ and $B_{i,j}$.

Subtask 1 and 2:

Here it is helpful to make the following observation:

Observation 1: The minimum edge weight in the graph is $min(A_{i,j})$, and the maximum edge weight is $max(B_{i,j})$.

In this subtask, this implies that if $B_{i,j} < A_{i,j}$, then it is impossible.

Otherwise, we can add edges of weight $A_{i,j}$ and $B_{i,j}$ that connect all pairs of vertices.

Subtask 3 ($N \leq 40$)

To solve the case when N is small, we need to find a construction that always works, but uses too many edges.

Observation 2: If there is a valid solution, then the following construction will also work: disregard all pairs of vertices where $A_{i,j} > B_{i,j}$. For all other pairs of vertices, add edges of weight $A_{i,j}$ and $B_{i,j}$.

Here is some motivation why this works:

First, if we have an edge between i and j of weight b, then $A_{i,j} \leq b$ and $B_{i,j} \geq b$, which means that $A_{i,j} \leq B_{i,j}$. So we can never have an edge between two vertices if $A_{i,j} > B_{i,j}$. This means that if a graph constructed as above is disconnected, then there is no solution.

Second, a path in the construction that minimizes the maximum weight between i and j will only use edges $A_{x,y}$ that we added, since they are always smaller than the $B_{x,y}$-edges. But it could happen that the minimum maximum weight ends up smaller than $A_{i,j}$, if there is a path $i, a_1, a_2, \ldots, j$ such that

$$\max(A_{i,a_1}, A_{a_1,a_2}, \ldots) < A_{i,j}$$

However, if such a path existed, then it could also be used in any other construction, so in this case the answer should be `NO` anyway.

Third, similar arguments can be made about the $B_{i,j}$-edges.

So all we have to do to get this subtask is to construct the graph above, and check that it is a valid solution. This check can be done by running an algorithm similar to Floyd-Warshall's, or by using minimum spanning trees.

Subtask 5 (all $B_{i,j}$ are the same)

Let B be the value of all $B_{i,j}$. Remember from subtask 1 that the maximum weight in the graph is the maximum value of $B_{i,j}$, which is B. So in this subtask, we must have that $B \geq \max(A_{i,j})$, otherwise there is no solution. But if this is the case, then we can connect all vertices with edges of weight B and then forget about the `maxmin` constraints.

After that, we only have to focus on the numbers $A_{i,j}$.

Observation 3: If we have a graph that satisfies all $A_{i,j}$-constraints, then the minimum spanning tree of that graph will still satisfy all the $A_{i,j}$-constraints.

This fact is a rather standard trick when it comes to minimum spanning trees. To see why it is true, assume that there exists a path from i to j such that the maximum weight is smaller than the maximum weight of the path along the minimum spanning tree. Then we could remove the largest weight edge on the path along the tree, and replace it with an edge of smaller weight. This would create a smaller spanning tree, which is a contradiction.

So, to solve the subtask, find a minimum spanning tree of the complete graph whose edge weights are $A_{i,j}$, and check that it is a valid solution. There are several algorithms to efficiently find a minimum spanning tree, like Prim's or Kruskal's.

Full score

To get full score, we will put together the things we learned in the previous subtasks.

First, take the construction from Subtask 3. Like we saw in Subtask 5, we can take a minimum spanning tree of this graph, and it will still satisfy the `minmax`-constraints. Similarly, we can take the maximum spanning tree to satisfy the `maxmin`-constraints. Furthermore, if we take the union of these two trees, then we get a solution, if one exists.

To see why this works, note that the minimum spanning tree will only use the $A_{i,j}$-edges added in Subtask 3, and the maximum spanning tree will only use the $B_{i,j}$-edges. Also, a path between i and j that minimizes the maximum weight will only use the edges from the MST, like we saw in Subtask 5, and this value is exactly $A_{i,j}$ like we want (and similarly for $B_{i,j}$).

Summary of how to get 100 points: Create a graph with edges of weight $A_{i,j}$ and $B_{i,j}$ between every pair of vertices such that $A_{i,j} \leq B_{i,j}$, take the union of the minimum and maximum spanning tree, and finally check that this is a valid solution.

Carnival Generals

EGOI 2023

Problem author: Nils Gustafsson.

The first task of Day 2 of EGOI 2023 utilizes a relatively simple idea, with approachable partial scores as well. Depending on the contestant, some subtasks may or may not be useful for thinking of the full solution.

Subtasks may not be ordered for ease of discussion.

1 The Problem

There are N generals numbered $0, 1, \ldots, N-1$ which you want to arrange in a row. The j-th general ranks all generals i such that $i < j$, and it is forbidden for two generals i and j $(i < j)$ to stand next to each other in the row if general i is **strictly** in the second half of general j's ranking.

Construct a way to arrange the N generals in a row such that the restrictions are satisfied. It is guaranteed that there is a solution.

2 Subtask 3: $N \leq 8$

In general, if one doesn't have the idea for the full solution, it is always nice to start with brute force.

It is easy to see that there are $N! = 1 \times 2 \times \ldots \times N$ ways to order the generals. When $N \leq 8, N! \leq 40320$. Therefore one can simply brute force all the possible orders and do checking. This brute force can be implemented recursively, or by iterating through all permutations (for example by using `next_permutation` in C++).

Note that when we say general i and general j cannot stand next to each other, that means general i cannot be directly to the left of general j, and vice versa.

Expected score: 29.

3 Subtask 1: $p_{i,j} = i - j - 1$

Maybe we can start with some smaller N and work our way to larger N. Let's just run the brute force written in subtask 3.

Let's say the brute force returns the lexicographically smallest solution. Then, for different N you should get the following constructions:

- $N = 2$: `0 1`
- $N = 3$: `0 1 2`
- $N = 4$: `0 1 2 3`
- $N = 5$: `0 1 2 3 4`

It seems that the solution to a certain N is $0, 1, \ldots, N-1$. It is easy to see why it works, as $p_{i,0} = i - 1$ and hence generals $i - 1$ and i can always stand next to each other.

Expected score: 11.

4 Subtask 2: $p_{i,j} = j$

Let's use the brute force again.

- $N = 2$: `0 1`
- $N = 3$: `1 0 2`
- $N = 4$: `1 3 0 2`
- $N = 5$: `2 0 3 1 4`
- $N = 6$: `2 5 0 3 1 4`
- $N = 7$: `3 0 4 1 5 2 6`
- $N = 8$: `3 7 0 4 1 5 2 6`

It seems that the solution to odd N is:

- Arrange generals $\frac{N-1}{2}, \frac{N-1}{2} + 1, \ldots, N - 1$ in that order
- Put general $0, 1, \ldots, \frac{N-3}{2}$ in between each pair of adjacent generals

The difference of the numbers of adjacent generals are either $\frac{N-1}{2}$ or $\frac{N-1}{2} + 1$. Since $p_{i,j} = j$, each general favors generals with smaller numbers. Hence it is easy to see that a difference of $\frac{N-1}{2}$ or $\frac{N-1}{2} + 1$ does not violate the constraints.

For even N, simply add general $N - 1$ in between the first two generals, which are general $\frac{N}{2} - 1$ and general 0 respectively. It is also easy to see that adding general $N - 1$ this way does not violate the constraints.

Expected score: 23, or 34 if both subtasks 1 and 2 are attempted.

5 Full Task

5.1 Induction

We can think of the construction in Subtask 1 in another way: start with the construction for $N = K$ and add general K somewhere in the row to find the correct construction for $N = K + 1$. This induction-like technique turns out to be useful in solving the full task.

5.2 Proving the Existence of a Solution

Note that if there is a way to insert general K in a row containing generals $0, 1, \ldots, K - 1$ currently (regardless of how they are arranged), and there is a construction for $N = 2$, then there is a construction for all valid inputs. This is because one can get the construction for $N = K + 1$ easily from $N = K$, by adding general K into the current row.

Now, how many ways are there to insert general K? Since there are K generals in the row currently, there must be $K + 1$ ways that general K can be inserted. Additionally, $\lceil \frac{K}{2} \rceil$ of the existing generals can be placed next to general K, while the rest cannot. Let's call the generals that can be placed next to general K "good" generals, and the rest "bad" generals.

We want to find one of the following:

- A pair of adjacent generals such that both of them are good (condition 1)
- Either the first or the last general that is good (condition 2)

Now, assume the contrary that neither of these two conditions are true. Then, the first and last generals are bad, and between every pair of good generals there is at least one bad general. There are $\lceil \frac{K}{2} \rceil$ good generals, so we need at least $\lceil \frac{K}{2} \rceil + 1$ bad generals for this to occur. But we also know that the number of bad generals equals $\lfloor \frac{K}{2} \rfloor$, and $\lceil \frac{K}{2} \rceil + 1 \neq \lfloor \frac{K}{2} \rfloor$.

We found a contradiction, hence the assumption that "neither of the two conditions are true" is false. Therefore, at least one of the conditions are true, and that means we can always find a way to insert general K.

5.3 Constructing the Solution

Start with 2 generals, then add each new general by exhausting each possible position the new general can be in.

The time complexity is $O(N^2)$.

Expected score: $11 + 23 + 29 + 37 = 100$.

6 Aftermath

Here are some questions you may think about after attempting this task, if you are interested:

- If we instead give K constraints in the form (a, b), such that a and b cannot stand next to each other, where $0 \leq K \leq \frac{N(N-1)}{2}$, is the general problem solvable in polynomial time? Alternatively, can you reduce it to a standard NP-complete problem?
- What if we impose additional constraints on the arrangement of the generals, such as requiring that certain pairs of generals must stand next to each other or that certain generals must be placed in specific positions?

Candy

EGOI 2023

Problem author: Yann Viegas.

1 The Problem

You are given an array $a_1, a_2, \ldots, a_N$ and two integers F and T. In a single operation, you are allowed to swap any two adjacent elements of the array. Find the minimum number of operations required so that the first F elements of the array sum to at least T.

2 Subtask 1: $N \leq 2, a_i \leq 100, T \leq 10^9$

Here, N can only take two values: either 1 or 2.

- Case $N = 1$: We cannot apply any operation and as $1 \leq F \leq N$ we know that $F = 1$. Thus it is enough to check whether or not $a_1 \geq T$
- Case $N = 2$: We can either swap the two numbers, or not swap them. We can just consider the two cases and see if one of them gives a correct solution.

Overall complexity: $O(1)$

Expected score: 6.

3 Subtask 2: $a_i \leq 1$

In this subtask, a_i is either 0 or 1 for all i. First we will try to find if we can reach the objective without minimising the number of swaps. Let's say there are X ones in the array. The largest possible sum would be achieved by moving all the ones at the beginning of the array. The sum of the first F elements of the array will then be $s = \min$(F,X). If $s < T$, the answer is `NO`. Else, we can construct an answer by reaching a state where there are at least T ones at the beginning of the array. It is optimal to chose ones in increasing order of their indices in the array.

Expected score: 19 for just subtask 2, $6 + 19 = 25$ for subtasks 1 and 2.

4 Subtask 3: $N \leq 20$

In this subtask, $N \leq 20$ which suggests some sort of backtracking/bitmask solution.

We can split the array in two parts:

- the left part (first F elements)
- the right part (everything except the first F elements)

Fixing the set of the elements that will end in the left part will be enough. Let's say that their indexes (in the original state of the array) are:

$$l_0 < l_1 < \ldots < l_{F-1}$$

The minimum required number of swaps to move them to the left part is:

$$(l_0 - 0) + (l_1 - 1) + \ldots (l_{F-1} - (F-1))$$

Indeed, it is never useful to swap l_i and l_j when $i < j$. Furthermore, as l_0 has to end up in the first position, it needs to be swapped with all the elements to its left, same argument for l_1 which need to be swapped with all the elements with indexes in $[1 \mathinner{..} l_1[$ etc. Thus, all these swaps are necessary. This amount of swaps is sufficient (applying them is enough to move $l_0, l_1, \ldots, l_{F-1}$ to the left part.

One way of implementing this would be by iterating over bitmasks to iterate through all the subsets of $[0 \mathinner{..} (N-1)]$
Overall complexity: $O(2^N \cdot N)$.

Expected score: $6 + 16 = 22$ (it solves subtasks 1 and 3).

5 Subtask 4: $a_i \leq 100$

The constraints of the problem suggest that we use DP. Also, the solution of subtask 3 presents clearly a process of "chosing/not chosing" elements which is definitely something that could be solved with DP.

The states are:

- The index of the integer we are currently considering
- How many elements we already selected to be in the left part
- The sum of the elements currently in the left part

So $dp[i][j][s]$ will give the minimum number of swaps required to select the first j integers of the left part while having only used the first i integers and such that their sum is exactly s.

The transitions are pretty simple:

- Either we choose the ith integer to be in the left part

- Or we don't choose it to be in the left part

We use the formula of subtask 3 to update the number of swaps while doing our transitions.

Overall complexity: $O(N^2 \cdot \max_{0 \leq i \leq N-1}(a_i))$

Expected score: $6 + 19 + 30 = 55$.

6 Full Task

Let's try to optimise the DP solution we got for the previous subtask. The thing that makes it slow is that the sum of the integers of the left part can be extremely high. On the other hand, the value that our DP computes is small. Indeed, by the formula we found in subtask 3, the number of swaps required to move all the selected integers to the left part is bounded by N^2. Thus we can change a bit our states:

- The index of the integer we are currently considering
- How many elements we already selected to be in the left part
- The minimum number of swaps required to move the selected elements to the first part

Given such a state, we would like to maximise the sum of the integers selected to be in the left part. Thus, $dp[i][j][s]$ will give the maximal sum we can reach by selecting the first j integers of the left part while having only used the first i integers and such that the minimum number of swaps required to move them to the left part is exactly s.

Overall complexity: $O(N^4)$

Expected score: 100

Sopsug

EGOI 2023

Problem author: Jakub Tarnawski.

1 The Problem

You are given a graph with N nodes, M good directed edges and K bad (forbidden) directed edges. Build a directed tree with any root, such that all edges are directed towards the root, all good edges are used, and none of the bad (forbidden) edges are used.

We denote by (u, v) an edge that is directed from u to v.

2 Subtask 1: $M = 0, K = 1$

This is an ad-hoc subtask; its solution is not particularly instructive for the full solution.

There aren't any edges that you must use, and there is exactly one bad edge that you can't use. Consider the following two trees, both of which are directed chains:

- A tree such that all edges are $(i, i + 1)$. (Think about "going right" if all nodes are placed on a line.)
- A tree such that all edges are $(i + 1, i)$. ("Going left".)

At least one of the trees does not contain the bad edge. You can just figure out which one, and output it.

3 Subtask 2: $M = 0, K = 2$

This is an ad-hoc subtask; its solution is not particularly instructive for the full solution.

Similar to Subtask 1, you may consider finding a directed chain such that the edges in the directed chain do not contain the bad edges. An easy way to achieve this without much thinking is to generate some random chains, then check whether the generated chain works. Since the number of bad edges is small, after a few iterations there will be a chain that satisfies the conditions.

There is a special case: when $N = 2$, there is no solution.

Another solution is to find a node that has no incoming bad edge; since $K = 2$, if $N \geq 3$ there must be such a node. Then connect all other nodes directly to that node.

4 Subtask 3: $K = 0$

There are no bad edges, so one simply needs to construct a directed tree that uses all the good edges.

If the good edges form a directed forest, then it is possible. Otherwise, it is impossible. The exact conditions for the edges to form a directed forest are:

- Each node should have out-degree at most 1.
- There should be no cycles.

One can check acyclicity using a graph traversal method such as DFS. Alternatively, one can use a Union-Find data structure; whenever processing a new edge (u, v), first check if u and v are not already in the same component (in which case this is not a directed forest).

If the answer is "possible", simply pick one of the trees in the forest and connect the remaining trees' roots to any node in that tree.

5 Subtask 4: $N \leq 100$

We begin from the solution to Subtask 3. First, note that the good edges must still form a directed forest for there to be a solution. Next, we look at the resulting forest (where some trees are perhaps just single nodes). It is not hard to see that any final solution must have the following shape: we must choose one of the trees in the forest as the "root", and then connect each other tree in the forest to the "root" tree (perhaps indirectly). Note that any new edge will need to go from *the root* of a tree to *any node* of another tree.

We will try every possible tree as the "root" tree. So let us fix a "root" tree. We perform a DFS starting from the root of the "root" tree. Note that edges have to be reversed in order to be able to traverse the tree. When we visit a node, we try to connect any of the roots of trees that aren't connected yet. This means checking if the appropriate edge is not forbidden. If we succeed, great; we should then also proceed with the DFS from the newly-connected tree root. If we do not succeed, then we have seen a forbidden edge.

What is the time complexity of this, for a fixed "root" tree? Note that we traverse tree edges at most $N - 1$ times (each traversal corresponds either to an existing forest edge, or to a new edge with which we “succeed” in connecting a new tree root). Moreover, we make at most K attempts to connect a new tree root that do not ”succeed” because the considered edge was forbidden. Each operation should be dominated by the cost of checking if an edge is forbidden, which should be at most $O(\log N)$.[1] Therefore we have $O((N+K)\log N)$ total, per “root” tree.

Finally, checking every possible “root” tree contributes another N factor, which is good enough for this subtask.

As we actually have $N \leq 100$ here, one can even afford to explicitly construct the entire ”complement” graph consisting of non-forbidden edges (of which there are at most $N^2 - K$) and run DFS on it appropriately. The time complexity of such an approach would be $O(N^3)$.

6 Subtask 5: Exists solution with 0 as root

As we only need to check one “root”, we do not incur the extra N runtime factor, so the solution from Subtask 4 is fast enough even for $N = 300\,000$. (However, explicitly constructing the complement graph would no longer be possible, and one has to be somewhat careful not to run into some quadratic-time behavior.)

7 Full task

We need to speed up the solution from Subtask 4. What is a good “root”, actually? It might be easiest to first think of the case where there are no good edges ($M = 0$). Then, a good “root” is a node from which every node is reachable in the complement graph (i.e., the complete graph from which all forbidden edges have been removed, and then all remaining edges were reversed). Suppose we run DFS from node $v_0 := 0$; once it’s done, check if every node was visited. If yes, then 0 was a good root. Otherwise, run another DFS from some yet-unvisited node v_1, without resetting the ”visited” status of any nodes. Continue doing this: in each iteration, if not all nodes have been visited, run another DFS from some yet-unvisited node, and so on. Eventually we will have visited all nodes. Let us pay special attention to the last node v_k from which we ran DFS.

Is v_k a good root? Well, perhaps not – it might, for example, be an isolated node. However, surely every node that was visited in previous DFS iterations (before we started from v_k) was *not* a good root, as v_k was not visited in these iterations. On the other hand, all the other nodes – those first visited in the last DFS iteration – are reachable from v_k. Therefore, either v_k is a good root, or there is no solution. To check whether v_k is a good root, we reset the ”visited” status of every node, and run one last DFS from v_k (building the directed tree if possible).

[1] This can be brought down to $O(1)$, e.g. using a hash-set, such as `unordered_set` in C++.

This solution idea can be seen as being inspired by Kosaraju's algorithm for finding strongly connected components of a directed graph (the one where we run a DFS, and then another DFS on the graph with reversed edges).

Now, when $M > 0$, for the proof of correctness one can think of contracting every tree in the forest (compressing it into one node) while handling the edges appropriately. For the implementation, we just run the solution from Subtask 5, but then proceed with further DFS runs if not all nodes were visited from 0, as above.

The time complexity analysis goes through just as for Subtasks 4–5.

Subtask 6 ($M = 0$) is used as a safety net for the contestants who had the idea for the full solution, but implemented something wrongly for the good edges, e.g. checking acyclicity.

Guessing Game

EGOI 2023

Problem author: Edward Xiao.

Solution 1: $K = N - 1$ (10 points)

On the first $N-1$ houses, Anna draws the numbers $1, 2, \ldots, N-1$ in order of visiting. The number Emma draws must also be one of $1, 2, \ldots, N-1$, and result in a duplicate value, and it will be the only duplicate value since all the other values are unique. Thus, we find the pair of houses that contains this duplicate value and guess those two houses.

Solution 2: $K = N/2$ (30 points)

Split the houses into two halves. For each half, we perform the same strategy as above. If either half contains a duplicate, we guess those. Otherwise, we guess the largest number in each half, since these were the last ones to be picked.

Solution 3: $K = \lceil N/3 \rceil + 1$ (36 points)

Note that we can actually improve on the strategy above by splitting the houses into thirds. For each third, repeat the strategy from $K = N - 1$ for all numbers except the last. For the last houses of the blocks, we pick two values x and y. Then, the strategy is simple again: if there are duplicates of x or y, guess those and return. Otherwise, there must be duplicates of the smaller values in one of the thirds, so we will guess those.

Solution 4: $K = \lceil \sqrt{N} - 1 \rceil + \lfloor \sqrt{N} - 1 \rfloor$ (60 points)

We can generalize the previous solution even further by splitting the houses into $\lfloor \sqrt{N} \rfloor$ blocks and picking a set X of $\lceil \sqrt{N} - 1 \rceil$ numbers for each number that is not the last in its block. For the last number in each of the $\lfloor \sqrt{N} \rfloor$ blocks, we assign them a different set Y of $\lfloor \sqrt{N} - 1 \rfloor$ numbers. If there is a duplicate of an element in Y, we guess those. Otherwise, there must be a duplicate of an element in X in one of the blocks, so we guess those instead.

Solution 5: $K = \lceil \log_2 N \rceil$ (90 points)

For 90 points, we will use divide and conquer to split the houses into multiple layers of blocks instead of just one layer. Consider a segment tree layout over the houses. For each index, we say it "completes" a segment in the segment tree if it is the last index in that segment to be assigned a value. In each round, Anna picks $v =$ the lowest depth of any segment that i completes. For Bertil, consider the values written on the doors assuming Anna picks the last value according to their strategy. In this case, Bertil can just find 1 and return its index. However, if there is no 1 in the array, then there are two cases:

1. The 1 was replaced by a 2. Then there are exactly two houses with 2s written on them, and we know that one of them must be Anna's, since there should only be one 2 if the strategy was followed for all indices. This is because the index that was picked later not only completes the segment on layer 2, but also necessarily completes the segment on layer 1 since the other half is already completed.
2. The 1 was replaced by a value greater than 2. Then, locate the half that contains the only 2 in the array. We know Anna's house must not be in this half, since it was completed first. Thus, we may recursively solve the problem on the other half, completing the solution.

Note that since 1 is never actually picked by Anna, we can subtract 1 from all values Anna picks to ensure we have exactly $K = \lceil \log_2 N \rceil$.

For illustrative purposes, consider an example where $N = 8$, Emma's house has index 2, and Emma and Anna visit the houses in the following order: $[0, 5, 4, 1, 6, 3, 7]$. Then, after Emma writes a number X on her own house, the array A will be $[3, 2, X, 3, 2, 3, 3, 1]$. We run through Bertil's strategy in the 3 possible scenarios:

1. $X = 1$: since there are duplicate 1s, it is clear that one of these must be for Emma's house. Guess $2, 7$ and return.
2. $X = 2$: since there is a unique 1, Emma's house must be in the half of the array that does not contain 1. Now we get to $A[0, 3] = [3, 2, 2, 3]$. Repeat our logic recursively. Since there are duplicate 2s, it is clear that one of these must be for Emma's house. Guess $1, 2$ and return.
3. $X = 3$: since there is a unique 1, Anna's house must be in the half that does not contain 1. Now we get to $A[0, 3] = [3, 2, 3, 3]$. Since there is a unique 2, Anna's house must be in the half of the array that does not contain 2. Now we get to $A[2, 3] = [3, 3]$. Since there are duplicate 3s, it is clear that one of these must be for Anna's house. Guess $2, 3$ and return.

Solution 6: $K = O(\log\log N)$ or $K = O(2^{\log^* n})$ (100 points)

To solve the problem fully, we consider the following two-phase approach:

In the first phase Anna just writes the number K on the first $N - \Theta(\log(n))$ houses which are visited. In the second phase (which we describe later), we promise that Anna will not write the number K on the remaining $\Theta(\log(n))$ houses.

Now we need to handle two cases:

- If Emma does not write the number K on her house, then we know it is one of the $\Theta(\log(n))$ houses not with a K.
- If Emma writes the number K, then we need to figure out which of the $N - \Theta(\log(n))$ houses with K written on it belongs to Emma.

In the former case, we have essentially reduced the problem to an instance with $N' = \Theta(\log(n))$, but now with numbers $1, \dots, K-1$.

For the latter case, we make the following observation: If we known the sum S of indices, modulo N, of the $\Theta(\log n)$ houses not part of the first phase, then we can figure out Emma's house in case she writes K. Indeed, we can identify that we are in the second case by counting the number of houses with K written on them. Then, we know that Emma's house index plus all indices of houses without a K, must equal S. Hence we can solve for Emma's house index.

The strategy for the second phase is now the following: we want to essentially solve an instance where $N' = \Theta(\log N)$ while simultaneously encoding the sum S (which is known at the start of the second phase).

One has to be a bit careful on how to encode the sum S, in the remaining $N' = \Theta(\log n)$ houses, since one of these houses we do not have control over and Emma can write K on it instead. One way of doing it is to write S (modulo N) in binary and then duplicate each bit, to make sure that we can recover the sum S even after one bit (the one corresponding to Emma's house) is dropped.

Solving the instance on $N' = \Theta(\log(N))$ in our strategy can be done in multiple ways:

- Using Solution 1 ($K = N' - 1$) for the recursive part, which will end up in ≈ 76 points with $K = \Theta(\log n)$.
- Recursively using our two-phase strategy here, which will result in ≈ 96 points and $K = \Theta(2^{\log^* N})$.
- Using Solution 5 ($K = \lceil \log_2 N' \rceil$), which happens to be quite good with small N. If implemented carefully this will obtain the full 100 points using $K = 7 = O(\log\log n)$, see below for details.

Detailed optimization tricks for 100 points:

- Use $K = 7$ and $N - 32$ numbers for the first phase (so $N' = 32$).
- In the second phase we run Solution 5 with $N' = 32$, which will use numbers $1, \ldots, 5$.
- Instead of encoding the sum S modulo N, we use the fact that we have two guesses and encode it modulo $N/2$. This requires 16 bits of information, since $2^{16} = 65536 > N/2$.
- Note that Anna will write exactly 16 "5"s in the second phase (see Solution 5 for more details). Hence we can change some of these "5"s to "6"s to encode the sum S.
- All in all, the solution becomes quite succinct: the jury's 100pt solution is just 40 lines of python code!

Harder Version: $K = O(1)$

It is possible to solve the problem with K being constant (that is not growing when N grows). The jury is aware of a solution using $K = 5$ no matter how large N is. We leave finding a solution with constant K (or even as low as $K = 5$) as a difficult bonus challenge.

The jury can also prove that using $K = 3$ is impossible (even for small N); so what about $K = 4$, is it possible or impossible?

How does the grader work?

The grader for guessinggame is a bit complex. It might run Anna's part twice to try to predict what would be a confusing number for Emma to write on her house.

In the grader, the order of houses is fixed per testcase, but the value Emma writes on her own house is chosen adaptively. The second line of each .in file explains how the value gets chosen for that test case:

- `random`: the last line of the input file contains a number X. The value is chosen as X
- `interact`: same as random, except that the judging is performed interactively, instead of non-interactively for performance. This is an implementation detail which should not matter. See includes/ for more details if curious.
- `copy`: the last line of the input file contains a number X. The value is copied from house with index X.

- `fork`: the last line of the input file contains a number X. Phase 1 of the submission is run as a trial run with Emma's house placed at (0-based) position X of the house visit order, and the house that was originally at position X removed. Let the number that Anna writes on that house be Y. Then in the real run of phase 1, Emma writes the number Y on her house.

  That is Emma, essentially asks Anna "if we had visited my house at time X, what number would you have written on it?".

The grading is performed by adding a file that gets compiled/run together with the submission and uses the fork() function to make it run multiple times.