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

- `INFLATION x`: Add p_i by x, for all $1 \le i \le N$.
- `SET x y`: For all i $(1 \le i \le N)$ 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, $N, Q \le 100$. 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 $N, Q, p_i, x, y \le 100$, 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 $\rightarrow$ equivalence class
2. Equivalence class $\rightarrow$ value
3. Equivalence class $\rightarrow$ set of indices
4. Index in array $\rightarrow$ 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 $O((N+Q)\log N)$.