The 2022 ICPC Caribbean Finals Qualifier

Editorial

Problem set developers

Alberto González Rosales - Volvo Cars
Alejandro Jiménez Fabián - Beroly S.A.
Anier Velasco Sotomayor - Harbour Space University
Carlos Joa Fong - Orange Dominicana
Daniel Enrique Cordovés Borroto - Harbour Space University
Ernesto Teruel Velazco - Universidad de las Ciencias Informáticas
Javier E. Forte Reyes - Universidad de las Ciencias Informáticas
Marcelo J. Fornet Fornés - Aurora Labs
Mariano Jason Rodriguez Cisnero - Universidad de Oriente
Ernesto David Peña Herrera - Harbour Space University
Reynaldo Gil Pons - University of Luxembourg
Roberto Carlos Abreu Díaz - Meta
Rubén Alcolea Núñez - Universidad de las Ciencias Informáticas

March 1st, 2023

2022 ICPC Qualifier Real Contest Problem Set

A Again? Solving Queries?

Authors: Rubén Alcolea Núñez - UCI and
Ernesto Teruel Velazco - UCI
Category: Data Structures 3
Unsolved problem

In this problem you have an array of size $1 \leq n \leq 10^5$, which initially has all elements equal to zero. You need to perform $1 \leq q \leq 50000$ queries of the following types:

- *1 i j v*: Increase the numbers between indices i and j (inclusive) by v. It is guaranteed that $1 \leq i \leq j \leq n$ and $1 \leq v \leq 1000$.
- *2 i j*: Replace the numbers between indices i and j (inclusive) with the increasing sequence $[1, 2, 3, \ldots, j - i + 1]$. It is guaranteed that $1 \leq i \leq j \leq n$.
- *3 i j*: Count how many numbers between indices i and j (inclusive) are divisible by 5. It is guaranteed that $1 \leq i \leq j \leq n$.

Due to the input sizes for n and q, it is not possible the simulation of queries with a naive method. It would take $\mathcal{O}(n \cdot q)$ which is not enough to pass the time limit for this problem.

This problem is a classical example of range queries problems. In this kind of problems, where we need to apply efficiently an operation to a given range, we usually need to use data structures specialized in the work with ranges like segment trees, binary indexed trees or sqrt decomposition.

In this problem, the size of input allows solutions using segment trees and sqrt decomposition pass the time limit. Below, it will be explained the most important hints about these solutions.

Solution with sqrt decomposition:
The solution based on sqrt decomposition splits the range $[1..n]$ in blocks of size $\sqrt{n}$ to speed up the processing of ranges. Each block stores the following information:

```
struct Block {
  int mod[5];
  int sum;
  int start;
};
```

- The attribute *mod* stores frequencies of remainders when dividing by 5 (0 to 4) for all numbers inside the block. This attribute is used to determine

the multiples of 5 for a block in constant time. When dealing with sums and multiples, it's needed to store this information for all possible remainders. In this way, no matter the sum is added to the block, it's possible to compute the right number of multiples of 5.

- The attribute *sum* stores the cumulative sum of the block and is used to handle queries of type 1. For this attribute, it's enough to store only the remainder of *sum* divided by 5 to compute the right rotation of frequencies. For example, when $sum \mod 5 = 0$, the multiples of 5 in the block is the frequency of remainders with value 0. However, when $sum \mod 5 = 1$, the multiples of 5 in the block is the frequency of remainders with value 4.

- The attribute *start* stores an integer that represents the beginning of the interval for which the block was updated last time with query 2 (replace).

It's also important to consider the following points:

1. Perform a query of type 2 (replace) disables the effect of any pending query.

2. If one block has pending operations for queries 2 (replace) and 1 (sum), the query 2 (replace) should be performed first (See point 1).

3. If a range is processed to perform a query of type 3 (count), the tail blocks might need to be rebuilt to update the current information.

This solution takes $\mathcal{O}(\sqrt{n})$ per *query* in the worst case and the final complexity is $\mathcal{O}(q\sqrt{n})$.

Solution with segment tree:
The solution based on segment tree uses lazy propagation due to the nature of queries 1 (sum) and 2 (replace) that modify a range. Each node of the segment tree stores the following information:

```
struct node {
  int mod[5];      // frequencies of remainders for numbers at the range
  bool lazy_set;   // lazy for queries of type 2 (replace)
  int lazy_add;    // lazy for queries of type 1 (sum)
};
```

The segment tree uses almost the same points discussed for the sqrt-based solution. However, there are some points to consider that are specific for the segment tree:

- When dealing with queries of type 2 (replace), the step of updating the frequencies of the remainders can be tricky. Please, be careful with the remainders.

- During the merge of two nodes, the frequencies of remainders are added. A possible way to do it is to overload the sum operator (+) for node data type.
- During the propagation of nodes, take into account that queries of type 2 (replace) should be performed before queries of type 1 (sum).

The final solution has two stages: the construction of the segment tree ($\mathcal{O}(N)$) and the processing of queries ($\mathcal{O}(Q \log N)$). The time complexity is $\mathcal{O}(Q \log N)$.

$$\begin{aligned} T(n) &= \mathcal{O}(N) + \mathcal{O}(Q \log N) \\ &= \mathcal{O}(Q \log N) \end{aligned}$$

B Beyond Connectivity

Authors: Ernesto David Peña Herrera - Harbour Space University and Alberto González Rosales - Volvo Cars
Category: Graph Theory 2
Solved by 26 teams

We should notice that for every query, the solution is either 1 or 2.

1. If the edge (a, b) doesn't exist in the graph G, then it exists on the complement graph G'. Therefore, the answer is 1.
2. If the edge (a, b) exists in the graph G, then it doesn't exist on the complement graph G', meaning that the solution cannot be 1. But, since G is non-connected, at least two connected components exist.

   Let's take a node c that belongs to a connected component different from the one that a and b belong to. The edges (a, c) and (b, c) don't exist in G, otherwise a, b, and c would belong to the same connected component.

   The fact that (a, c) and (b, c) are not present in the graph G means that they are present in G' and the path $a \longrightarrow c \longrightarrow b$ exists. Therefore, the solution is 2.

To determine if an edge exists on the graph, we can use data structures such as the built-in implementation of Set that most languages have. That will make the algorithm run in a time complexity of $O(q \log m)$.

C Computing Near-Death Experience

Authors: Alejandro Jiménez Fabián - Beroly S.A. & Daniel Enrique Cordovés Borroto - Harbour Space University

Category: Dynamic Programming 5
Solved by 1 team

In this problem you need to find the minimum number of shots (circles with radius r) that cover all the points given in the input. This problem is one of the hardest of the contest according to the judges. The solution requires knowledge from areas like dynamic programming and computational geometry.

The first observation is that the number of points given in the input is small ($1 \leq N \leq 20$). It allows to design an algorithm using dynamic programming and bitmasks to find the optimal solution. The algorithm proposed by the judges has three steps:

1. Determine the sets of points that can be covered with a single circle of radius r.
2. Find the maximal sets for each bit.
3. Apply dynamic programming using maximal sets to compute the optimal solution.

The first step computes for each subset of points, if it's possible to cover them with a circle of radius r. This step requires finding the minimum enclosing circle for the points of the subset. Each subset can be represented as a bitmask and the time complexity of this step is equal the number of different bitmasks, that is $\mathcal{O}(2^N - 1)$. A subset can be covered by the circle if the radius of the minimum enclosing circle is less than or equal to the radius of the circle shooting machine.

The second step finds for each bit, the maximal sets that contain it. Each bitmask is tested to see if it covers all the points. The bitmasks covering all the points are considered maximal sets and are stored for each bit. This step takes $\mathcal{O}(2^N \times N)$.

The third step applies dynamic programming using the maximal sets computed at the second step. It means, for each bitmask, we try to improve the current solution by adding the maximal sets belonging to the least significant bit of the bitmask with value 0. It takes $\approx \mathcal{O}(2.45^N)$.

Optimization:
It can be proven that there are not more than $\mathcal{O}(N)$ maximal sets. The proof is left as an exercise to the reader.

It means it's possible to improve the current solution for each bitmask through the maximal sets. The time complexity of this solution is $\mathcal{O}(2^N \times N)$.

D Domino

Author: Marcelo J. Fornet Fornés - Aurora Labs
Category: Ad Hoc 1
Solved by 121 teams

Easiest problem of the contest. The solution verifies if the last piece of Fito matches at least one of the open heads of the board. If there is a match, print "YES", otherwise print "NO".

E Enjoy the Chat Group

Author: Roberto Carlos Abreu Díaz - Meta
Category: Sorting Searching 2
Solved by 8 teams

To solve this problem, there are essentially two things we need to do:

1. Find the last message that was sent at or before the access time T_q, and
2. Compile the list of users who last saw the identified message when an access to the group is performed at T_q

Let's solve the first part. There are two cases to consider:

1. $T_q < tm_1$. That is, the access time is performed before any message has yet been exchanged. Since there'd be no message to display icons for, there couldn't be a set of viewers either. In this case, there's nothing to display.
2. $T_q \geq tm_1$. That is, the access time is performed at or after the first message. If T_q matches the time any message was sent at, then that message is the one we have to show read notification icons for. Otherwise, there must be a message k sent before T_q that minimizes $T_q - tm_k$ and that is the message to show read notifications for.

The second case can solved with the binary search algorithm modified to retrieve the lower bound for a given value V. In a sorted data structure, the lower bound of V is the minimum element among the range of elements that are at least as large as V. So, by searching for the lower bound of T_q in the list of messages we will either find a message who was sent also at T_q, or we will pick the message k with the largest time such that $tm_k < T_q$. This operation can be performed in $\Theta(\log n)$.

Having identified the message of interest k, we can now attempt to compile the list of users who last saw the message when a group access is performed at time T_q. One of the following statements must be true for a given user:

1. T_q exists in their list of accesses,
2. Their greatest access not exceeding T_q occurs before tm_k,
3. The greatest access not exceeding T_q occurs after at or after tm_k.

The first case is trivial. Since we have already associated message k to T_q, and T_q exists in the user's access times list, then we can conclude that message k is the last message the user saw. In the second case, the closest access occurs before tm_k, so the user could not have seen message k. In the last case, the user must have an access in the $[tm_k, T_q]$ range. So, we can infer that the user did see the message and we should add the user to the result.

So, we can loop through the list of users and perform these kinds of queries in order to populate the answer. The total complexity will be $\Theta(\log n) + \Theta(v \log m)$.

F Finding the Origins

Authors: Alejandro Jiménez Fabián - Beroly S.A.,
Reynaldo Gil Pons - University of Luxembourg and
Daniel Enrique Cordovés Borroto - Harbour Space University
Category: Graph Theory 4
Unsolved problem

The problem asks to find the number labelled of directed acyclic graphs G with a fixed dominator tree T that is given, with the extra condition that G contains T. Notice that for every directed edge $u \to v$ in $G \setminus T$ we have that the lowest common ancestor of u and v must be the parent of v in the tree. Adding these type of edges doesn't break the dominator property, but may add cycles. In order to count in how many ways this can be done, the computation can be done separately for each subtree. The product of this value for all subtrees of T gives the required answer. The value for a single subtree can be computed in $\mathcal{O}(3^d \cdot d)$ if d is the degree of the root of the subtree.

G Go Kill the Monsters

Author: Anier Velasco Sotomayor - Harbour Space University
Category: Dynamic Programming 2
Solved by 17 teams

In the worst case, each time that we press the button, the half consisting of the monsters with the smallest strength will be killed.

If we decide to kill a monster by hand, then it's optimal to kill the one with the smallest strength among the current ones.

Then, we can sort the monsters by strength, and on every step, we either kill the leftmost monster by hand or press the button and kill the left half of the current monsters.

Given that algorithm, we can find the answer to the problem by doing dynamic programming.

Let $dp(i)$ be the minimum total cost needed to kill all the monsters from 1 to i. If we already killed all the monsters from 1 to i, we are left with $n - i$ monsters. We can kill the monster $i + 1$ "by hand" with cost a_{i+1}, or we can kill the monsters from $i + 1$ to $i + \lfloor \frac{n-i}{2} \rfloor$ by pressing the button.

Then, the transitions are:

$$dp(i+1) = min(dp(i+1), dp(i) + a_{i+1})$$
$$dp\left(i + \lfloor \frac{n-i}{2} \rfloor\right) = min\left(dp\left(i + \lfloor \frac{n-i}{2} \rfloor\right), dp(i) + K\right)$$

Time and Space Complexity: $\mathcal{O}(n)$

H Hurrying Back Home

Authors: Marcelo J. Fornet Fornés - Aurora Labs and Alberto González Rosales - Volvo Cars
Category: Ad Hoc 1
Solved by 68 teams

First, follow the given instructions to determine the final coordinates (x_e, y_e) where WALL-E ends up at. Then, run a 0-1 BFS to obtain the minimum number of moves required to return to the origin $(0, 0)$... Just kidding! There is a "simpler" way to do the last step: the *shortest* path to the origin is to first reach either the X or Y axis; and once there, make some number of rotations and finally head back to the origin. Let's do some casework analysis:

- If both x_e and y_e are zero, we already reached the origin, so the answer is 0.
- If one of x_e or y_e is zero, WALL-E is already at one of the axis. So, we simply need to turn around (if necessary) and then move to the origin:
  - WALL-E is already facing toward the origin: there is no need to turn, so the answer is 1 (just move to the origin).

  - WALL-E is facing in the opposite direction (ie, away) from the origin: WALL-E needs two rotations and then move to origin. Total number of moves is 3.
  - Otherwise, WALL-E is facing "sideway" and needs one single rotation before moving to the origin. Total numbers of moves is 2.

- If neither x_e nor y_e is zero, let's first head to one of the axis.
  - If WALL-E is facing either of the axis, immediately move to the axis he is facing, and then turn and move to the origin. This requires 3 moves.
  - Otherwise, after a single turn, WALL-E will be facing one of the axis and we will be in the same situation as above. So, we need 4 moves in total.

Final complexity is $\mathcal{O}(n)$ due to the simulation to determine the coordinates (x_e, y_e).

I Inspecting Trees

Authors: Rubén Alcolea Núñez - UCI and
Ernesto Teruel Velazco - UCI
Category: Data Structures 2
Solved by 18 teams

Solution 1:
As a tree is a graph, the *heavy-light* decomposition solution for problem "Queries on Graph" also works here. For details, read the editorial for that task.

The complexity of the solution is $\mathcal{O}(\sqrt{n})$ per operation.

Solution 2:
Root the tree at any vertex v and run a Breadth-First Search (BFS) from the root, and appending all vertices to an array in the order they were seen in the BFS. This way, the children of a vertex make a contiguous range on this array. Therefore, each increasing operation is equivalent to adding $+x$ to a range, and each query is equivalent to asking the value at a position on the array, which can be easily handled with standard data structures like segment trees or sqrt decomposition.

Solution 3:
Note: In this explanation, we will use the terms *update* and *query* for operations of type 1 an 2, respectively.

In a naive implementation, whenever we perform an $update(v, x)$, we iterate over all neighbors of a vertex v and increase the values at those neighbors. For a $query(u)$, we simply print the value stored at u.

```
update(v, x):
   for each neighbord u of v:
      value_at[u] += x

query(u):
   print( value_at[u] )
```

This costs us $\mathcal{O}(n)$ per *update* in the worst case and $\mathcal{O}(1)$ per *query*.

Let's explore another option: instead of modifying the values at the neighbors of a vertex v, we accumulate the incremented values in some other location and ask v's neighbors to collect these values whenever they need to answer a query.

```
update(v, x):
   shared_by[v] += x   # accumulate x into location shared by node v

query(u):
   sum = 0
   for each neighbor v of u:   # go thru all locations shared by a neighbor
      sum += shared_by[v]
   print( value_at[u] + sum )
```

The complexity of an *update* reduces to $\mathcal{O}(1)$, but *queries* are $\mathcal{O}(n)$ in the worst-case.

Let's combine both approaches: in some situations, we will modify the values at the neighbors and in others we ask the neighbors to collect the accumulated increments. The trick is to avoid "double-counting": cases where we may be adding the same increment more than once.

If we root the tree, each node's neighbors become its children and one of them becomes the parent (except the root, which has no parent). On an $update(v, x)$, we will increase the value at the parent of v, but will skip the modification of the values at the children of vertex v. Instead, we will simply accumulate the increment in the location that vertex v shares for its children. In a $query(u)$, we will take the value shared by the parent of u and sum it to the current value stored at vertex u.

```
update(v, x):
   if v is not the root:
      p = parent[v]
      value_at[p] += x
   shared_by[v] += x

query(u):
   if u is the root:
      print( value_at[u] )
   else:
      p = parent[u]
      print( value_at[u] + shared_by[p] )
```

The complexity of this solution is $\mathcal{O}(1)$ for each operation.

J Join the Game

Author: Daniel Enrique Cordovés Borroto - Harbour Space University
Category: Constructive 3
Solved by 7 teams

In the problem, we can note the following: if we have k ($1 \leq k \leq 3$) piles with the same sizes modulo 3, then after applying an operation on any of the piles (if it can be done), we will still have exactly k piles with the same sizes modulo 3 (I).

For example, if we have:

- $(A, B, C) = (2, 4, 5)$, modulo 3 would be $(2, 1, 2)$, and by applying an operation for example on pile B, we obtain $((2-1) \bmod 3, (4+2) \bmod 3, (5-1) \bmod 3) = (1, 0, 1)$, both tuples have exactly $k = 2$ piles with the same sizes modulo 3.
- $(A, B, C) = (1, 2, 3)$, modulo 3 would be $(1, 2, 0)$, and after applying an operation on pile A, we obtain $((1+2) \bmod 3, (2-1) \bmod 3, (3-1) \bmod 3) = (0, 1, 2)$ and both tuples have exactly $k = 1$ piles with the same sizes modulo 3.

Let's proof (I). We have the piles (A, B, C), without loss of generality we apply the operation on pile A and we have:

- $A + 2 \stackrel{?}{\equiv} B - 1 \pmod 3 \iff A \stackrel{?}{\equiv} B \pmod 3$
- $A + 2 \stackrel{?}{\equiv} C - 1 \pmod 3 \iff A \stackrel{?}{\equiv} C \pmod 3$
- $B - 1 \stackrel{?}{\equiv} C - 1 \pmod 3 \iff B \stackrel{?}{\equiv} C \pmod 3$

From here we see that effectively if $A \equiv B \pmod 3$ then after the operation it would still hold true and also, if $A \not\equiv B \pmod 3$ after the operation it would still hold as well. The same can be said for the other pairs of piles (A, C) and (B, C). Therefore this means that the number of piles with the same size modulo 3 does not change after an operation.

Then if the sizes of the three piles modulo 3 are different ($k = 1$) there is no solution.

Otherwise, if $k \geq 2$ a solution always exists. Following we give such a solution:

- If there are two piles with the same size we are done, because we apply the operation on the third pile until the other two piles are 0.

  Now, suppose without loss of generality that $A \equiv B \pmod 3$ and $A \leq B$. The goal is to make $A = B$.

- If $C > 0$, we perform the operation on A and obtain $(A+2, B-1, C-1)$.
- If $C = 0$, we perform the operation on C and obtain $(A-1, B-1, 2)$, so now we have $C = 2$, and we go back to the previous step.

  In both cases, it still holds that $A \equiv B \pmod 3$ and $A \leq B$ after the operation.

We repeat the procedure until $A = B = 0$. This always terminates because as long as $C > 0$, we perform the operation on A, and then we have $C = 0$, in that case, we perform the operation on C and then on A successively, until $A = B$, and finally, we apply the previously explained first point until $A = B = 0$.

The number of operations performed by this solution is less than or equal to $\max(A, B, C)$, this is because we are always decreasing the size of the same pile (which would be B in the explanation) by one in each operation.

Complexity: $O(\max(A, B, C))$.

K Keyboard Patterns

Authors: Ernesto Teruel Velazco - UCI,
Rubén Alcolea Núñez - UCI and
Carlos Joa Fong - Orange Dominicana
Category: Combinations 3
Solved by 3 teams

Solution 1:

Since the number of keys, 9, is small, we can count the number of valid patterns by brute force: generate all possible permutations of integers 1 to 9 and check if the generated permutation represents a valid sequence of moves.

Languages such as C++ and Python already include modules that can iteratively generate all possible permutations. Since there are $N!$ possible permutations (of a length-N sequence) and it takes $\mathcal{O}(N)$ time to check whether a sequence is valid, the total time is $\mathcal{O}(N * N!)$ per test case. This may be too slow to process the $T = 1000$ test cases (even if we precompute the answers for each of the 2^9 possible inputs).

Instead, we can recursively generate the permutations. Before adding a key k to the end of the sequence, we verify whether the "slide" from the last key in the sequence to key k is valid. This allows us to prune the search from considering all permutations with a prefix that is equal to the sequence + k.

The complexity of this solution is still $\mathcal{O}(N*N!)$, but it is fast enough to pass all test cases due to effectiveness of the pruning when there are many defective keys.

How do we check the validity of a move? We can build a graph where each key is a node in the graph and the edges represent valid moves from one

non-defective key to another. Now, there is the tricky part that restricts slides between two keys to always be in a straight line. For example, to go from key 1 to key 3, we need to pass through key 2. To deal with this rule, we store these special edges associating them with the "middle" key they must go through. For example: $special[(1,3)] = 2$. When validating a move, we check whether the edge is special and, if so, check whether the associated middle key is already used. If this middle key is used, we can use the edge.

Solution 2:

A more efficient approach is to apply dynamic programming, where we define $dp[m, u]$ as the number of valid sequences of keys used in the bitmask m and the last key is u.

- Transition is $dp[m, u] = \sum_{v=0}^{8} dp[m - 2^u, v]$ where (v, u) is a valid move.
- Base case is $dp[2^v, v] = 1$ for all possible non-defective keys v (from 0 to 8).
- Answer is $\sum_{m=1}^{2^9-1} \sum_{u=0}^{8} dp[m, u]$

Complexity is $\mathcal{O}(N^2 * 2^N)$.

L Luminaries

Author: Mariano Jason Rodriguez Cisnero - Universidad de Oriente
Category: Arithmetic-Algebra 3
Solved by 1 team

We will first assume that there are no point-locking operations.

For the translation, rotation, and scaling operations, we will use matrices of linear applications. Let $M_{2\times2}$ be the matrix, and $p = (x, y)$ the point, the new point $\hat{p}$ is obtained, by multiplying the two:

$$M_{2\times2} \cdot p^T = M_{2\times2} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \hat{p}$$

The translation operation is not linear, but a trick can be employed to make it "linear". The trick to include the translation is to convert the above matrices from 2D to 3D as follows:

$$M_{3\times3} = \begin{pmatrix} M_{2\times2} & & 1 \\ & & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

and the points from (x, y) to $(x, y, 1)$.

The application of t successive operations to a point can be represented as follows:

$$M^t_{3\times3} \cdot M^{t-1}_{3\times3} \cdot M^{t-2}_{3\times3} ... \cdot M^0_{3\times3} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \hat{p}^t$$

Applying any of the above operations can be represented as the change of basis of a vector system. For this, a matrix is created with the vectors of the new basis as columns. So the matrices for scaling in 2D would be: $M_{3\times3}$ and the points from (x, y) to $(x, y, 1)$.

The application of t successive operations to a point can be represented as follows:

$$M_{3\times3} = \begin{pmatrix} s_x & 0 & 1 \\ 0 & s_y & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad with \quad s_{x,y} = \begin{cases} m_{x,y} & \text{if it is a multiplication} \\ \frac{1}{d_{x,y}} & \text{if it is a division} \end{cases}$$

on the other hand, since the rotation operation is simplified only to 90° angles, the four matrices for the four rotations can be created just by using the $\pm x$ and $\pm y$ axes as vectors of the new basis, as follows:

$$M^{0^\circ}_{3\times3} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \quad M^{90^\circ}_{3\times3} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$M^{180^\circ}_{3\times3} = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \quad M^{270^\circ}_{3\times3} = \begin{pmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

While the translation matrix would be:

$$M_{3\times3} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Locking a point

To include the locking operation, the only thing to do is to store each matrix in the order of appearance in a list, and to answer the value of a point is just to multiply the matrices in which the point was active. Obviously, it is quite time-consuming to go through each matrix and multiply it, but one can make use of the linear property of matrices, and the fact that each is a basis and has an inverse to generate a "cumulative" list of the matrices, and if an interval of consecutive operations is required, it can be obtained in the following way:

Let A_t be the value of the element t in the "cumulative" list of the matrices, with

$$A_t = M^t_{3\times 3} \cdot A_{t-1} \qquad \text{with } M^0_{3\times 3} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Then to know the value of a point by applying only consecutive operations between a and b where the point is active, with $a \leq b$:

$$A_{[a,b]} = A_b \cdot (A_{a-1})^-$$

with $(A_{a-1})^-$ the inverse of A_{a-1}. Then, the value of p^* after applying all operations between a and b:

$$A_{[a,b]} \cdot p^* = \hat{p}^*$$

Due to the presence of the inverse, it is necessary to implement the data type fraction and the operations of multiplication, addition, subtraction, and division for it. Although it will only be necessary to print the numerators since the constraints in the problem statement ensure that the output will be an integer.

Final solution

For the solution in $O(n)$, just accumulate each operation in a "cumulative" list.

- When a point is locked at time b, and changes from locked to active for the last time at time a, it is only necessary to calculate $A_{[a,b]}$ and multiply the point with it, saving its new value.
- If you are asked the value of the point at time b and it is locked, just print its value. If it is active, calculate $A_{[a,b]}$ with a the time when it went from locked to active, multiply it by the point, and print the result.

The inverse can be calculated with Cramer:
https://libraryguides.centennialcollege.ca/c.php?g=717345&p=5124944

M Queries on Graphs

Authors: Carlos Joa Fong - Orange Dominicana and
Rubén Alcolea Núñez - UCI
Category: Data Structures 3
Solved by 6 teams

Let's call vertex *light* if its degree is less than $\sqrt{n}$, else we will call it *heavy*.

- **Query 1:** If v is *light*, it's easy. If v is *heavy*, let's store x into *lazy[v]* –the value we have to add to all the neighbors of v.
- **Query 2:** Let's take value of vertex v and add *lazy[i]* for every *heavy* neighbor i of v.

For this to work efficiently, if u is a *heavy* vertex, we will prune all *light* neighbors of u from its list of neighbors (ie, the adjacency list of node u), since there may be many of them and they are useless: a query of type 1 applied to one this neighbors already updates the value at node u, and in a query of 2 on one this neighbors, these neighbors will consider the lazy value stored at node u. The remaining neighbors of these heavy neighbors are all *heavy*. But, we cannot have more than $\mathcal{O}(\sqrt{m})$ of these heavy neighbors. Proof of this is left as an exercise to the reader.

The total complexity of the solution is $\mathcal{O}(q\sqrt{n})$.