Data Structure and Algorithm, Spring 2021

Homework 2

Due: 13:00:00, Tuesday, May 18, 2021

TA E-mail: dsa_ta@csie.ntu.edu.tw

Rules and Instructions

- Any form of cheating, lying, or plagiarism will not be tolerated. Students can get zero scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.
- In Homework 2, the problem set contains 6 problems and is divided into two parts, the non-programming part (Problems 1, 2, 3) and the programming part (Problems 4, 5, 6).
- For problems in the non-programming part, you should combine your solutions in ONE PDF file. Your file should generally be legible with a white/light background—using white/light texts on a dark/black background is prohibited. Your solution must be as simple as possible. At the TAs' discretion, solutions which are too complicated will be regarded as incorrect. Moreover, if you would like to use any theorem which is not mentioned in the classes, please include its proof in your solution.
- The PDF file for the non-programming part should be submitted to Gradescope as instructed, and you should use Gradescope to tag the pages that correspond to each subproblem to facilitate the TAs' grading. Failure to tagging the correct pages of the subproblem can cause losing part or all of the scores on the subproblem.
- For the programming part, you should have visited the *DSA Judge* (https://dsa-2021.csie.org) and familiarized yourself with how to submit your code via the judge system in Homework 0.1126.
- For problems in the programming part, you should write your code in C programming language, and then submit the code via the judge system. You can submit up to 5 times per day for each problem. The judge system will compile your code with

```
gcc main.c -static -O2 -std=c11
```

- Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from. For *each non-programming problem*, you have to specify the references (the Internet URL you consulted with or the people you

discussed with) on the first page of your solution for that problem; for *each programming problem*, you have to specify the references on the first lines (comments) of your code (`main.c`) for that problem.

- Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.
- The score of the part that is submitted after the deadline will get some penalties according to the following rule:

$$\text{LateScore} = \max\left(0, \frac{86400 \cdot 5 - \text{DelayTime (sec.)}}{86400 \cdot 5}\right) \times \text{OriginalScore}$$

- If you have questions about HW2, please go to the course forum and discuss (*strongly preferred*, which will provide everyone a better learning experience). If you really need an email answer, please follow the rules outlined below to get a fast response:
  - The subject should contain two tags, `"[HW2]"` and `"[Px]"`, specifying the problem where you have questions. For example, `"[HW2][P1] Can swap in subproblem 7 be done in constant time?"`. Adding these tags allows the TAs to track the status of each email and to provide faster responses to you.
  - If you want to provide your code segments to us as part of your question, please upload it to Gist or similar platforms and provide the link. Screenshots or code segments directly included in the email is discouraged and may not be reviewed.

Non-Programming Part

Problem 1 - Sort (80 pts + 20 pts)

Suppose there are n pancakes $P_1, \ldots, P_n$ with unknown tastiness $t_1, ..., t_n$ respectively. Arvin, as a pancake lover, wants to "sort" the pancakes according to their tastiness. However, he is prohibited to try the pancakes on his own. Luckily, he can pray to the Pancake-God for help. In each round of praying, Arvin specifies three different pancakes, and the Pancake-God indicates the pancake with the **median** tastiness. With the help of the Pancake-God, Arvin wants to find the order of the n pancakes. Notice that by knowing only median-of-three's, it is impossible to distinguish the order of any list from its reverse, since the median query results in the same answer for any three pancakes in both cases. Therefore, we call a list of pancakes to be "sorted" as long as their tastiness is totally increasing or totally decreasing.

Formally, suppose there is a query model that involves Pancake-God-Oracle($\mathbf{P}, i, j, k$), a black-box function that compares the pancakes P_i, P_j, P_k and returns the pancake with the median tastiness. In the following subproblems, we start by checking the query complexity of the algorithm, i.e., the number of Pancake-God-Oracle queries required for the algorithm. That is, we assume that all the implementation details, such as control, data movement, etc. does *not* take any efforts. We just focus on whether we have the necessary information from querying the Pancake-God to complete the task.

1. (10 pts) Given n pancakes, design an algorithm with $O(n)$ query complexity that returns two "boundary" pancakes, one of them being of the maximum tastiness, and the other of the minimum tastiness.

2. (20 pts) Using the $O(n)$ query complexity algorithm above to get two "boundary" pancakes on hand (if you cannot solve the previous problem, you can also assume to have the algorithm above as a black box), design an divide-and-conquer algorithm to "sort" the n pancakes with query complexity $O(n \log n)$. *(Hint: There are more than one approaches. Concepts of Merge Sort or Quick Sort may help.)*

3. (10 pts) Arvin wants to avoid unnecessary calls to Pancake-God-Oracle, otherwise the Pancake-God might get angry. Suppose there is a list L of m pancakes that is already "sorted." Given another pancake, design an algorithm with $O(\log n)$ query complexity to insert this pancake into L such that the new list of pancakes is still "sorted."

4. (5 pts) Using the $O(\log n)$ query complexity algorithm above to "insert" the pancake to the right place (if you cannot solve the previous problem, you can also assume to have

the algorithm above as a black box), design an algorithm to sort n pancakes with query complexity $O(n \log n)$ without first locating the boundary pancakes. *(Hint: Concepts of Insertion Sort may help.)*

5. (Bonus 20 pts) Argue that it is impossible to find an algorithm with $o(n \log n)$ query complexity to "sort" all pancakes. Therefore, the result above is asymptotically optimal! *(Hint: Please check the textbook for the formal definition of the little-oh notation, if needed. You will get all bonus credits only if your proof is fully rigorous, while the TAs can choose to give partial credits.)*

After a few rounds of answering Arvin's calls, the Pancake-God is tired. The Pancake-God then asks Arvin to go find the Pancake-Elf, who would directly not just "sort" all pancakes, but also order them from the smallest to the largest tastiness and return the sorted result to Arvin. The sorting algorithm that the Pancake-Elf uses is as follows.

Algorithm 1: The fancy algorithm of the Pancake-Elf

```
Function ELF-SORT(pancake array P, l, r): // with r ≥ l
    if r − l ≥ 2 then
        Δ = floor((r−l+1)/3)
        ELF-SORT(P, l, r − Δ)
        ELF-SORT(P, l + Δ, r)
        ELF-SORT(P, l, r − Δ)
    end
    else if t_r > t_l then
        swap P_l with P_r in P
    end
```

6. (15 pts) Prove that the ELF-SORT algorithm can correctly sort a pancake array $\mathbf{P}$ in ascending order by calling ELF-SORT($\mathbf{P}$,1,n). *(Hint: You may need to prove by induction on n.)*

7. (5 pts) Let $T(n)$ denote the worst-case running time of ELF-SORT$(\mathbf{P}, l, r)$, where $n = r - l + 1$ is a positive number denoting the problem size. Explain the recurrence relation of $T(n)$, which should be expressed as a form similar to $T(n) = aT(bn) + \Theta(f(n))$.

8. (15 pts) Prove by definition that $T(n) = O(n^3)$. That is, you need to prove that there exists positive constants n_0, c such that $T(n) \leq cn^3$ for all $n \geq n_0$. (In fact, $T(n) = \Theta(n^{\log_{1.5} 3})$, which implies its worst-case running time complexity is far worse than the usual sorting algorithms that we have discussed!)

Problem 2 - Tree (65 pts)

HowHow is a very diligent student. Recently, he started to explore the stock market by himself. He learned that the Brownian Motion model for financial markets can be very useful. He wanted to be an expert on the model with the hope of being the next Warren Buffett. One use of the Brownian Motion model is in estimating stock prices. Those who are interested in the details can go check the popular Black-Scholes model. But we will keep the story short here. Assume that at each of the time step t_k, where $k \in \{1, 2, \ldots, n\}$ and $t_1 < t_2 < \ldots < t_n$, a piece of market information data X_{t_k} is recorded. Based on the models and the so-called efficient market hypothesis that HowHow will rely on, at any time stamp t_k, HowHow only needs the information data $X_{t_{k-1}}$ recorded at the previous time stamp t_{k-1} to predict the stock price at t_k.

Given that the market that HowHow studied has a long history, the data set $\{(t_k, X_{t_k})\}_{k=1}^{N}$ is super big. HowHow thus decided to organize the data set with a binary search tree T using t_k as the key and X_{t_k} as the data of every node. For the binary search tree T, we will assume that every sub-tree of T is a binary search tree with its left (right) sub-sub-tree containing data earlier (later) than the sub-tree root data in time.

1. (10 pts) After constructing such a giant tree, HowHow was exhausted and could not think of how to get the node that he needed. Please help him by designing an algorithm that takes a node with key t_k as the input, and returns the "previous" node, which has key t_{k-1}. You can assume each node of the tree to contain three links: `left`, `right`, and `parent`. The algorithm should return NIL when taking the earliest node t_1 as the input.

2. (15 pts) Prove the correctness of the algorithm that you have designed.

After getting the node needed, HowHow applied his model, which predicted that the price will go up by 15%. HowHow then decided to throw all his money in, and guess what, a black swan came and HowHow lost all his money. He learned that the market is not as easy as he imagined.

Suddenly, VP Junior appeared and told HowHow that he is a Fuerdai—son of the VP of Crypto Arsenal, which is a crypto currency behemoth. VP Junior told HowHow his little secret of being wealthy in modern ages—mining crypto currency. If HowHow can help VP Junior solve some algorithm problems that have been bothering VP Junior recently, he can give HowHow a new RTX 3090 to help HowHow make a fresh start.

3. (10 pts) VP Junior asked HowHow to warm up with a concrete task. Given two sequences of traversal from a binary tree, reconstruct the original binary tree.

```
inorder[] = {31, 43, 74, 57, 86, 15, 21, 60, 38}
preorder[] = {86, 43, 31, 57, 74, 21, 15, 60, 38}
```

4. (15 pts) Then, a more difficult task came from VP Junior. If all nodes are of distinct key values, are there two binary trees that result in the same (`inorder`, `preorder`) pair? Prove or disprove this statement.

After years of blood, sweat, and tears, HowHow finally solved the second task. The brand new life is right here under his nose. VP Junior asked HowHow to construct any binary tree (among many of them, in case there are more than one) from a pair of very long (`inorder`, `preorder`) sequences.

5. (15 pts) Design an algorithm that returns a binary tree from the (`inorder`, `preorder`) traversal sequences. Your algorithm should be time-wise as efficient as possible.

Problem 3 - Heap (55 pts)

First, consider a normal binary min-heap `h`, which is implemented with a complete binary tree, that supports following operations. Let $|h|$ be the size of heap `h`.

- `h.insert(x)`: inserts an element `x` to `h` in $O(\lg |h|)$-time
- `h.extractMin()`: remove the minimum element from heap `h` in $O(\lg |h|)$-time
- `h.modify(x, v)`: modify the value of an element `x` to `v` in $O(\lg |h|)$-time
- `h.delete(x)`: delete an element `x` in `h` in $O(\lg |h|)$-time

1. (15 pts) We basically taught how to do `h.insert(x)` and `h.extractMin()` in class. Now, please design the corresponding algorithm for `h.modify(x, v)`, and prove why it meets the time complexity requirement. (Technically, `h.delete(x)` can be implemented by generalizing `h.extractMin()`, or by combining `h.modify(x,` $-\infty$`)` with `h.extractMin()`).

After finishing the task above, you are now a heap master! Now, we will work on a more challenging task. Consider an initially empty 2D array $A_{N\times M}$, with indices $\{(0,0),\ldots,(N-1,M-1)\}$. There is a data structure `D` that supports following operations on A. You can assume all the operations to be valid: only adding to empty locations and extracting/deleting from non-empty ones.

- `D.add(i, j, v)`: add an element `v` to $A_{i,j}$ in $O(\lg(NM))$-time.
- `D.extractMinRow(i)`: remove the minimum element in i^{th} row of A in $O(\lg(NM))$-time.
- `D.extractMinCol(j)`: remove the minimum element in j^{th} column of A in $O(\lg(NM))$-time.
- `D.delete(i, j)`: delete the element at $A_{i,j}$ in $O(\lg(NM))$-time.

2. (10 pts) The following operations are called sequentially. The `ShowState()` command shows the contents of $A_{4\times 4}$. Please illustrate the output of each `ShowState()` command.

   (a) `D.add(2, 3, 1); D.add(3, 3, 2); D.add(3, 0, 4); ShowState();`

   (b) `D.extractMinRow(3) ; ShowState();`

   (c) `D.add(1, 3, 3); ShowState();`

   (d) `D.delete(2, 3); ShowState();`

   (e) `D.extractMinCol(3 ); ShowState();`

3. (10 pts) Briefly explain the design of your data structure `D`. Your design should be reasonable and detailed enough to get full points. *(Hint: Did we say Heap in this problem?)*

4. (20 pts) Describe the design of the four operations and explain why they meet the time complexity requirements.

Programming Part

Problem 4 - Fake Binary Search Tree (100 pts)

Time Limit	:	1 s
Memory Limit	:	1024 MB

Problem Description

Giver, just like you, is a diligent student of the DSA class. He just learned a new data structure: *Binary Search Tree*. Let us recap Binary Search Tree (BST) a bit. A BST is a rooted binary tree that can help you find elements fast. Formally, if a binary tree satisfies the following constraints, then it is a BST.

- Every node of the tree has its own key.
- The key in each node is greater than or equal to any keys stored in its left sub-tree.
- The key in each node is less than or equal to any keys stored in its right sub-tree.

Then, the following algorithm, "binary search", checks whether an element x is in the BST.

Algorithm 2: Lookup of a key in a BST.

Input: A BST rooted at *tree_node* and a *key* to look for

Output: **True** if the *key* is found, **False** otherwise

Function SEARCH*(tree_node, key):*

```
if tree_node == NIL then
    return False
end
if tree_node.key == key then
    return True
end
if tree_node.key > key then
    return SEARCH(tree_node.leftchild, key)
end
else
    return SEARCH(tree_node.rightchild, key)
end
```

After studying this simple algorithm, Giver is super excited. He tries to apply it on general binary trees to see understand more about the algorithm. To his dismay, the algorithm does

not work for general binary trees. Giver is curious about the number of keys that are in the general binary tree which can be found by applying the algorithm. Please help Giver locate those keys and compute the number!

Input

The first line of the input contains only one integer N ($1 \leq N \leq 10^6$), indicating the number of nodes in the given binary tree. The nodes are indexed from 1 to N. In the following N lines, the ith line contains the content of node i, which is represented by three integers w, l, r ($1 \leq w \leq 10^9, l, r \in \{-1\} \cup \{1, 2, \ldots, N\}$), representing the key, the id of left child, and the id of right child. An index of -1 is used to indicate NIL—that is, when there is no child. The keys in each node are unique. It is guaranteed that the input forms a valid binary tree rooted at 1.

Output

Print an integer representing the number of keys (within the binary tree) that can be found by the algorithm (returns **True**).

Subtask 1 (30 pts)

- $N \leq 1000$.

Subtask 2 (70 pts)

- No other constraints.

Sample Input 1

```
5
1 2 4
2 -1 3
5 -1 -1
6 -1 5
7 -1 -1
```

Sample Output 1

```
3
```

Sample Input 2

```
6
2 2 -1
1 -1 3
5 4 6
6 5 -1
7 -1 -1
9 -1 -1
```

Sample Output 2

```
2
```

Sample Input 3

```
7
10 2 7
3 3 4
1 -1 -1
7 5 -1
9 -1 6
5 -1 -1
4 -1 -1
```

Sample Output 3

```
4
```

Problem 5 - Intersecting Triangles (100 pts)

Time Limit	:	3 s
Memory Limit	:	1024 MB

Problem Description

There are N ants who live on a 2D plan. The i-th ant rests on a point p_i in the line of $y = 1$ during the night. Then, at some starting time, the ant first walks by a straight line to another point q_i in the line of $y = 0$ to start working. After going to q_i, the ant starts working by going straight from point q_i to r_i, both in the line of $y = 0$. Then, at some ending time, the ant walks back to p_i from r_i by a straight line. That is, the path of each ant follows a triangle (p_i, q_i, r_i). Each different ant can have a different walking speed, a different starting time, and/or a different end time.

The Queen Ant recently received some complaints from the ants, saying that some ants bump into each other in their daily triangular path. To understand how serious the situation is, the Queen Ant asks the Scientist Ant to list, from the paths of all ants, which ants may bump into each other—at the vertex of the triangular path, at one other point of the path, on a segment of the path. Please help the scientist determine the number of **pairs** of ants that may bump into each other on their triangular paths.

Input

The input can be read through the provided header file `generator.h`. Please first call `generator.init()` to initialize the generator, then calling `generator.getT()` will return the number of test cases T. For each test case, calling `generator.getData(&N, &P, &Q, &R)` will let you get the number of triangles N, and their vertices. Note that the type of `N` should be `int`, and the type of `P, Q, R` should be `int*`. After calling, `P[0..N-1]` would store $\{p_1, p_2, \ldots, p_N\}$, and `Q[0..N-1]`, `R[0..N-1]` will store the other points, respectively. Below is an example usage, the program will run and get the data correctly; however, you will get WA if you submit it directly:

```
#include <stdio.h>
#include "generator.h"
int main() {
    generator.init();
    int t = generator.getT();
    while (t--) {
        int n, *p, *q, *r;
```

```
        generator.getData(&n, &p, &q, &r);
        /* do something
        int ans = 0;
        for (int i = 0; i < n; i++) ans += p[i] * q[i] * r[i];
        printf("%d\n", ans);
        */
    }
}
```

Note that you should not read anything from standard input in your program; otherwise, the behavior is undefined.
You can download the generator header "generator.h" here.

Output

For each test case, print the number of pairs of intersecting triangles.

Subtasks

In all subtasks, $-2^{20} \leq p_i, q_i, r_i \leq 2^{20} - 1$ holds for each $1 \leq i \leq N$.

Subtask 1 (20 pts)

- $1 \leq N \leq 3000$, $1 \leq T \leq 10$

Subtask 2 (20 pts)

- $1 \leq N \leq 10^5$, $1 \leq T \leq 10$
- all p_i are distinct.
- all q_i and r_i are distinct.

Subtask 3 (10 pts)

- $1 \leq N \leq 10^5$, $1 \leq T \leq 10$
- all p_i are distinct.

Subtask 4 (10 pts)

- $1 \leq N \leq 10^5$, $1 \leq T \leq 10$

Subtask 5 (40 pts)

- $1 \leq N \leq 3 \times 10^6$, $T = 1$

Sample Cases

Sample Input 1

```
1 1 5 16
538724387836423741 325591348600219474 187178394057222755 353408734984306357
```

Actual Sample Input 1

```
T = 1
N = 5
P = {12, 7, -10, 12, 9}
Q = {11, 5, 5, -16, 5}
R = {-2, -13, -8, 8, 10}
```

Sample Output 1

```
10
```

Sample Input 2

```
2 1 5 16
51414933668525662 550301789874357166 622479167726386043 650347521267739593
```

Actual Sample Input 2

```
T = 1
N = 5
P = {2, -3, 6, -16, 0}
Q = {-7, -14, -5, 1, 6}
R = {-12, 5, 9, 13, -8}
```

Sample Output 2

```
9
```

Sample Input 3

```
3 1 5 16
203739077647024131 805985539835675567 140205801930598907 908190957489194415
```

Actual Sample Input 3

```
T = 1
N = 5
P = {2, 14, -16, -7, 13}
Q = {7, 14, 0, 7, -5}
R = {11, -12, 8, -12, -1}
```

Sample Output 3

```
10
```

Sample Input 4

```
1 2 10 16
869534322540300934 837268419296844257 456729939812480767 541019751318820673
```

Actual Sample Input 4

```
T = 2
N = 10
P = {-5, 15, 15, -8, -11, -5, -13, -14, 15, 10}
Q = {-7, -12, -4, -13, 5, 1, 14, -16, -16, -14}
R = {2, -13, 1, 9, 9, 1, 1, 12, -15, 6}
N = 10
P = {7, -4, -7, 12, 9, -13, -11, 5, 3, 8}
Q = {8, 5, 3, -4, -10, -10, 2, 2, -7, -9}
R = {-11, -1, -1, -12, 4, 13, -16, -3, -7, 4}
```

Sample Output 4

```
45
44
```

Problem 6 - Package Arrangement

Time Limit	:	3 s
Memory Limit	:	1024 MB

Problem Description

Ling is a worker in a factory. Each day, a sequence of packages, each of a different height, are "pushed" to several production lines. Each production line can be viewed as a queue where the packages that are "pushed" into the line earlier are in its front.

Ling's job, on the other hand, is to "pop" a package from some production line. The popped package will then be moved to a target line. The control panel that Ling uses has three buttons for each production line: popping the (1) first / (2) last / (3) highest package from the production line, and moving it to the target line.

Occasionally, some production line may be closed for maintenance. When it happens, all the packages on the closed production line will be dequeued and pushed to another running production line in order. We call this operation to be "merging" the closed production line and the running production line. In other words, the sequence of packages in the closed production line is concatenated to the end of the running production line after the merge operation. The closed production line then becomes empty and will not be used again.

The pushing and merging operations are factory-controlled, and Ling cannot do anything about them. What Ling can do is to decide whether to execute the popping action of the first / last / highest package from one of the production lines after each operation. The decision of the popping executions from Ling forms a particular height sequence of the target line. After Ling's executions every day, he writes down the height sequence of the target line on the daily log of the factory.

Nevertheless, Ling is not the most careful person in the world, and hence sometimes makes mistakes in his writing. An obvious mistake is that the line in the record is not possible from any combination of the popping actions. Ling hopes to capture this kind of mistake before sending the daily log to the factory. Can you help him do that?

Input

The first line contains an integer T, which indicates the number of test cases. The following lines contain T test cases and each test case is formatted as follows:

- The first line of the test case contains three integers N, O, and L. N indicates the number of packages. O indicates the total number of "push" and "merge" operations. L indicates the number of production lines.

- The next O lines are the "push" and "merge" operations, one in each line. The line with `push` comes with two numbers, a package height $1 \leq h \leq N$ and a production line number $0 \leq \ell < L$ to push the package to; the line with `merge` comes with two numbers, a broken production line number $0 \leq \ell_b < L$ and a different running production line number $0 \leq \ell_r < L$.
- The last line contains the record from Ling, which is a permutation of $\{1, 2, \ldots, N\}$ indicating the heights of the packages on the target line.

Output

If it is possible to arrange the packages to the given order by any combinations of the popping actions, please print `possible` in a single line; otherwise please print `impossible` in a single line.

Constraints

- $1 \leq T \leq 10$
- $1 \leq N, L \leq 10^5$
- $N \leq O < N + L$

Subtask 1 (15 pts)

- $1 \leq N, L \leq 1000$

Subtask 2 (25 pts)

- $L = 1$, which also means no "merge" operations

Subtask 3 (10 pts)

- No "merge" operations

Subtask 4 (50 pts)

- No other constraints.

Sample Input 1

```
2
5 5 1
push 2 0
push 1 0
push 3 0
push 4 0
push 5 0
2 1 5 3 4
5 5 1
push 1 0
push 4 0
push 2 0
push 5 0
push 3 0
5 1 3 2 4
```

Sample Output 1

```
possible
possible
```

Sample Input 2

```
2
10 13 5
push 10 3
merge 3 4
push 2 1
push 7 4
push 8 4
push 9 4
push 5 4
merge 1 4
push 4 0
merge 4 2
push 1 2
```

```
push 6 2
push 3 2
10 4 8 7 9 1 3 5 6 2
10 13 5
push 7 1
push 5 1
push 1 1
merge 1 4
push 9 4
push 4 0
push 2 4
push 6 0
push 8 4
merge 4 3
push 3 0
merge 2 3
push 10 3
4 6 9 7 8 3 2 10 5 1
```

Sample Output 2

```
impossible
possible
```

Sample Input 3

```
2
10 19 10
push 3 3
push 1 7
push 10 3
merge 6 2
push 4 8
merge 1 3
push 8 3
merge 9 7
merge 7 3
```

```
push 7 3
merge 8 3
merge 5 3
push 6 3
push 9 3
merge 4 0
push 2 2
merge 3 0
push 5 0
merge 0 2
10 3 6 8 2 7 9 1 5 4
10 19 10
push 7 6
merge 3 2
merge 6 2
push 6 4
merge 7 8
merge 4 5
push 9 5
merge 0 1
push 8 1
push 5 5
push 2 8
push 3 2
merge 8 5
push 1 2
merge 9 1
push 10 2
merge 2 1
push 4 1
merge 1 5
6 9 7 5 3 1 2 10 4 8
```

Sample Output 3

```
impossible
possible
```