Homework #1

Due Time: 2019/10/17 (Thu.) 14:20
Contact TAs: `ada-ta@csie.ntu.edu.tw`

Instructions and Announcements

- There are **four programming problems** and **three hand-written problems**.
- **Programming.** The judge system is located at `https://ada-judge.csie.ntu.edu.tw`. Please login and submit your code for the programming problems (i.e., those containing "Programming" in the problem title) by the deadline. NO LATE SUBMISSION IS ALLOWED.
- **Hand-written.** For other problems (also known as the "hand-written problems"), you **MUST** turn in **a printed/written version** of your answers to the instructor at the beginning of the class. You can also upload your homework to the NTU COOL system as a backup; however, it will be marked **only when** you have turned in the printed/written answer but it is lost during the grading process.
NO LATE SUBMISSION IS ALLOWED.
- **Collaboration policy.** Discussions with others are strongly encouraged. However, you should write down your solutions **in your own words**. In addition, for **each and every** problem you have to specify the references (e.g., the Internet URL you consulted with or the people you discussed with) on the first page or comment in code of your solution to that problem. You may get zero point due to the lack of references.

Problem 1 - ADA Meetup (Programming) (10 points)

Problem Description

N people are lining up to enter a meetup held by the Aged Drivers Association, also known as the ADA. Each person is characterized by two parameters: a preference index p_i and a friendliness factor f_i. The former indicates what a person likes, and the closer two people's indices are, the more similar their tastes are, and vice versa. The friendliness factor, on the other hand, indicates how friendly a person is. A person with a higher friendliness factor is more tolerant to people with different tastes. As we all know, old drivers tend to argue about which kinds of cars are better for driving. Therefore, when the i-th person enters the room, the person will only greet those whose tastes the person considers tolerable.

More formally, person i will greet the person j if and only if $i > j$ and $|p_j - p_i| \leq f_i$. WillyPillow, being a weeb with no friends to greet, decides to count the number of greetings that takes place.

Input

The first line of the input file contains an integer indicating N.

The second line contains N integers separated by spaces, with the i-th integer indicating p_i, the preference index of the i-th person.

The third line contains N integers separated by spaces, with the i-th integer indicating f_i, the friendliness level of the i-th person.

Test Group 0 (0 %)

- Sample Input

Test Group 1 (10 %)

- $1 \leq N \leq 10^6$
- $0 \leq p_i, f_i \leq 10^9$
- $|f_i - 5 \times 10^8| \geq 5 \times 10^8$

Test Group 2 (15 %)

- $1 \leq N \leq 5 \times 10^3$
- $0 \leq p_i, f_i \leq 10^9$

Test Group 3 (75 %)

- $1 \leq N \leq 10^6$
- $0 \leq p_i, f_i \leq 10^9$

Output

Please output an integer indicating the number of greetings that takes place.

Sample Input 1

```
5
18 0 14 8 12
9 8 14 4 11
```

Sample Output 1

```
5
```

Sample Input 2

```
10
17 17 10 10 0 1 6 6 17 13
12 16 18 12 16 6 6 16 15 16
```

Sample Output 2

```
35
```

Hint

Because the input files are large, please add

- `std::ios_base::sync_with_stdio(false);`
- `std::cin.tie(nullptr);`

to the beginning of the main function if you're using `std::cin`.

Problem 2 - Maximum Subarray Revisited (Programming) (10 points)

Problem Description

WillyPillow has recently learned about the famous *Maximum Subarray Problem* in the *Algorithm Design and Analysis* course. However, this problem had already been studied by many people for a long time, which makes it less challenging to solve for WillyPillow. Therefore, he quickly came up with the *dynamic* version of the maximum subarray problem, but was not able to solve it. Since WillyPillow has no friends to discuss with, you decide to help him with the modified task.

Formally speaking, you are given an integer array A of length N and Q updates. The i-th update is specified by two integers p_i, v_i, meaning that the p_i-th entry of A is updated to v_i. You should output the answer to the *Maximum Subarray Problem* on the modified array after each update.

Recall that the answer to the *Maximum Subarray Problem* is

$$\max_{1 \le l \le r \le N} \left(\sum_{i=l}^{r} A_i \right)$$

In particular, if the answer is negative, you should output 0 instead.

Input

The first line of the input contains two integers N and Q, representing the length of the array and the number of updates, respectively.

The second line contains N integers, representing the original array A.

Q lines follow, the i-th of which contains two integers p_i, v_i, representing an update of A_{p_i} into v_i.

- $1 \le N, Q \le 5 \cdot 10^5$
- $|A_i| \le 10^9$
- $1 \le p_i \le N$
- $|v_i| \le 10^9$

Test Group 0 (10 %)

- $N, Q \le 2000$

Test Group 1 (20 %)

- $N, Q \le 5000$

Test Group 2 (30 %)

- $Q = 0$

Test Group 3 (10 %)

- $N, Q \le 10^5$

Test Group 4 (30 %)

- No other constraints.

Output

Output $Q + 1$ lines. The first line should contain the answer before any updates. The rest Q lines should contain the answer after each update, following the chronological order.

Sample Input 1

```
10 10
-1 -5 -10 0 7 -1 4 -6 -3 -4
5 -4
3 10
5 -5
10 8
3 -2
8 1
10 7
1 -7
3 4
4 9
```

Sample Output 1

```
10
4
10
10
10
8
10
9
9
9
16
```

Hint

It may be useful to store the recursion tree of the Divide and Conquer approach to the classic maximum subarray problem.

Because the input files are large, please add

- `std::ios_base::sync_with_stdio(false);`
- `std::cin.tie(nullptr);`

to the beginning of the main function if you're using `std::cin`.

Problem 3 - Good Subpermutations (Programming) (15 points)

Problem Description

A continuous segment $[l, r]$ of a permutation P is *good* if the value of $(P_l, P_{l+1}, \ldots, P_r)$ is also continuous. That is, if we collect $P_l, P_{l+1}, \ldots, P_r$ into a new list B and sort it, then $B_1 = B_2 - 1, B_2 = B_3 - 1, \ldots, B_{i-1} = B_i - 1, \ldots, B_{|B|-1} = B_{|B|} - 1$ hold.

Given a permutation P of length N, count the number of good segments. Note that a sequence A of N integers is called a permutation of length N if $1 \leq A_i \leq N$ for all $1 \leq i \leq N$ and $A_i \neq A_j$ for all $1 \leq i \neq j \leq N$.

Input

The first line contains an integer N indicating the length of the permutation, where $1 \leq N \leq 5 \cdot 10^5$. The second line contains N integers P_i indicating the input sequence. It's guaranteed that the input sequence is a permutation of length N.

Test Group 0 (10 %)

- $N \leq 200$

Test Group 1 (10 %)

- $N \leq 2000$

Test Group 2 (20 %)

- $N \leq 5000$

Test Group 3 (50 %)

- $N \leq 10^5$

Test Group 4 (10 %)

- No other constraints.

Output

Print an integer indicating the number of *good segments* in the given permutation.

Sample Input 1

```
5
1 2 3 4 5
```

Sample Output 1

```
15
```

Sample Input 2

```
5
5 1 2 4 3
```

Sample Output 2

```
10
```

Sample Input 3

```
10
7 6 5 8 9 10 1 2 3 4
```

Sample Output 3

```
26
```

Sample Input 4

```
20
1 2 3 4 10 9 8 7 6 5 11 14 13 12 20 19 18 17 16 15
```

Sample Output 4

```
81
```

Hint

The following observation might be useful:

- The value of a set of numbers S is continuous if and only if

$$\max_{x \in S} x - \min_{y \in S} y = |S| - 1$$

Problem 4 - Different Strings (Programming) (15 points)

Problem Description

You are given two strings S_1 and S_2, and you have 3 string operations:

- **Insertion** of a single character. If $a = uv$, then inserting a character x between u and v produces uxv.
- **Deletion** of a single character. For example, deleting x changes uxv to uv.
- **Substitution** of a single character. For example, substituting x for a character $y \neq x$ changes uxv to uyv.

Your goal is to use these operations to make all the corresponding characters in these two strings different. To be more specific, we need to change S_1 to S_1' and S_2 to S_2', such that:

$$S_1'[i] \neq S_2'[i] \text{ for } 0 \leq i < \min(|S_1'|, |S_2'|)$$

You need to output the minimum number of operations needed to change the two strings.

Input

There are two lines, which are S_1 and S_2.

- S_1 and S_2 contain only uppercase and lowercase alphabets.
- $|S_1|, |S_2| \leq 2000$

Output

An integer indicates the minimum number of operations needed.

Subtask 1 (25 %)

- S_1 and S_2 contains only alphabet "A".

Subtask 2 (25 %)

- S_1 and S_2 contains only alphabets "A" and "B".

Subtask 3 (50 %)

- No other constraints.

Sample Input 1

```
abcdcdc
abcddc
```

Sample Output 1

```
2
```

Sample Input 2

```
AAAAA
AAAA
```

Sample Output 2

```
4
```

Problem 5 - Time Complexity & Recurrence (Hand-Written) (15 points)

Note: In this problem, if you use any theorem not covered by the lectures, slides, and the textbook, you should prove it first.

(1) **Asymptotic Notations** (5%, 1% each)

True or *False*.
If your answer is *True*, no explanation is needed.
If your answer is *False*, give a **counterexample**, or **briefly** explain the reason.

Note: All functions ($f(n)$, $g(n)$) mentioned below are **non-negative** and **increasing**.

(a) $f(n) + g(n) = O(\min(f(n), g(n)))$
(b) $e^{f(n)} = O(e^{g(n)})$ implies $f(n) = O(g(n))$
(c) if $g(n) = o(f(n))$, then $f(n) + g(n) = \Theta(f(n))$
(d) $f(n) = \Theta(f(\frac{n}{2}))$
(e) $\log_2(n!) = \Omega(n^2)$

(2) **Solve Recurrences** (10%)

Give the tight bound (Θ-bound, *e.g.* $T(n) = \Theta(n^3)$) of the following recurrence equations.

Assume: $T(n) = 1, \forall n \leq 1$

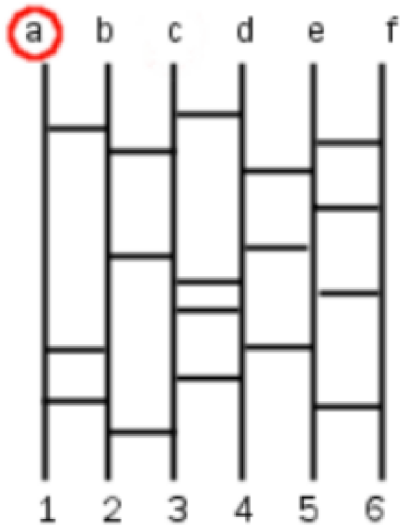
Note: Show your derivation thoroughly. Following hints is strongly recommended.

(a) (2%) $T(n) = 6T(\frac{n}{3}) + 108n$
(b) (2%) $T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + T(\frac{n}{12}) + 24n$
(c) (3%) $T(n) = \sqrt{n}T(\sqrt{n}) + 2n \lg n$
(*hint: use the transformation* $S(n) = \frac{T(n)}{n}$)
(d) (3%) $T(n) = 2T(\frac{n}{2}) + \frac{4n}{\lg n}$
(*hint: expand the equation to observe the pattern; recall* $\sum_{k=1}^{n} \frac{1}{k} \approx \ln n$)

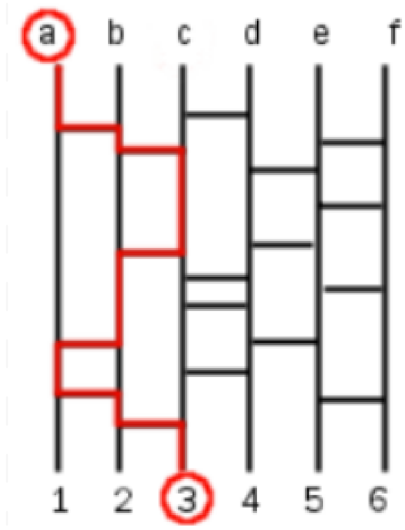
Problem 6 - Controllable Ghost Leg (Hand-Written) (20 points)

"Ghost Leg", known in Japan as Amidakuji ("Amida lottery" because the paper was folded into a fan shape resembling Amida's halo), is a method of lottery designed to create random pairings between two sets of the same number of prizes. This is often used to distribute prizes to people, where the number of prizes is the same as the number of people.

Ghost Leg consists of vertical lines with horizontal lines connecting two adjacent vertical lines; the horizontal lines are called "legs". Note that "legs" cannot touch other legs. The number of vertical lines equals the number of people, and at the bottom of each line there is an item - a prize that will be paired with a player. The general rule for this game is: choosing a line on the top, and following it downwards. When a horizontal line is encountered, the player needs to transit to another vertical line and then continue going down. The procedure is repeated until reaching the end of the vertical line, and then the player gets the corresponding prize. Therefore, choosing the line decides which prize you get. You can refer to the link for more details: `https://en.wikipedia.org/wiki/Ghost_Leg`.



The example of one ghost leg.



Example of finding the prize if A is chosen.

You, the host of the party, want to distribute N prizes to N players by ghost leg. In addition, you plan to perform some black-box tricks on this game so that the person choosing a specific start must be able to get a specific prize you planned.

Here comes a problem: Now given a ghost leg board without any horizontal lines, you have k constraints, which assign k starting points with their corresponding prizes. What is the minimum number of horizontal lines to be added in order to satisfy all k constraints?

For example, you have a ghost leg without any horizontal lines as Fig. 1, and the constraints are {(c should go to 1), (b should go to 4)}. Thus, the answer should be 3, and one of the solution is illustrated in Fig. 2.

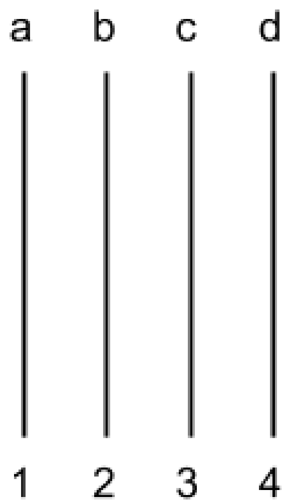


Fig 1. An empty ghost leg if $N = 4$

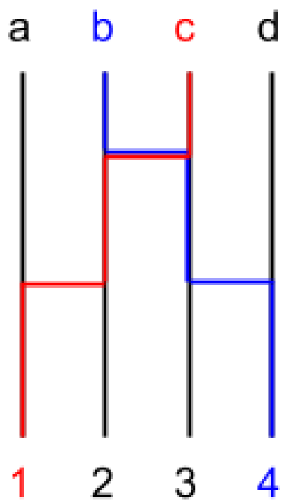


Fig 2. A solution meets all the constraints

Before solving this problem, "inversion" is introduced for helping you find the solution.

Inversion Given a sequence of unique numbers $B = b_1, b_2, ..., b_n$, an inversion is a pair of numbers b_i and b_j in this sequence such that $i < j$ and $b_i > b_j$. Let $I(B)$ be the set of inversions in B. For example, if $B = 1, 3, 5, 2$, $I(B) = \{(3, 2), (5, 2)\}$, and the number of inversions is 2 ($|I(B)| = 2$).

(1) Given an unsorted and unique sequence B with n numbers, please design an efficient algorithm to calculate the number of inversions $|I(B)|$ in $O(N \log N)$ time. (7 points)

(2) Explain why your algorithm runs in $O(N \log N)$. (5 points)

(3) Prove that the number of exchanges when performing bubble sort on the sequence S is equal to the number of inversions in S. (2 points)

Controllable Ghost Leg

(4) Describe a $O(N \log N)$ algorithm that calculates the minimum number of horizontal lines when $|constraints| = N$. You are required to prove the correctness in this problem. (2 points)

(5) Describe a $O(N \log N)$ algorithm that calculates the minimum number of horizontal lines when $|constraints| \leq N$. You are required to prove the correctness in this problem. (4 points)

Problem 7 - Folding Blocks (Hand-Written) (15 points)

"Folding Blocks" is a puzzle game, in which you need to unfold the blocks in order to fill the whole space. When you unfold a block, the block is extended with the same size to the unfolding direction. You can play the 2D version of this game here: `https://www.crazygames.com/game/folding-blocks-puzzle`.

Here we play the puzzle on a 1D board, where the whole space can be viewed as a line illustrated below. If a 2-length block initially fills 3-4 positions, and now we unfold it to the right side, it will fill 3-6 positions of the board. Similarly, if we unfold it to the left side, it will fill 1-4 positions of the board.

Now given the length of the board, denoted as N, and the initial status (each block's length and position), please decide whether this puzzle can be solved. Note that the given position of the block is its leftmost side and a puzzle is *solved* once all positions are filled after unfolding without overlapping or out-of-board blocks.

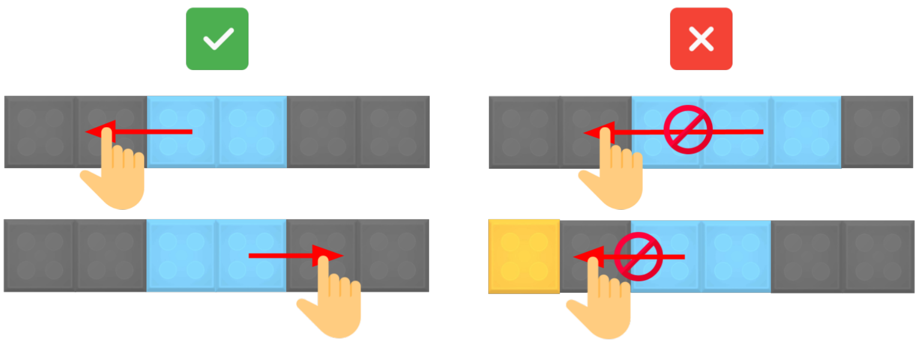


Fig 1. You can unfold the block to the left or right, but you cannot overlap other blocks or make blocks out of board when unfolding.

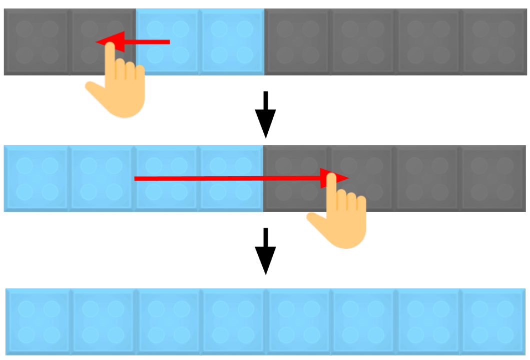


Fig 2. There's an example puzzle and solution.

(1) Let's start with a special case which has only a block on the board initially. Derive a method which can determine whether the problem is solvable in constant time. (Assume the time complexity of a logarithmic operation (log) is $O(1)$) (2 points)

(2) Assume there is only one block on the board initially. Design a $O(\log N)$ algorithm to output the sequence of unfold operations that solve the puzzle if it is solvable. (1 point)

(3) Assume there is only one block on the board and the distance between the block to the left and

right boundaries are d_1 and d_2, respectively. Prove that there are $O(d_1 + d_2)$ possibilities of the status after unfold operations. You can think of a status as a binary array representing whether each position is occupied. (3 points)

(4) Let's now consider the general case which can contain more than one block initially. Please design a DP algorithm to solve the problem in $O(N)$ time. (5 points)

(5) Explain your algorithm in (4) in terms of the properties of overlapping sub-problems and optimal substructure. (2 points)

(6) Prove that the time complexity of your DP algorithm is $\Theta(N)$. (2 points)