Question - 1

SCORE: **75 points**

Minimum Unique Array Sum

Medium | Data Structures | Greedy | Dynamic Programming | Algorithms | Arrays | Core Skills | Problem Solving

Given an array, you must increment any duplicate elements until all its elements are unique. In addition, the sum of its elements must be the minimum possible within the rules. For example, if *arr* = [3, 2, 1, 2, 7], then arr_{unique} = [3, 2, 1, 4, 7] and its elements sum to a minimal value of 3 + 2 + 1 + 4 + 7 = 17.

Function Description

Complete the *getMinimumUniqueSum* function in the editor below to create an array of unique elements with a minimal sum. Return the integer sum of the resulting array.

getMinimumUniqueSum has the following parameter(s):

arr: an array of integers to process

Constraints

- $1 \le n \le 2000$
- $1 \le arr[i] \le 3000$ where $0 \le i < n$

▸ **Input Format For Custom Testing**

▾ **Sample Case 0**

Sample Input 0

```
3
1
2
2
```

Sample Output 0

```
6
```

Explanation 0

arr = [1, 2, 2]

The duplicate array elements 2 must be addressed. The minimum unique array will be achieved by incrementing one of the twos by 1, creating the array [1, 2, 3]. The sum of elements in the new array is 1 + 2 + 3 = 6.

▸ **Sample Case 1**

▸ **Sample Case 2**

Question - 2

SCORE: **100 points**

Super Stack

Dynamic Programming | Hard | Algorithms | Core Skills | Problem Solving

Implement a simple *stack* that accepts the following commands

and performs the operations associated with them:

- `push k`: Push integer *k* onto the top of the stack.
- `pop`: Pop the top element from the stack.
- `inc e k`: Add *k* to each of the bottom *e* elements of the stack.

Function Description

Complete the function *superStack* in the editor below. The function must create an empty stack and perform each of the operations in order. After performing each operation, print the value of the stack's *top* element on a new line. If the stack is empty, print `EMPTY` instead.

superStack has the following parameter(s):

operations[operations[0],...operations[n-1]]: an array of strings

Constraints

- $1 \leq n \leq 2 \times 10^5$
- $-10^9 \leq k \leq 10^9$
- $1 \leq e \leq |S|$, where $|S|$ is the size of the stack at the time of the operation.
- It is guaranteed that *pop* is never called on an empty stack.

▸ Input Format for Custom Testing

▾ Sample Case 0

Sample Input 0

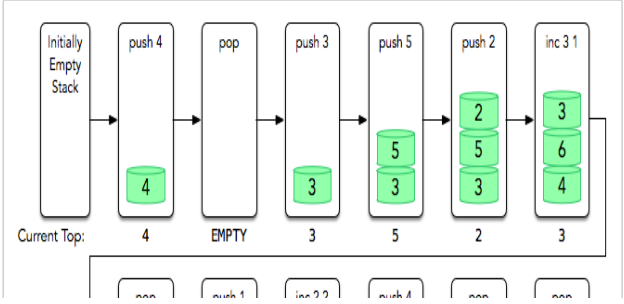
```
12
push 4
pop
push 3
push 5
push 2
inc 3 1
pop
push 1
inc 2 2
push 4
pop
pop
```

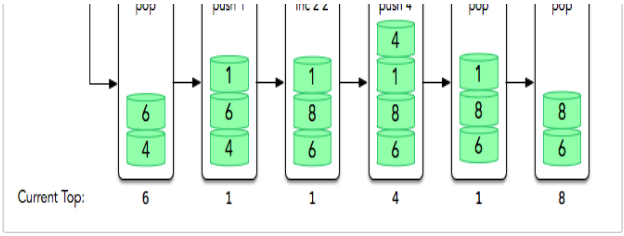
Sample Output 0

```
4
EMPTY
3
5
2
3
6
1
1
4
1
8
```

Explanation 0

The diagram below depicts the stack after each operation:





After each operation, we print the value denoted by *Current Top* on a new line.

In other words, we have an empty stack, *S*, we express as an array where the leftmost element is the *bottom* of the stack and the rightmost element is its *top*. We perform the following sequence of *n = 12* operations as given in the *operations* array:

1. `push 4`: Push *4* onto the top of the stack, so *S = [4]*. We then print the top (rightmost) element, *4*, on a new line.
2. `pop`: Pop the top element from the stack, so *S = []*. Because the stack is now empty, we print *EMPTY* on a new line.
3. `push 3`: Push *3* onto the top of the stack, *S = [3]*. Print 3, and the top of the stack after each of the following operations.
4. `push 5`: Push *5* onto the top of the stack, *S = [3, 5]*.
5. `push 2`: Push *2* onto the top of the stack, *S = [3, 5, 2]*.
6. `inc 3 1`: Add *k = 1* to bottom *e = 3* elements of the stack, *S = [4, 6, 3]*.
7. `pop`: Pop the top element from the stack, *S = [4, 6]*.
8. `push 1`: Push *1* onto the top of the stack, *S = [4, 6, 1]*.
9. `inc 2 2`: Add *k = 2* to bottom *e = 2* elements of the stack, *S = [6, 8, 1]*.
10. `push 4`: Push *4* onto the top of the stack, *S = [6, 8, 1, 4]*.
11. `pop`: Pop the top element from the stack, *S = [6, 8, 1]*.
12. `pop`: Pop the top element from the stack, *S = [6, 8]*.

Question - 3

SCORE: 75 points

Counting Pairs

Binary Search | Data Structures | Medium | Algorithms | Arrays | Core Skills | Problem Solving

Two pairs of integers *(a, b)* and *(c, d)* are considered distinct if at least one element of *(a, b)* $\notin$ *(c, d)*. For example given a list *(1, 2, 2, 3)*, *(1, 2)* is distinct from *(1, 3)* and *(2, 3)* but not from *(1, 2)* with *2* chosen from a different index in the list. A pair is valid if $a \leq b$.

You will be given an integer *k* and a list of integers. Count the number of distinct valid pairs of integers *(a, b)* in the list for which $a + k = b$.

For example, the array *[1, 1, 1, 2]* has two different valid pairs: *(1, 1)* and *(1, 2)*. Note that the three possible instances of pair *(1, 1)* count as a single valid pair, as do the three possible instances of pair *(1, 2)*. If $k = 1$, then this means we have a total of *1* valid pair which satisfies $a + k = b \Rightarrow 1 + 1 = 2$, the pair *(1, 2)*.

Function Description

Complete the function *countPairs* in the editor below. The function must return an integer denoting the count of valid *(a, b)* pairs in the *numbers* array that have a difference of *k*.

countPairs has the following parameter(s):

numbers[numbers[0],...numbers[n-1]]: an array of integers
k: an integer

Constraints

- $2 \leq n \leq 2 \times 10^5$
- $0 \leq numbers[i] \leq 10^9$, where $0 \leq i < n$
- $0 \leq k \leq 10^9$

▸ **Input Format for Custom Testing**

▾ **Sample Case 0**

Sample Input 0

```
6
1
1
2
2
3
3
1
```

Sample Output 0

```
2
```

Explanation 0

There two valid pairs in *[1, 1, 2, 2, 3, 3]* that have a difference of $k = 1$:

1. (1, 2)
2. (2, 3)

▸ **Sample Case 1**

▸ **Sample Case 2**