1. Bitwise AND

Given an array of non-negative integers, count the number of unordered pairs of array elements such that their bitwise AND is a power of 2.

For example, let's say the array is *arr = [10, 7, 2, 8, 3]*, and let '&' denote the bitwise AND operator. There are 6 unordered pairs of its elements that have a bitwise AND that is a power of two:

- For indices (0,1), 10 & 7 = 2, which is a power of 2.
- For indices (0,2), 10 & 2 = 2, which is a power of 2.
- For indices (0,3), 10 & 8 = 8, which is a power of 2.
- For indices (0,4), 10 & 3 = 2, which is a power of 2.
- For indices (1,2), 7 & 2 = 2, which is a power of 2.
- For indices (2,4), 2 & 3 = 2, which is a power of 2.

Therefore, the answer is 6.

Function Description
Complete the function *countPairs* in the editor below.

countPairs has the following parameter:
  int *arr[n]:* an array of integers
Returns:
  int: the number of unordered pairs of elements of *arr* such that their bitwise AND is a power of 2

Constraints

- $1 \le n \le 2*10^5$
- $0 \le arr[i] < 2^{12}$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of elements in *arr*.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains an integer describing *arr[i]*.

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN       Function
-----       --------
4       =>  n = 4
1       =>  arr = [1, 2, 1, 3]
2
1
3
```

Sample Output

```
4
```

Explanation

All unordered pair of elements whose bitwise AND is a power of 2 are:

- For indices (0,2), 1 & 1 = 1, which is a power of 2.
- For indices (0,3), 1 & 3 = 1, which is a power of 2.
- For indices (1,3), 2 & 3 = 2, which is a power of 2.
- For indices (2,3), 1 & 3 = 1, which is a power of 2.

Therefore, the answer is 4.

▼ Sample Case 1

Sample Input For Custom Testing

```
3
0
2
4
```

Sample Output

```
0
```

Explanation

There are no pairs of array elements such that their bitwise AND is a power of 2. Therefore, the answer is 0.

```
def countPairs(arr):   #inefficient algo, timed out
    cnt = 0
    n = len(arr)
    for i in range(n-1):
        for j in range(i+1,n):
            x = arr[i] & arr[j]
            if (x & (x-1) == 0) and x != 0:
                cnt += 1
    return cnt

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    arr_count = int(input().strip())
    arr = []
    for _ in range(arr_count):
        arr_item = int(input().strip())
        arr.append(arr_item)
    result = countPairs(arr)
    fptr.write(str(result) + '\n')
    fptr.close()
```

```
def check_power2(x):   #try this?
    # returns true if x is power of 2
    # ie, x = arr[i] & arr[j]
    # return (x & (x-1) == 0) and x != 0
    return x and (not(x & (x-1)))

def countPairs(arr):
    cnt = 0
    n = len(arr)
    for i in range(n-1):
        for j in range(i+1, n):
            if check_power2(arr[i] & arr[j]):
                cnt += 1
    return cnt
```

2. Task of Pairing

A company sells dumbbells in pairs. These are weights for exercising. They receive a shipment of dumbbells weighing anywhere from 1 unit up to a certain maximum. A pair can only be sold if their weights are sufficiently close: no greater than 1 unit difference. Given an inventory of various weights, determine the maximum number of pairs the company can sell.

For example, if there are 2 dumbbells of weight 1, 4 of weight 2, 3 of weight 3 and 1 of weight 4, they can be paired as [1,1], [2,2], [2,2], [3,3], [3,4] for a total of 5 pairs.

Function Description

Complete the function *taskOfPairing* in the editor below. The function must return an integer representing the maximum number of similar pairs that can be made from the given supply of weights.

taskOfPairing has the following parameter(s):

freq[0... n-1]: a frequency array of integers where the i^{th} element represents the number of dumbbells having a weight of *i+1*.

Constraints

- $1 \leq n \leq 10^5$
- $0 \leq freq[i] \leq 10^9$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the upper limit for the weight of the dumbbells, the size of *freq*.

Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains an integer *freq[i]* which represents the number of balls having weight *i + 1*.

▼ Sample Case 0

Sample Input For Custom Testing

```
4
3
5
4
3
```

Sample Output

```
7
```

Explanation

n = 4

freq = [3, 5, 4, 3]

One possible maximum pairing is [1,1],[1,2],[2,2],[2,2],[3,3],[3,4],[4,4] making 7 similar pairs.

▼ Sample Case 1

Sample Input For Custom Testing

```
3
5
6
2
```

Sample Output

```
6
```

Explanation

n = 3

freq = [5, 6, 2]

One possible maximum pairing is [1,1],[1,1],[2,2],[2,2],[3,2],[3,2] making 6 similar pairs.

```
def taskOfPairing(freq):
    # Write your code here
    lis = []
    loan = 0
    for i in range(len(freq)-1):
        freq[i] += loan
        if freq[i]%2==0:   #even
            lis.append(freq[i])
            loan = 0
        elif freq[i]%2==1 and freq[i+1]>0:   #odd and next number is not zero
            lis.append(freq[i] + 1)
            loan = -1
        elif freq[i]%2==1 and freq[i+1]==0:   #odd and next number is zero
            lis.append(freq[i] - 1)
            loan = 0

    result = sum(lis)/2 + ((freq[-1] + loan) //2)   #add even number of final term
    return int(result)

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    freq_count = int(input().strip())
    freq = []
    for _ in range(freq_count):
        freq_item = int(input().strip())
        freq.append(freq_item)
    result = taskOfPairing(freq)
    fptr.write(str(result) + '\n')
    fptr.close()
```

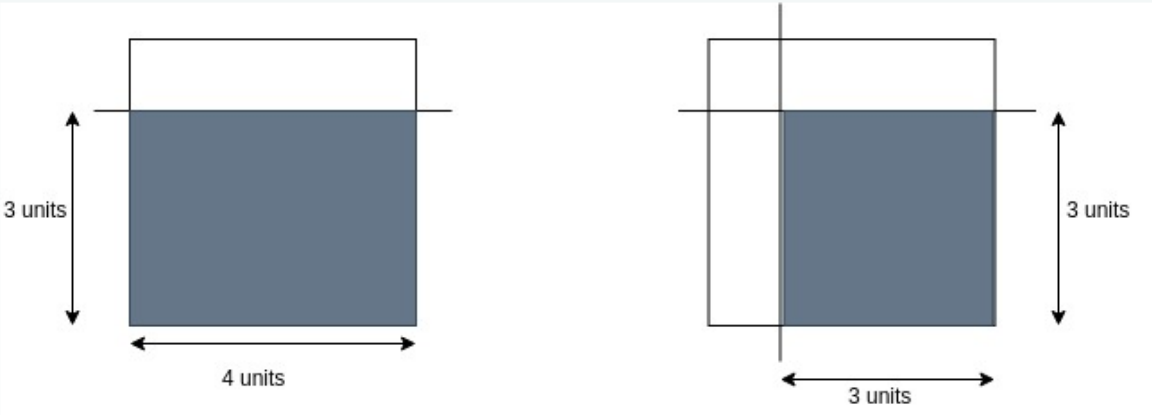
Compiled successfully. All available test cases passed

3. Largest Area

Given a rectangle, add vertical and horizontal separators at various locations. Determine the area of the largest open space after each line is added. Return these values in an array.

Example

w = 4
h = 4
isVertical = [0, 1]
distance = [3, 1]



The values in *isVertical* indicate whether a boundary is horizontal (*isVertical[i] = 0*) or vertical (*isVertical[i] = 1*),
The values in *distance* are the distances from the boundary at 0 in either direction. In the graphics above, that is from the bottom if the boundary is horizontal and from the left if is is vertical.

In the first graph, a horizontal line (*isVertical[0] = 0*) is created *distance[0] = 3* units above the bottom. This creates two areas of *1 x 4 = 4* and *3 x 4 = 12* units each. The largest area is *12*.

In the second graph, a vertical line (*isVertical[1] = 1*) is created at *distance[1] = 1* unit from the left. There are now four areas, the largest of which is *3 x 3 = 9* units in size.

The return array is *[12, 9]*.

Note: The horizontal and vertical lines may not be distinct. Placing a new boundary at a previous boundary location does not change the graph.

Function Description
Complete the function *getMaxArea* in the editor below.

getMaxArea has the following parameters:
int w: the width of the rectangle
int h: the height of the rectangle
bool isVertical[n]: 0 denotes a horizontal boundary and 1 denotes a vertical boundary
int distance[n]: the distance to each boundary, either from the bottom or from the left of the rectangle

Returns:
int[n]: each element *i* is the size of the largest open area after adding boundary *i*

Constraints

- $2 \le w, h \le 10^5$
- $1 \le n \le 10^5$
- It is guaranteed that *isVertical[i]* is either 0 or 1.
- $1 \le distance[i] \le w-1$ for a vertical boundary
- $1 \le distance[i] \le h-1$ for a horizontal boundary

▼ Input Format For Custom Testing

The first line contains an integer, *w*, the width of the rectangle.
The second line contains an integer, h, the height of the rectangle.
The third line contains an integer, *n*, the size of the *isVertical* array.
Each line *i* of the *n* subsequent lines (where $1 \le i \le n$) contains either 0 or 1, corresponding to *isVertical[i]*.
The next line contains an integer, *n*, the size of the *distance* array.
Each line *i* of the *n* subsequent lines (where $1 \le i \le n$) contains an integer, *distance[i]*.

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN     Function
-----     --------
2     →   w = 2
2     →   h = 2
```

```
2     →   isVertical[] size n = 2
0     →   isVertical = [0, 1]
1
2     →   distance[] size n = 2
1     →   distance = [1, 1]
1
```

Sample Output

```
2
1
```

Explanation

The rectangle is *2 x 2* units size.

The first boundary is horizontal at distance 1 from the bottom. It creates 2 smaller rectangles with areas of *2 x 1 = 2* units each.

▼ Sample Case 1

Sample Input For Custom Testing

```
STDIN    Function
-----    --------
4    →   w = 4
3    →   h = 3
2    →   isVertical[] size n = 2
1    →   isVertical = [1, 1]
1
2    →   distance[] size n = 2
1    →   distance = [1, 3]
3
```

Sample Output

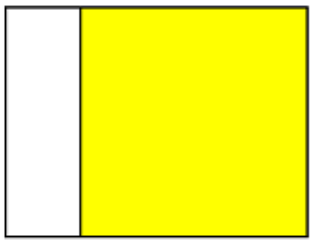
```
9
6
```

Explanation

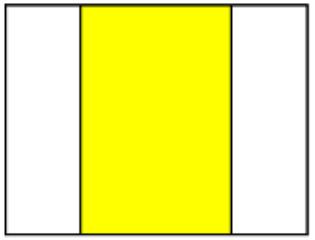
The rectangle is *4 x 3* units size.

first boundary is a vertical boundary made at a distance of 1 unit from the left. This creates 2 smaller rectangles of *1 x 3 = 3* units and *3 x 3 = 9* units.

The second boundary is vertical at position 3 from the left. This leaves 3 smaller rectangles, 2 with areas *1 x 3 = 3* units and 1 with an area of *2 x 3 = 6* units.



Max area: 3 x 3 = 9



Max area: 2 x 3 = 6

```
#!/bin/python3
import math
import os
import random
import re
import sys
#
# Complete the 'getMaxArea' function below.
#
# The function is expected to return a LONG_INTEGER_ARRAY.
# The function accepts following parameters:
#  1. INTEGER w
#  2. INTEGER h
#  3. BOOLEAN_ARRAY isVertical
#  4. INTEGER_ARRAY distance
#
```

```python
def getMaxArea(w, h, isVertical, distance):
    # Write your code here
    #(work in progress)

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    w = int(input().strip())
    h = int(input().strip())
    isVertical_count = int(input().strip())
    isVertical = []
    for _ in range(isVertical_count):
        isVertical_item = int(input().strip()) != 0
        isVertical.append(isVertical_item)
    distance_count = int(input().strip())
    distance = []
    for _ in range(distance_count):
        distance_item = int(input().strip())
        distance.append(distance_item)
    result = getMaxArea(w, h, isVertical, distance)
    fptr.write('\n'.join(map(str, result)))
    fptr.write('\n')
    fptr.close()
```

4. Equalizing Array Elements

Given an array of integers, transform it so that at least a certain number of elements in the array are equal. To achieve this, you can perform an operation where you select an element in the array and divide it by the given division parameter using integer division. What is the minimum number of operations that must be performed to achieve this goal on a certain array?

For example, let's say *arr = [1, 2, 3, 4, 5]*. The desired number of equal elements is denoted as *threshold = 3*, and the division parameter is *d = 2*. If you divide the value 4 once and the value 5 once using integer division, you get the array [1, 2, 3, 2, 2], which contains 3 equal elements. There is no way to achieve this in less than 2 operations. Therefore, the answer is 2.

Function Description
Complete the function *minOperations* in the editor below.

minOperations has the following parameter(s):
  int *arr[n]:* an array of integers
  int *threshold:* the minimum number of desired equal elements in the array
  int *d:* the division parameter used to divide an element in a single operation
Returns:
  int: the minimum number of operations required to have at least *threshold* number of equal elements in the array

Constraints

- $1 \le n \le 3*10^4$
- $1 \le arr[i] \le 2*10^5$
- $1 \le threshold \le n$
- $2 \le d \le 1000$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the size of the array.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains an integer that describes *arr[i]*.
The next line contains an integer, *threshold*, denoting the minimum number of desired equal elements in the array.
The next line contains an integer, *d*, denoting the division parameter.

▼ Sample Case 0

Sample Input For Custom Testing

```
4
64
30
25
33
2
2
```

Sample Output

```
3
```

Explanation

In this case, *arr = [64, 30, 25, 33]*, *threshold = 2*, and the division parameter *d = 2*. In other words, the minimum required number of equal elements is 2, and in one operation we can divide a single element by 2 using integer division. If we divide 64 twice to get 16, and we divide 33 once to also get 16, Then the array becomes [16, 30, 25, 16], which has 2 equal elements. There is no way to get at least 2 equal elements with fewer than 3 operations. Therefore, the answer is 3.

▼ Sample Case 1

Sample Input For Custom Testing

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

Sample Output

```
6
```

Explanation

In this case, the *arr = [1, 2, 3, 4]*, *threshold = 4*, and the division parameter *d = 3*. In other words, the minimum required number of equal elements is 4, and in one operation we can divide a single element by 3 using integer division. The only way to get all 4 elements to be equal is to divide all of them so they all become 0. One operation is required to convert 1 to 0, and another single operation is required to convert 2 to 0. Two operations are required to convert 3 to 0, and another two operations are needed to convert 4 to 0. Therefore, the total number of required operations is 1+1+2+2 = 6.

```python
#!/bin/python3
import math
import os
import random
import re
import sys
#
# Complete the 'minOperations' function below.
#
# The function is expected to return an INTEGER.
# The function accepts following parameters:
#  1. INTEGER_ARRAY arr
#  2. INTEGER threshold
#  3. INTEGER d
#

def minOperations(arr, threshold, d):
    # Write your code here
    n = len(arr)
    witness = 0
    # print(threshold, d)
    while len(set(arr)) > (n - threshold + 1):
        arr = sorted(arr)
        # print(arr)
        arr.append(arr.pop(-1) //d)
        witness += 1
    return witness

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    arr_count = int(input().strip())
    arr = []
    for _ in range(arr_count):
        arr_item = int(input().strip())
        arr.append(arr_item)
    threshold = int(input().strip())
    d = int(input().strip())
```

```
    result = minOperations(arr, threshold, d)
    fptr.write(str(result) + '\n')
    fptr.close()
```

Passed Test Case 0,1,2

Failed Test Case 3,4,5,…,14