1. Active Traders

An institutional broker wants to review their book of customers to see which are most active. Given a list of trades by customer name, determine which customers account for at least 5% of the total number of trades. Order the list alphabetically ascending by name.

Example

n = 23
customers = ["Bigcorp", "Bigcorp", "Acme", "Bigcorp", "Zork", "Zork", "Abc", "Bigcorp", "Acme", "Bigcorp", "Bigcorp", "Zork", "Bigcorp", "Zork", "Zork", "Bigcorp", "Acme", "Bigcorp", "Acme", "Bigcorp", "Acme", "Littlecorp", "Nadircorp"].

Bigcorp had 10 trades out of 23, which is 43.48% of the total trades.

Both Acme and Zork had 5 trades, which is 21.74% of the total trades.

The Littlecorp, Nadir, and Abc had 1 trade each, which is 4.35% of the total trades.

So the answer is ["Acme", "Bigcorp", "Zork"] (in alphabetical order) because only these three companies placed at least 5% of the trades.

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

mostActive has the following parameter:
string customers[n]: an array customer names

Returns
string[]: an alphabetically ascending array of customer names

Constraints

- $1 \le n \le 10^5$
- $1 \le$ length of *customers[i]* ≤ 20
- The first character of *customers[i]* is a capital English letter.
- All characters of *customers[i]* except for the first one are lowercase English letters.
- It is guaranteed that at least one customer makes at least 5% of trades.

▼ Input Format For Custom Testing

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

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN       Function
-----       --------
20      →   customers[] size n = 20
Omega   →   customers = ["Omega", "Alpha", "Omega", ..., "Beta"]
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Alpha
Omega
Beta
```

Sample Output

```
Alpha
Beta
Omega
```

Explanation
Alpha made 10 trades out of *20* (*50%* of the total), Omega made 9 trades (45% of the total), and Beta made 1 trade (*5%* of the total). All of them have met the *5%* threshold, so all the strings are returned in an alphabetically ordered array.

▼ Sample Case 1

Sample Input For Custom Testing

```
STDIN       Function
-----       --------
21      →   customers[] size n = 21
Alpha   →   customers = ["Alpha", "Beta", "Zeta", ..., "Beta"]
Beta
Zeta
Beta
Zeta
Zeta
Epsilon
Beta
Zeta
Beta
Zeta
Beta
Delta
Zeta
Beta
Zeta
Beta
Zeta
Beta
Zeta
Beta
```

Sample Output

```
Beta
Zeta
```

Explanation
Both Beta and Zeta made *9* trades out of *21* (*42.86%* of the total). Alpha, Delta and Epsilon made *1* trade each, which is only *4.76%* of the total number of trades. Only *Beta and Zeta* meet the threshold.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int compare(const char **a, const char **b)
{
  return strcmp(*a, *b);
}

char** mostActive(int customers_count, char** customers, int* result_count)
{
  qsort(customers, customers_count, sizeof(char *), compare);

  int size = 0;
  char **arr = (char **)malloc(size * sizeof(char *));

  int count = 1;
  for (int i = 0; i < customers_count - 1; i++)
  {
    if (strcmp(customers[i], customers[i + 1]) == 0)
    {
      count++;
      continue;
    }

    if (count >= customers_count * 0.05)
    {
      arr = (char **)realloc(arr, ++size * sizeof(char *));
      arr[size - 1] = customers[i];
    }
    count = 1;
  }

  if (count >= customers_count * 0.05)
  {
    arr = (char **)realloc(arr, ++size * sizeof(char *));
    arr[size - 1] = customers[customers_count - 1];
  }

  *result_count = size;
  return arr;
}
```

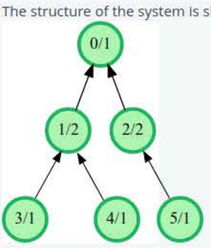
2. Balanced System Files Partition

The directory structure of a system disk partition is represented as a tree. Its *n* directories are numbered from *0* to *n-1*, where the root directory has the number *0*. The structure of the tree is defined by a *parent* array, where *parent[i] = j* means that the directory *i* is a direct subdirectory of *j*. Since the root directory does not have a parent, it will be represented as *parent[0] = -1*. The value in *files_size[i]* denotes the sum of the sizes in kilobytes of the files located in directory *i*, but excluding its subdirectories. The size of the content of a directory is defined as the size of all files contained in this directory, plus the sum of the sizes of all of its subdirectories. Partition the tree into two smaller ones by cutting one branch so that the sizes of the resulting subtrees are as close as possible.

Example

parent = [-1, 0, 0, 1, 1, 2]
files_size = [1, 2, 2, 1, 1, 1]

The structure of the system is shown in the diagram below. The nodes are labeled as *<directory>/<file_size>*.



Cut the branch between directories 1 and 0.
The partition {0, 2, 5} has size *files_size[0] + files_size[2] + files_size[5] = 1 + 2 + 1 = 4*.
The partition {1, 3, 4} has size *files_size[1] + files_size[3] + files_size[4] = 2 + 1 + 1 = 4*.
The absolute difference between the sizes of the two new trees is *4 - 4 = 0*.
Since no other partition can have a smaller absolute difference, the final answer is *0*.

Function Description

Complete the function *mostBalancedPartition* in the editor below.

The function has the following parameter(s):
int parent[n]: each *parent[i]* is the parent directory of directory *i*
int files_size[n]: each *file_sizes[i]* is the sum of file sizes in directory *i*

Returns

int: the minimum absolute difference in the size of content between two subtrees

Constraints

- $2 \le n \le 10^5$
- $1 \le files\_size[i] \le 10^4$
- *parent[0] = -1*
- *parent[i] < i* for *1 ≤ i < n*
- The depth of each directory tree is at most 500.

▼ Input Format Format for Custom Testing

The first line contains an integer, *n*.
Each *i* line of the *n* subsequent lines (where *0 ≤ i < n*) contains an integer that describes *parent[i]*.
The next line contains an integer, *n*.
Each *i* line of the *n* subsequent lines (where *0 ≤ i < n*) contains an integer that describes *files_size[i]*.

▼ Sample Case 0

Sample Input

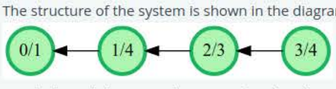
```
STDIN       Function
-----       --------
4       →   parent[] size n = 4
-1      →   parent[] = [ -1, 0, 1, 2 ]
0
1
2
4       →   files_size[] size n = 4
1       →   files_size[] = [ 1, 4, 3, 4 ]
4
3
4
```

Sample Output

```
2
```

Explanation

The structure of the system is shown in the diagram below.



Cut the branch between directories *1* and 2. This will result in partitions *{0, 1}* with size *1 + 4 = 5* and *{2, 3}* with size *3 + 4 = 7*. The absolute difference between their sizes is *|5 - 7| = 2*.

▼ Sample Case 1

Sample Input

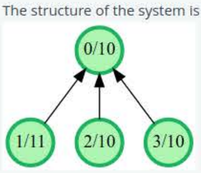
```
STDIN       Function
-----       --------
4       →   parent[] size n = 4
-1      →   parent[] = [ -1, 0, 0, 0 ]
0
0
0
4       →   files_size[] size n = 4
10      →   files_size[] = [ 10, 11, 10, 10 ]
11
10
10
```

Sample Output

```
19
```

Explanation

The structure of the system is shown in the diagram below.



Cut the branch between directories *0* and *1*. This will result in partitions *{0, 2, 3}* with size *10 + 10 + 10 = 30* and *{1}* with size *11*. The absolute difference between their sizes is *|30 - 11| = 19*.

```c
#include <stdio.h>
#include <stdlib.h>

int mostBalancedPartition(int parent_count, int* parent, int files_size_count, int* files_size)
{

  for (int i = parent_count - 1; i > 0; i--)
  {
    files_size[parent[i]] += files_size[i];
  }

  int mindiff = files_size[0];
  int diff = mindiff;
  for (int i = 1; i < files_size_count; i++)
  {
    diff = abs(files_size[0] - 2 * files_size[i]);
    mindiff = (diff < mindiff) ? diff : mindiff;
  }
  return mindiff;
}
```

2. Longest Subarray

Given an array of integers, what is the length of the longest subarray containing no more than two distinct values such that the distinct values differ by no more than 1?

Example

arr = [0,1,2,1,2,3]

The largest such subarray has length 4: [1,2,1,2].

arr = [1, 1, 1, 3, 3, 2, 2]

The largest such subarray has length 4: [3, 3, 2, 2]. The values 1 and 3 differ by more than 1 so [1, 1, 1, 3, 3] is not valid.

Function Description

Complete the function *longestSubarray* in the editor below.

longestSubarray has the following parameter(s):

int arr[n]: an array of integers

Returns:

int: the length of the longest subarray

Constraints

- The longest subarray will have fewer than 35 elements.
- $1 \le n \le 10^5$
- $1 \le arr[i] \le 10^9$

▼ 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 contains a single integer denoting *arr[i]*.

▼ Sample Case 0

Sample Input For Custom Testing

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

Sample Output

```
2
```

Explanation

n = 5

arr = [1, 2, 3, 4, 5]

All elements are distinct, so any subarray of length 2 is the maximum.

▼ Sample Case 1

Sample Input For Custom Testing

```
3
2
2
1
```

Sample Output

```
3
```

Explanation

n = 3

arr = [2, 2, 1]

The maximum subarray is length 3 (i.e. the entire array), as it contains only 2 distinct values.

```
#include <stdio.h>
#include <stdlib.h>

int longestSubarray(int arr_count, int* arr)
{
  int x, y, max;
  x = y = max = 1;

  for (int i = 1; i < arr_count; i++)
  {
    if (arr[i - 1] == arr[i])
    {
      x++;
      y++;
    }
    else if (arr[i - 1] == (arr[i] - 1))
    {
      x = y + 1;
      y = 1;
    }
    else if (arr[i - 1] == (arr[i] + 1))
    {
      y = x + 1;
      x = 1;
    }
    else
    {
      x = y = 1;
    };

    max = (x > max) ? ((x > y) ? x : y) : ((max > y) ? max : y);
  }

  return max;
}
```

1. Maximum Cost of Laptop Count

A company manufactures a fixed number of laptops every day. However, if some defect is encountered during the testing of a laptop, it is labeled as "illegal" and is not counted in the laptop count of the day. Given the cost to manufacture each laptop, its label as "illegal" or "legal", and the number of legal laptops to be manufactured each day, find the maximum cost incurred by the company in a single day in manufacturing all the laptops.

Note that if a laptop is labeled as illegal, its manufacturing cost is still incurred by the company, even though it is not included in the laptop count. Also, days are only taken into when the daily laptop count has been completely met. If there are no such days, the answer is 0.

For example, let's say there are *n = 5* laptops, where *cost = [2, 5, 3, 11, 1]*. The labels for these laptops are *labels = ["legal", "illegal", "legal", "illegal", "legal"]*. Finally, the *dailyCount = 2*, which means that the company needs to manufacture 2 legal laptops each day. The queue of laptops can be more easily viewed as follows:

- **cost 2, "legal"**
- **cost 5, "illegal"**
- **cost 3, "legal"**
- **cost 11, "illegal"**
- **cost 1, "legal"**

On the first day, the first three laptops are manufactured in order to reach the daily count of 2 legal laptops. The cost incurred on this day is 2 + 5 + 3 = 10. On the second day, the fourth and fifth laptops are manufactured, but because only one of them is legal, the daily count isn't met, so that day is not taken into consideration. Therefore, the maximum cost incurred on a single day is 10.

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

maxCost has the following parameter(s):
  int *cost[n]*: an array of integers denoting the cost to manufacture each laptop
  string *labels[n]*: an array of strings denoting the label of each laptop ("legal" or "illegal")
  int *dailyCount:* the number of **legal laptops to be manufactured each day**
Returns:
  int: the maximum cost incurred in a single day of laptop manufacturing

Constraints

- $1 \le n \le 10^5$
- $0 \le cost[i] \le 10^4$
- $1 \le dailyCount \le n$
- $labels[i] \in \{"legal", "illegal"\}$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of laptops and the size of the array *cost*.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains an integer, *cost[i]*, denoting the cost to manufacture each laptop.
The next line again contains an integer, *n*, denoting the size of the array *labels*.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains an integer, *labels[i]*, denoting the label of each laptop ("legal" or "illegal").
The last line contains an integer, *dailyCount*, denoting **the number of legal laptops to be manufactured each day.**

▼ Sample Case 0

Sample Input For Custom Testing

```
1
2
1
illegal
1
```

Sample Output

```
0
```

Explanation
Here, there is only 1 laptop manufactured, where *cost = [1]* and *labels = ["illegal"]*. Also, *dailyCount = 1*. Since the only laptop manufactured is labeled as illegal, there is no day where the daily count for legal laptops is met, so no days are taken into consideration. Therefore, the answer is 0.

▼ Sample Case 1

Sample Input For Custom Testing

```
5
0
3
2
3
4
5
legal
legal
illegal
legal
legal
1
```

Sample Output

```
5
```

Explanation
Here, there are *n = 5* laptops manufactured, and the *dailyCount = 1*. The laptops have the following costs and labels:

- **cost 0, "legal"**
- **cost 3, "legal"**
- **cost 2, "illegal"**
- **cost 3, "legal"**
- **cost 4, "legal"**

On the first day, one legal laptop is manufactured, incurring a cost of 0. Another legal laptop is manufactured on the second day, incurring a cost of 3. On the third day, there are two laptops manufactured (because one of them is illegal), incurring a cost of 2 + 3 = 5. Finally, on the fourth day, one legal laptop is manufactured, incurring a cost of 4. The greatest cost incurred in a single day is 5, which happened on the third day. Therefore, the answer is 5.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int maxCost(int cost_count, int* cost, int labels_count, char** labels, int dailyCount)
{
  int maxcost = 0;
  int sumcost = 0;
  int count = 0;
  for (int i = 0; i < labels_count; i++)
  {
    if (!strcmp(labels[i], "legal"))
    {
      count++;
    }
    sumcost += cost[i];
    if (count == dailyCount)
    {
      maxcost = maxcost > sumcost ? maxcost : sumcost;
      sumcost = 0;
      count = 0;
    }
  }

  return maxcost;
}
```

2. Nearly Similar Rectangles

Recently, while researching about similar rectangles, you found the term "Nearly Similar Rectangle." Two rectangles with sides *(a, b)* and *(c, d)* are nearly similar only if $a/c = b/d$. The order of sides matter in this definition, so rectangles [4, 2] and [6, 3] are nearly similar, but rectangles [2, 4] and [6, 3] are not. Given an array of rectangles with the lengths of their sides, calculate the number of pairs of nearly similar rectangles in the array.

For example, let's say there are $n = 4$ rectangles, and *sides = [[5, 10], [10, 10], [3, 6], [9, 9]]*. In this case, the first and third rectangles, with sides [5, 10] and [3, 6], are nearly similar because 5/3 = 10/6. Also, the second and fourth rectangles, with sides [10, 10] and [9, 9], are nearly similar because 10/9 = 10/9. This means there are 2 pairs of nearly similar rectangles in the array. Therefore, the answer is 2.

Function Description

Complete the function *nearlySimilarRectangles* in the editor below.

nearlySimilarRectangles has the following parameter:

int *sides[n][2]*: a 2-dimensional integer array where the i^{th} row denotes the sides of the i^{th} rectangle

Returns:

int: the number of nearly similar rectangles in the array

Constraints

- $1 \le n \le 10^5$
- $1 \le sides[i][0], sides[i][1] \le 10^{15}$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of rows in *sides*.

The next line contains an integer, 2, denoting the number of columns in *sides*.

Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains 2 space-separated integers, *sides[i][0]* and *sides[i][1]*, denoting the lengths of the i^{th} rectangle's sides

▼ Sample Case 0

Sample Input For Custom Testing

```
3
2
4 8
15 30
25 50
```

Sample Output

```
3
```

Explanation

In this example, *n = 3* and *sides = [[4, 8], [15, 30], [25, 50]]*.

- The first and second rectangles, with sides [4, 8] and [15, 30], are nearly similar because 4/15 = 8/30.
- The first and third rectangles, with sides [4, 8] and [25, 50], are nearly similar because 4/25 = 8/50.
- The second and third rectangles, with sides [15, 30] and [25, 50] are nearly similar because 15/25 = 30/50.

This means there are 3 pairs of nearly similar rectangles in this array. Therefore, the answer is 3.

▼ Sample Case 1

Sample Input For Custom Testing

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

Sample Output

```
0
```

Explanation

In this example, n = 5 and *sides = [[2, 1], [10, 7], [9, 5], [6, 9], [7,3]]*. There are no pairs of nearly similar rectangles in this array. Therefore, the answer is 0.

```
#include <stdio.h>
#include <stdlib.h>

// ------------------------------------  O(n log n)  Using Quick sort  ----------------------------------------------//

long gcd(long a, long b) // Greatest Common Divisor Function
{
 return b > 0 ? gcd(b, a % b) : a;
}

long compare(const long *a, const long *b) // Compare Function To Sort 2D Array
{
 return (*a - *b) ? (*a - *b) : (*(a + 1) - *(b + 1));
};


long nearlySimilarRectangles(int sides_rows, int sides_columns, long **sides)
{
 long arr[sides_rows][2];

 for (int i = 0; i < sides_rows; i++)
 {
  long z = gcd(sides[i][0], sides[i][1]);
  arr[i][0] = sides[i][0] / z;
  arr[i][1] = sides[i][1] / z;
 }

 qsort(arr, sides_rows, sizeof(arr[0]), compare);

 long count = 1;
 long sum = 0;
 long acc = 0;

 for (int i = 0; i < sides_rows - 1; i++)
 {
  if (arr[i][0] == arr[i + 1][0] && arr[i][1] == arr[i + 1][1])
  {
   count++;
   acc = ((count * (count - 1)) / 2);          // n(n-1)/2
   continue;
  }
  sum += acc;
  count = 1;
  acc = 0;
 }
 sum += acc;

 return sum;
};
```

1. Parallel Processing

A computer has a certain number of cores and a list of files that need to be executed. If a file is executed by a single core, the execution time equals the number of lines of code in the file. If the lines of code can be divided by the number of cores, another option is to execute the file in parallel using all the cores, in which case the execution time is divided by the number of cores. However, there is a limit as to how many files can be executed in parallel. Given the lengths of the code files, the number of cores, and the limit, what is the minimum amount of time needed to execute all the files?

For example, let's say that there are $n = 5$ files, where $files = [4, 1, 3, 2, 8]$ (indicating the number of lines of code in each file), $numCores = 4$, and $limit = 1$. Even though both the first and fifth files can be executed in parallel, you must choose only one of them because the limit is 1. The optimal way is to parallelize the last file, so the minimum execution time required is 4 + 1 + 3 + 2 + (8/4) = 12. Therefore, the answer is 12.

Function Description

Complete the function *minTime* in the editor below.

minTime has the following parameter(s):

int *files[n]*: an array of integers where *files[i] indicates the number of lines of code in the* i^{th} *file*

int *numCores:* the number of cores in the computer

int *limit*: the maximum number of files that can be executed in parallel

Returns:

long int: the minimum units of time needed to execute all the files

Constraints

- $1 \le n \le 10^5$
- $1 \le files[i] \le 10^9$
- $1 \le numCores \le 10^9$
- $1 \le limit \le 10^9$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of *files* on the computer.

Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains a long integer, *files[i]*, denoting the *number of lines of code in the* i^{th} *file*.

The next line contains a long integer, *numCores*, denoting the number of cores in the computer.

The last line contains a long integer, *limit*, denoting the max number of files that can be executed in parallel.

▼ Sample Case 0

Sample Input For Custom Testing

```
3
5
3
1
5
5
```

Sample Output

```
5
```

Explanation

Here, there are $n = 3$ files on the computer, where $files = [5, 3, 1]$, $numCores = 5$, and $limit = 5$. Even though we can parallelize up to 5 pieces of code, we only parallelize the first file because the lines of code can be divided between cores equally. So, the minimum time required is (5/5) + 3 + 1 = 5. Therefore, the answer is 5.

▼ Sample Case 1

Sample Input For Custom Testing

```
3
3
1
5
1
5
```

Sample Output

```
9
```

Explanation

Here, there are $n = 3$ files on the computer, where $files = [3, 1, 5]$, $numCores = 1$, and $limit = 5$. Even though we can parallelize up to 5 pieces of code, we only have 1 core, so parallelization won't help us. So, the minimum time required is 3 + 1 + 5 = 9. Therefore, the answer is 9.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
long compare(long *a, long *b)
{
  return *b - *a;
}

long minTime(int files_count, int* files, int numCores, int limit)
{
  long total = 0;
  int n = 0;
  long *max = (long *)calloc(n, sizeof(long));
  for (int i = 0; i < files_count; i++)
  {
    if (!(files[i] % numCores))
    {
      max = (long *)realloc(max, ++n * sizeof(long));
      max[n - 1] = files[i];
      continue;
    }
    total += files[i];
  }

  qsort(max, n, sizeof(long), compare);

  for (int i = 0; i < n; i++)
  {
    if (i < limit)
    {
      total += (max[i] / numCores);
    }
    else
    {
      total += max[i];
    }
  }
  free(max);
  return total;
}
```

2. Password Decryption

In a computer security course, you just learned about password decryption. Your fellow student has created their own password encryption method, and they've asked you to test how secure it is. Your task is to recover the original password given the encrypted password provided to you by your classmate.

At first, it seems impossible. But one day after class, you catch a peek of your classmate's notebook where the encryption process is noted. You snap a quick picture of it to reference later. It says this:

Given string *s*, let *s[i]* represent the i^{th} character in the string *s*, using 0-based indexing.

1. Initially i = 0.
2. If *s[i]* is lowercase and the next character *s[i+1]* is uppercase, swap them, add a '*' after them, and move to *i+2*.
3. If *s[i]* is a number, replace it with *0*, place the original number at the start, and move to *i+1*.
4. Else, move to *i+1*.
5. Stop if *i* is more than or equal to the string length. Otherwise, go to step 2.

There's even an example mentioned in the notebook. When encrypted, the string "hAck3rr4nk" becomes "43Ah*ck0rr0nk". (*Note*: the original string, "hAck3rr4nk", does not contain the character *0*.)

Note:

- The original string always contains digits from 1 to 9 and does not contain 0.
- The original string always contains only alpha-numeric characters.

Using this information, your task is to determine the original password (before encryption) when given the encrypted password from your classmate.

Function Description
Complete the function *decryptPassword* in the editor below. *decryptPassword* must return the original password string before it was encrypted by your classmate.

decryptPassword has the following parameter:
s: the password string after it was encrypted by your classmate

Constraints

- $1 \leq$ length of $s \leq 10^5$
- *s* can contain Latin alphabet characters (a-z, A-Z), numbers (0-9), and the character '*'.

▼ Input Format For Custom Testing

The first line contains the password string obtained after it was encrypted by your classmate.

▼ Sample Case 0

Sample Input For Custom Testing

```
51Pa*0Lp*0e
```

Sample Output

```
aP1pL5e
```

Explanation
If we apply the sequence of operations on the string aP1pL5e, we get the string 51Pa*0Lp*0e.

1. We start at the letter *a* since i = 0.
2. Since a is lowercase and the next character *P* is uppercase, we swap them, add a '*' after, and move to the next designated character (i+2). So currently it is **Pa*1pL5e**.
3. Now we're on the character *1*. This is a number, so we replace it with *0*, put the original number *1* at the start, and continue to the next character (i+1). Now it is **1Pa*0pL5e**.
4. We're still in the middle of the string (*i* does not equal the string length), so we repeat the process again.

After that, **1Pa*0pL5e** becomes **1Pa*0Lp*5e**. Then, **1Pa*0Lp*5e** becomes **51Pa*0Lp*0e**. Since *e* is at the end of the string, it can't be swapped with the next character. Thus, aP1pL5e becomes 51Pa*0Lp*0e when encrypted.

▼ Sample Case 1

Sample Input For Custom Testing

```
pTo*Ta*0
```

Sample Output

```
poTaT0
```

Explanation
If we apply the sequence of operations on the string poTaTO, we get the string pTo*Ta*O.

1. We start at the letter *p* since i = 0.
2. The character *p* is lowercased, but the next character is also lowercased. So there's no need to swap them.
3. We move on to the next character (i+1), which is *o*. Now, since *o* is followed by capital *T*, we swap them, add a '*' after, and move to the next designated character (i+2). So currently it is **pTo*aTO**.
4. Moving to character *a*, it is followed by a capitalized letter, so we likewise swap these, add a '*' after, and move to i+2. Now it is **pTo*Ta*O**.
5. *O* is at the end of the string, we stop there.

Thus, poTaTO becomes pTo*Ta*O when encrypted.

```
char *decryptPassword(char *s)
{
  int len = strlen(s);
  int j = 0;
  int x = 0;
  char *newS = (char *)calloc(x, sizeof(char));
  for (int i = 0; i < len; i++)
  {
    if (isdigit(s[i]) && s[i] != '0')
    {
      j++;
      continue;
    }
    if ((i < len - 2) && s[i + 2] == '*')
    {
      x += 2;
      newS = (char *)realloc(newS, x * sizeof(char));
      newS[x - 2] = s[i + 1];
      newS[x - 1] = s[i];
      i += 2;
    }
    else if (s[i] == '0')
    {

      newS = (char *)realloc(newS, ++x * sizeof(char));
      newS[x - 1] = s[j - 1];
      j--;
    }
    else
    {

      newS = (char *)realloc(newS, ++x * sizeof(char));
      newS[x - 1] = s[i];
    }
  }

newS = (char *)realloc(newS, ++x * sizeof(char));
newS[x-1]='\0';

  return newS;
}
```

```
char * decryptPassword(char *s)
{
  int len = strlen(s);
  char temp;
  int j = 0;
  int i = 0;
  int x = 0;
  char *numbers = (char *)calloc(x, sizeof(char));
  for (i; i < len; i++)
  {
    if (isdigit(s[i]) && s[i] != '0')
    {
      numbers = (char *)realloc(numbers, ++x * sizeof(char));
      numbers[x - 1] = s[i];
      j++;
      continue;
    }

    if (s[i] == '*')
    {

      temp = s[i - j - 1];
      s[i - j - 1] = s[i - j - 2];
      s[i - j - 2] = temp;
      j++;
    }
    else if (s[i] == '0')
    {
      s[i - j] = numbers[(x--) - 1];
    }
    else
    {
      s[i - j] = s[i];
    }
  }

  {
    s[i - j] = '\0';
  }

  free(numbers);
  return s;
}
```

1. Road Repair

A number of points along the highway are in need of repair. An equal number of crews are available, stationed at various points along the highway. They must move along the highway to reach an assigned point. Given that one crew must be assigned to each job, what is the minimum total amount of distance traveled by all crews before they can begin work?

For example, given crews at points {1,3,5} and required repairs at {3,5,7}, one possible minimum assignment would be {1→ 3, 3 → 5, 5 → 7} for a total of 6 units traveled.

Function Description
Complete the function *getMinCost* in the editor below. The function should return the minimum possible total distance traveled as an integer.

getMinCost has the following parameter(s):
crewId[crewId[0],...crewId[n-1]] : a vector of integers
jobId[jobId[0],...jobId[n-1]] : a vector of integers

Constraints

- $1 \leq n \leq 10^5$
- $crewId[i] : 1 \leq crewId[i] \leq 10^9$
- $jobId[i] : 1 \leq jobId[i] \leq 10^9$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of elements in *crew_id* and *job_id*.
Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains an integer describing *crew_id[i]*.
The next line again contains the integer *n* as above.
Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains an integer describing *job_id[i]*.

▼ Sample Case 0

Sample Input For Custom Testing

```
5
5
3
1
4
6
5
9
8
3
15
1
```

Sample Output

```
17
```

Explanation
By index, *crewId[i]* → *jobId[i]*, { (0 → 0) , (1 → 2) , (2 → 4) , (3 → 3) , (4 → 1) } is one possible assignment for a minimum cost of 17. Showing element values, this is { (5 → 9) , (3 → 3) , (1 → 1) , (4 → 15) , (6 → 8) } yielding a total travel distance of 4 + 0 + 0 + 11 + 2 = 17.

▼ Sample Case 1

Sample Input For Custom Testing

```
4
5
1
4
2
4
4
7
9
10
```

Sample Output

```
18
```

Explanation
By index, { (1 → 0) , (0 → 1) , (2 → 2) , (3 → 3) } is one possible assignment for a minimum cost of 18.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int compare(int *a, int *b)
{
  return *b - *a;
}
long getMinCost(int crew_id_count, int* crew_id, int job_id_count, int* job_id)
{
  qsort(crew_id, crew_id_count, sizeof(int), compare);
  qsort(job_id, job_id_count, sizeof(int), compare);

  long totalDistance = 0;
  for (int i = 0; i < crew_id_count; i++)
  {

    totalDistance += abs(crew_id[i] - job_id[i]);
  }

  return totalDistance;
}
```

1. String Anagram

An anagram of a string is another string with the same characters in the same frequency, in any order. For example *'abc', 'bca', 'acb', 'bac', 'cba', 'cab'* are all anagrams of the string *'abc'*. Given two arrays of strings, for every string in one list, determine how many anagrams of it are in the other list. Write a function that receives *dictionary* and *query*, two string arrays. It should return an array of integers where each element *i* contains the number of anagrams of *query[i]* that exist in *dictionary*.

Example
dictionary = ['hack', 'a', 'rank', 'khac', 'ackh', 'kran', 'rankhacker', 'a', 'ab', 'ba', 'stairs', 'raits']
query = ["a", "nark", "bs", "hack", "stair"]

query[0] = 'a' has *2* anagrams in *dictionary*: *'a'* and *'a'*.
query[1] = 'nark' has *2* anagrams in *dictionary*: *'rank'* and *'kran'*.
query[2] = 'bs' has *0* anagrams in *dictionary*.
query[3] = 'hack' has *3* anagrams in *dictionary*: *'hack'*, *'khac'* and *'ackh'*.
query[4] = 'stair' has *1* anagram in *dictionary*: *'raits'*. While the characters are the same in *'stairs'*, the frequency of *'s'* differs, so it is not an anagram.
The final answer is *[2, 2, 0, 3, 1]*.

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

stringAnagram has the following parameters:
string dictionary[n]: an array of strings to search in
string query[q]: an array of strings to search for

Returns
int[q]: an array of integers where the i^{th} value is the answer to *query[i]*

Constraints

- $1 \le length(dictionary), length(query) \le 10^5$
- $1 \le length(dictionary[i]) \le 15$
- $1 \le length(query[i]) \le 15$
- *Every string consists of lowercase English letters.*

▼ Input Format For Custom Testing

The first line of input contains an integer, *n*, the number of strings in *dictionary[]*.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains a string, *dictionary[i]*.
The next line contains an integer, *q*, the number of strings in *query[]*.
Each line *i* of the *q* subsequent lines (where $0 \le i < q$) contains a string, *query[i]*.

▼ Sample Case 0

Sample Input

```
STDIN     Function
-----     -----
5      →  dictionary[] size n = 5
heater →  dictionary = ['heater', 'cold', 'clod', 'reheat', 'docl']
cold
clod
reheat
docl
3      →  query[] size q = 3
codl   →  query = ['codl', 'heater', 'abcd']
heater
abcd
```

Sample Output

```
3
2
0
```

Explanation
query[0] = 'codl' has *3* anagrams in *dictionary*: *'cold'*, *'clod'* and *'docl'*.
query[1] = 'heater' has *2* anagrams in *dictionary*: *'heater'* and *'reheat'*.
query[2] = 'abcd' has *0* anagrams in *dictionary*.
The final answer is *[3, 2, 0]*.

▼ Sample Case 1

Sample Input

```
STDIN     Function
-----     -----
8      →  dictionary[] size n = 8
listen →  dictionary = ['listen', 'tow', 'silent', 'lisent', 'two', 'abc', 'no', 'on']
tow
silent
lisent
two
abc
no
on
4      →  query[] size q = 4
two    →  query = ['two', 'bca', 'no', 'listen']
bca
no
listen
```

Sample Output

```
2
1
2
3
```

Explanation
query[0] = 'two' has *2* anagrams in *dictionary*: *'tow'* and *'two'*.
query[1] = 'bca' has *1* anagram in *dictionary*: *'abc'*.
query[2] = 'no' has *2* anagrams in *dictionary*: *'no'* and *'on'*.
query[3] = 'listen' has *3* anagrams in *dictionary*: *'listen'*, *'silent'* and *'lisent'*.
The final answer is *[2, 1, 2, 3]*.

```c
#include <stdio.h>
#include <stdlib.h>
int compare(const char **a, const char **b)
{
  return strcmp(*a, *b);
}

int compare_chars(const char *a, const char *b)
{

  return *a - *b;
}
int binarySearchString(char **arr, int n, char target[])      //------------ log(n) ----------//
{
  int low = 0;
  int high = n - 1;
  int count = 0;
  int i = 0;
  while (low <= high)
  {

    i++;
    int mid = low + (high - low) / 2;
    int cmp = strcmp(arr[mid], target);
    if (cmp == 0)
    {
      count++;
      int left = mid - 1;
      int right = mid + 1;
      while (left >= 0 && strcmp(arr[left], target) == 0)
      {
        count++;
        left--;
      }
      while (right < n && strcmp(arr[right], target) == 0)
      {
        count++;
        right++;
      }
      break;
    }
    if (cmp < 0)
    {
      low = mid + 1;
    }
    else
    {
      high = mid - 1;
    }
  }

  return count;
}
```

```
int* stringAnagram(int dictionary_count, char** dictionary, int query_count, char** query, int* result_count)
{
  char **new_query = (char **)calloc(query_count, sizeof(char *));

  for (int i = 0; i < query_count; i++)
  {
    int n = strlen(query[i]);
    new_query[i] = (char *)calloc(n + 1, sizeof(char));
    strcpy(new_query[i], query[i]);
    qsort(new_query[i], n, sizeof(char), compare_chars);                //---------- n log(n) ----------//
  }

  char **new_dec = (char **)calloc(dictionary_count, sizeof(char *));
  for (int i = 0; i < dictionary_count; i++)
  {
    int n = strlen(dictionary[i]);
    new_dec[i] = (char *)calloc(n + 1, sizeof(char));
    strcpy(new_dec[i], dictionary[i]);
    qsort(new_dec[i], n, sizeof(char), compare_chars);                //---------- n log(n) ----------//
  };

  qsort(new_dec, dictionary_count, sizeof(char *), compare);          //---------- n log(n) ----------//

  int *ans = (int *)calloc(query_count, sizeof(int));
  for (int i = 0; i < query_count; i++)
  {
    ans[i] = binarySearchString(new_dec, dictionary_count, new_query[i]);         //---------- log(n) ----------//
  }

  *result_count = query_count;

  return ans;
}
```

2. Subarray Sums

Subarray sums are often useful in finding the cumulative frequency of a given interval. In this problem, your goal is to find the subarray sums of the given array for the given queries.

You are given a 1-indexed array, *numbers*, of length *n*. The number of queries is given to you as *q*. Each query is defined by three integers: start index *l*, end index *r*, and a number *x*. For each query, find the sum of numbers between indexes *l* and *r* (both extremes included), and for each occurrence of zero within the range, add the value of *x* to the sum.

For example, let's say there are $n = 4$ numbers, where the array $numbers = [20, 30, 0, 10]$. Also, there is $q = 1$ query, the **start index is $l = 1$, and the end index is $r = 3$. For each occurrence of zero within the range, we'll add $x = 10$ to it. So, for this example, we are looking for the sum of the numbers between index 1 and index 3, which gives us 20 + 30 + 0 = 50. Because there is 1 zero in this range, we also add 10 to it, so the final answer is 50 + 10 = 60.**

Function Description

Complete the function *findSum* in the editor below.

findSum has the following parameter(s):

int *numbers[n]*: the array of integers, 1-indexed, that will be queried

int *queries[q][3]*: a 2-dimensional array of integers, 0-indexed, containing **start index *l*, end index *r*, and *x* for each query**

Returns:

long int *arr[q]*: the subarray sums of the given queries

Constraints

- $1 \le n \le 10^5$
- $1 \le q \le 10^5$
- $-10^9 \le numbers[i] \le 10^9$
- length of $queries[i] = 3$, for all i
- $1 \le queries[i][0] \le queries[i][1] \le n$
- $-10^9 \le queries[i][2] \le 10^9$

▼ Input Format for Custom Testing

The first line contains an integer, *n*, denoting the length of the array *numbers*.

Each line *i* of the *n* subsequent lines (where $1 \le i \le n$) contains an integer that describes *numbers[i]*.

Then the next line contains an integer, *q*, denoting the number of queries.

The next line contains an integer, 3, denoting the number of integers in each query.

Each line *i* of the *q* subsequent lines (where $0 \le i < q$) contains three space-separated integers**—start index *l*, end index *r*, and *x* —for *queries[i]*.**

▼ Sample Case 0

```
3
5
10
10
1
3
1 2 5
```

Sample Output

```
15
```

Explanation

Here, *numbers = [5, 10, 10]*. There is a single query for the range [1, 2]. The sum of the numbers in this range is 15. As there are no zeroes in the queried range, the answer remains 15.

▼ Sample Case 1

```
2
-5
0
2
3
2 2 20
1 2 10
```

Sample Output

```
20
5
```

Explanation

Here, *numbers = [-5, 0]*. There are 2 queries. The first query is for the range [2, 2], which equals a sum of 0. Because there is one zero in the queried range, *x = 20* is added to 0 to get the final answer of 20. The second query is for the range [1, 2], which equals a sum of -5. Because there is one zero in this queried range, *x = 10* is added to -5 to get the final answer of 5.

```c
#include <stdio.h>
long* findSum(int numbers_count, int* numbers, int queries_rows, int queries_columns, int** queries, int*
result_count)
{

  long *result = (long *)malloc(queries_rows * sizeof(long));
  long a[numbers_count + 1];
  long b[numbers_count + 1];
  a[0] = 0;
  b[0] = 0;

  for (int i = 0; i < numbers_count; i++)
  {
    a[i + 1] = a[i] + numbers[i];
    b[i + 1] = b[i] + (numbers[i] == 0);
  }

  for (int i = 0; i < queries_rows; i++)
  {
    int x = queries[i][2];
    int r = queries[i][1];
    int l = queries[i][0];

    result[i] = a[r] - a[l - 1] + x * (b[r] - b[l - 1]);
  }

  *result_count=queries_rows;
  return result;
}
```

1. Unexpected Demand

A widget manufacturer is facing unexpectedly high demand for its new product,. They would like to satisfy as many customers as possible. Given a number of widgets available and a list of customer orders, what is the maximum number of orders the manufacturer can fulfill in full?

Function Description
Complete the function *filledOrders* in the editor below. The function must return a single integer denoting the maximum possible number of fulfilled orders.

filledOrders has the following parameter(s):
order : an array of integers listing the orders
k : an integer denoting widgets available for shipment

Constraints

- $1 \leq n \leq 2 \times 10^5$
- $1 \leq order[i] \leq 10^9$
- $1 \leq k \leq 10^9$

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of orders.

Each line *i* of the *n* subsequent lines contains an integer *order[i]*.
Next line contains a single integer *k* denoting the widgets available.

▼ Sample Case 0

Sample Input For Custom Testing

```
2
10
30
40
```

Sample Output

```
2
```

Explanation
order = [10,30] with 40 widgets available. Both orders can be fulfilled.

▼ Sample Case 1

Sample Input For Custom Testing

```
3
5
4
6
3
```

Sample Output

```
0
```

Explanation
order = [5,4,6] with 3 widgets available
None of the orders can be fulfilled.

```c
#include <stdio.h>
int compare(int *a, int *b)
{
  return *a - *b;
}

int filledOrders(int order_count, int* order, int k)
{

  qsort(order, order_count, sizeof(int), compare);
  int filled;
  for (int i = 0; i < order_count; i++)
  {
    if (order[i] <= k)
    {
      filled++;
      k -= order[i];
    }
    else
    {
      break;
    }
  }
  return filled;
}
```

2. Usernames Changes

A company has released a new internal system, and each employee has been assigned a username. Employees are allowed to change their usernames but only in a limited way. More specifically, they can choose letters at two different positions and swap them. For example, the username "bigfish" can be changed to "gibfish" (swapping 'b' and 'g') or "bighisf" (swapping 'f' and 'h'). The manager would like to know which employees can update their usernames so that the new username is smaller in alphabetical order than the original username.

For each username given, return either "YES" or "NO" based on whether that username can be changed (with one swap) to a new one that is smaller in alphabetical order.

Note: For two different strings A and B of the same length, A is smaller than B in alphabetical order when on the first position where A and B differ, A has a smaller letter in alphabetical order than B has.

For example, let's say *usernames = ["bee", "superhero", "ace"]*. For the first username, "bee", it is not possible to make one swap to change it to a smaller one in alphabetical order, so the answer is "NO". For the second username, "superhero", it *is* possible get a new username that is smaller in alphabetical order (for example, by swapping letters 's' and 'h' to get "hupersero"), so the answer is "YES". Finally, for the last username "ace", it is not possible to make one swap to change it to a smaller one in alphabetical order, so the answer is "NO". Therefore you would return the array of strings ["NO", "YES", "NO"].

Function Description

Complete the function *possibleChanges* in the editor below.

possibleChanges has the following parameter(s):

string *usernames[n]*: an array of strings denoting the usernames of the employees

Returns:

string[n]: an array of strings containing either "YES" or "NO" based on whether the i^{th} username can be changed with one swap to a new one that is smaller in alphabetical order

Constraints

- $1 \leq n \leq 10^5$
- The sum of lengths of all usernames does not exceed 10^6.
- *usernames[i]* consists of only lowercase English letters.

▼ Input Format For Custom Testing

The first line of input contains an integer, *n*, denoting the number of employees.

Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains a string, *usernames[i]*, denoting the username of the i^{th} employee.

▼ Sample Case 0

Sample Input For Custom Testing

```
1
hydra
```

Sample Output

```
YES
```

Explanation

There is just one username to consider in this case, and it is the username "hydra". One can swap, for example, the last two letters of it to get the new username "hydar". This is smaller in alphabetical order than "hydra", so the answer for this username is "YES" (without quotes).

▼ Sample Case 1

Sample Input For Custom Testing

```
3
foo
bar
baz
```

Sample Output

```
NO
YES
YES
```

Explanation

There are three usernames to consider in this case. For the first of them, "foo", it is not possible to make one swap to change it to a smaller one in alphabetical order, so the answer is "NO" (without quotes). For the second one, "bar", one can swap the letters "b" and "a" to get the new username "abr", which is smaller in alphabetical order, so the answer is "YES" (without quotes). Similarly, For the third username, "baz", one can swap the letters "b" and "ab" to get the new username "abz", which is smaller in alphabetical order, so the answer is "YES" (without quotes).

```c
#include <stdio.h>
#include <string.h>

int compare_chars(const char *a, const char *b)
{

  return *a - *b;
}

char** possibleChanges(int usernames_count, char** usernames, int* result_count)
{

  char **result = (char *)malloc(usernames_count * sizeof(char *));

  for (int i = 0; i < usernames_count; i++)
  {
    char *temp = (char *)calloc(strlen(usernames[i]) + 1, sizeof(char));
    strcpy(temp, usernames[i]);
    qsort(temp, strlen(temp), sizeof(char), compare_chars);
    if (strcmp(usernames[i], temp))
    {
      result[i] = "YES";
    }
    else
    {
      result[i] = "NO";
    }
    free(temp);
  }

  *result_count=usernames_count;
  return result;
}
```

2. Vowel-Substring

Given a string of lowercase English letters and an integer of the substring length, determine the substring of that length that contains the most vowels. Vowels are in the set {a, e, i, o, u}. If there is more than one substring with the maximum number of vowels, return the one that starts at the lowest index. If there are no vowels in the input string, return the string *'Not found!'* without quotes.

Example 1

s = 'caberqiitefg'

k = 5

The substring of length *k = 5* that contains the maximum number of vowels is *'erqii'* with 3 vowels.

The final answer is *'erqii'*.

Example 2

s = 'aeiouia'

k = 3

All of the characters are vowels, so any substring of length 3 will have 3 vowels. The lowest index substring is at index 0, 'aei'.

Function Description

Complete the function *findSubstring* in the editor below.

findSubstring has the following parameters:

s: a string

k: an integer

Returns:

string: a string containing the final answer

Constraints

- $1 \le length(s) \le 2 * 10^5$
- $1 \le k \le length(s)$

▼ Input Format For Custom Testing

The first line contains a string, *s*.

The second line contains an integer, *k*.

▼ Sample Case 0

Sample Input

```
STDIN       Function
-----       -----
azerdii →   s = 'azerdii'
5       →   k = 5
```

Sample Output

```
erdii
```

Explanation

s = 'azerdii'

k = 5

The possible 5 character substrings are:

'azerd' which contains 2 vowels

'zerdi' which contains 2 vowels

'erdii' which contains 3 vowels

The final answer is *'erdii'*.

▼ Sample Case 1

Sample Input

```
STDIN       Function
-----       -----
qwdftr →    s = 'qwdftr'
2      →    k = 2
```

Sample Output

```
Not found!
```

Explanation

The given string does not contain any vowels, so *'Not Found!'* is returned.

```c
#include <stdio.h>
#include <string.h>

char* findSubstring(char* s, int k)
{
  char *result = (char *)malloc((k + 1) * sizeof(char));
  char vowels[] = "aeiou";
  int count = 0;
  int ans = 0;
  char *ptr;

  for (int i = 0; i < k; i++)
  {
    ptr = strchr(vowels, s[i]);
    count += (ptr != NULL);
  }
  int max = count;
  for (int i = k; i < strlen(s); i++)
  {
    ptr = strchr(vowels, s[i]);
    count += (ptr != NULL);
    ptr = strchr(vowels, s[i - k]);
    count -= (ptr != NULL);

    if (count > max)
    {
      max = count;
      ans = i - k + 1;
    }
  }

  if (max > 0)
  {
    strncpy(result, s + ans, k);
    result[k] = '\0';
    return result;
  }

  free(result);
  return "Not found!";
}
```

```c
#include <stdio.h>
#include <string.h>

char* findSubstring(char* s, int k)
{
  char *result = (char *)malloc((k + 1) * sizeof(char));
  int key[256] = {};
  char vowels[] = "aeiou";
  for (int i = 0; i < strlen(vowels); i++)
  {
    key[vowels[i]] = 1;
  }

  int count = 0;
  int ans = 0;

  for (int i = 0; i < k; i++)
  {
    count += key[s[i]];
  }
  int max = count;
  for (int i = k; i < strlen(s); i++)
  {
    count += key[s[i]];
    count -= key[s[i - k]];

    if (count > max)
    {
      max = count;
      ans = i - k + 1;
    }
  }

  if (max > 0)
  {
    strncpy(result, s + ans, k);
    result[k] = '\0';
    return result;
  }

  free(result);
  return "Not found!";
}
```