CS213M: Assignment 3

General Instructions and Tips:

- For each problem, test cases (p<x>_t<y>.txt) will be provided in the test folder. Note that your code will be tested on hidden test cases on submission.
- Use the following command to compile p1.cpp: `g++ p1.cpp`. An executable a.out is created. To run the executable, use the command `./a.out`
- USE 'cin' and 'cout' for taking input and printing respectively. To take input from a file abc.txt, use `./a.out < abc.txt`
- Students are expected to adhere to the highest standards of integrity and academic honesty. Acts such as copying in the examinations and sharing code for the programming assignments will be in dealt with strictly, in accordance with the institute's procedures and disciplinary actions for academic malpractice.
- You can use well known algorithms, provided you write the code yourselves, such cases will be handled subjectively.
- We will be using the g++ compiler for compiling the code

P1 : Polynomial Addition

Build a class to represent "sparse" polynomials (think about it!) and use it to add two polynomials. You are expected to build the necessary member functions too.

<u>Input :</u>
No. of terms in polynomial 1 having non-zero coefficients : n_1
Space separated coefficients of polynomial 1
Space separated powers having the corresponding coefficients of polynomial 1
No. of terms in polynomial 2 : n_2
Space separated coefficients of polynomial 2
Space separated powers having the corresponding coefficients of polynomial 2

<u>Output:</u>
Space separated coefficients of the multiplied polynomial
Space separated powers having the corresponding coefficients of the multiplied polynomial

<u>Constraints:</u>
$n_1 < 10^3$
$n_2 < 10^3$
Integral Coefficients $< 10^6$

<u>Examples:</u>
Worked out example :
Input:
A[] = {5, 0, 10, 6}
B[] = {1, 0, 0, 0, 4}
Output:
add[] = {6,0,10,6,4}

The first input array represents => $5 + 10x^2 + 6x^3$
The second array represents => $1 + 4x^4$
And Output is => $6 + 10x^2 + 6x^3 + 4x^4$

Input	Output

3
5 10 6
0 2 3
2
1 4
0 4
	6 10 6 4
	0 2 3 4

<u>File to be submitted : p1.cpp</u>

P2: Merge Sort and Quick sort

Having learned the problem of sorting in the context of the divide and conquer paradigm, you have to implement the Merge Sort and Quick sort algorithm to sort arrays in files p2-ms.cpp and p2-qs.cpp respectively.

Input Format:

The first line contains a single integer **n**, the length of the array.
The next line contains **n** integers.

Output:

Sorted array space separated

Examples:

Input	**Output**
6 3 4 2 1 5 6	1 2 3 4 5 6
6 3 3 4 1 5 7	1 3 3 4 5 7

File to be submitted : p2-ms.cpp and p2-qs.cpp

P3: Longest Prefix Common in a list of strings

Given a list/array of strings, find the longest common prefix among the list of strings. You need to assume case sensitivity, i.e. ‘L’ != ‘l’.

Expected complexity: O(n*log(n)), where n is the size of the list/array.

Input:
The first line contains a single integer **n**, the length of the list.
Next **n** lines : ‘**s**’, s.t. |**s**| < 100

Output :
Longest common prefix, or **-1** in case there are no common prefixes

Constraints :

$n < 10^5$
$|s| < 100$

Examples :

Input	**Output**
3 hello hell holy	h
2 great big	-1
4 Man Mango Mankind Mandatory	Man
2 Bat bat	-1

File to be submitted : p3.cpp

P4: Playing with array search

Consider that you are communicating with a backend server using an http connection. The backend server has an array of Strings, **‘arr’** of length N. Each element of the array is a lowercase string **s** such that |**s**| <= 20, i.e. length of the string is <= 20. This array is sorted according to the lexicographical order i.e. dictionary order. *(Read about it if you are not aware: Lexicographical Order Wiki)*. You are allowed to communicate with the backend in a very simple way : You send an index i s.t. (i < N), and the backend returns the string at index i, i.e. arr[i]. This communication or ‘access’ takes a certain time due to the network delays.

You have to search a given string in the array, i.e. output the index of the string in the array else **-1** if it is not present. Assume 0-based indexing.

There are two types of accesses --

1. **Good Access** : An access of index i, i.e. arr[i] is a good access if arr[i] < given string.
2. **Bad Access** : An access of index i, i.e. arr[i] is a bad access if arr[i] >= given string.

Here $s_1 < s_2$ if s_1 occurs lexicographically before s_2 (If s_1 is a prefix of s_2 , then $s_1 < s_2$)

It turns out that the backend code written has some bugs in it due to which it can handle only at most **k** bad accesses. At the **(k+1)th** bad access the backend code will crash and you should have known your answer by then. i.e. at max you can do **(k+1)** bad accesses. You are to minimize the number of good accesses (thereby time) done by your search algorithm while keeping the bad accesses at max **k+1**. Keep in mind that this minimization is to take into account the different possibilities of the given strings positions in the array. i.e. it should not be the case that your algorithm works for certain cases really well and for certain cases it performs very poorly.

- For k = 0 : the answer is a simple linear search starting from the start of the array. If you encounter the given string **a**, output the index else keep on moving till you encounter the first element **s** s.t. **s** > **a**, in which case it is a bad query and hence the backend will crash. At this point you know the answer that the given string is not present in the array. And hence you can output **-1**. This is the best guarantee we can give, hence the number of accesses would be O(n).

- For k >= log(**N**) + 1 : The answer is a O(log(**N**)). This can be done using a binary search algorithm.

The task of this assignment is to do the search for k = 1, i.e. only one bad access allowed by the backend server.

Coding it up :

For the purpose of the assignment you have to model the backend server as an array and maintain a counter of the number of accesses that you are doing of an array. Thus, counter * Delay would model the actual delay of your algorithm. Essentially every single time you access an element, i.e. write arr[i] in your code you need to increment the counter by 1. Also you need to ensure that the number of bad queries in your algorithm is <= **k + 1**.

Input Format:

N : number of elements in the array
Next N lines : elements of the array, strings of size <= 20.
Last line : given string to search, str

Output Format:

1st line contains the answer : index i of given search string, -1 is not present.
Next line contains the number of good accesses made by your algorithm.

(Extra : It would be a good idea to test your code by running it on different instances searching for different strings for a given array and also for strings that are not present. You can even see the algorithm complexity by plotting the accesses made against the inputs of different sizes. **Do not submit these graphs**)

Grading :

You will be given 3 marks for writing a code with correct implementation, i.e. the first line of output matching the correct solution. Also your number of accesses would be compared with the best solution and based on how close your solution is you will be given partial marks for the remaining 7 marks. So it is highly recommended to explore and try out different solutions.

Examples:

For a naive linear Search implementation :

Input	Output
5 apple banana grapes mango orange guava	-1 5
5 apple banana grapes mango orange	3 4

mango	

Files to be submitted : p4.cpp

Bonus Question : Do the same for k = 2. Submit another file p4-bonus.cpp. (Hint O($n^{1/3}$) solution can be achieved) (5 marks)

P5: (Bonus Question) Electric Force Calculation :

Consider **n** point charges located at points on the x axis (1D) at coordinates 0, 1, 2, … , n - 1. The charge at point **i** has value q_i, which may be positive or negative. The net force on charge **i** due to all other charges is defined as :

$$\sum_{i<j} (q_i * q_j) / [(i-j)^2] - \sum_{j<i} (q_i * q_j) / [(j-i)^2]$$

Write an O(n*log(n)) algorithm to calculate the force on each charged particle calculated using the above formula.

Input :

n : number of particles

q_i where $0 <= i < n$

Output :

n space separated integers, i.e. force on particle at position **i** rounded to the nearest integer

Constraints :

$0 < n < 10^5$

$|q_i| < 10^6$

Hint: Is the problem a bit ***convoluted*** ? Maybe that's why it is a bonus

Examples :

Input	Output
2 1 -1	1 -1
1 100	0

File to be submitted : p5.cpp