Assignment-5

OJ

Question 1:

During the Navratri time, when Arshvardhan returned home, he met his old crush, Emi. Emi told Arsh that she knows about a good place where free coffee is offered in return for solving a question. After they went there, the Manager gave them a string S and an integer K. He asked them to calculate the h-factor of the strings.

h-Factor of a string is defined as the maximum number of non-overlapping substrings each of length K such that there is exactly one distinct character across all the substrings picked. For eg:-(ab)(ab) non overlapping substrings of the string “abcdab” has 2 distinct character a & b. Similarly , (aa)(aa)(aa) substrings of “aabcaagfaaj” has 1 distinct character. Given the string S and the integer K , output its h-factor .

Formally, Output the maximum number of single-valued non-intersecting substrings of length K in the string S.

> 💡 A substring is defined as a sequence of contiguous characters obtained by deleting one or more (or all) characters from the beginning of the string and one or more (or all) characters from the end of the string.

Input Constraints:

- *N* is the length of the string S (N ≤ 2 X 10^5)
- *S* is a string of length N consisting of lowercase English letters
- K is an integer (1 ≤ K ≤ N)

Input:

```
N K
S
```

Output:

Integer (answering h-factor(S))

Sample Input 1:

```
8 2
aaacaabb
```

Sample Output 1:

```
2
```

Sample Input 2:

```
2 1
ab
```

Sample Output 2:

```
1
```

Sample Input 3:

```
4 2
abab
```

Sample Output 3:

```
0
```

Note:

💡 - In the first example, we can select 2 non-intersecting/non-overlapping substrings consisting of letter 'a': "(aa)ac(aa)bb", so the H-Factor is 2.

- In the second example, we can select either substring "a" or "b" to get the answer 1

- In the third example, we can't find any single-valued substrings (substring consisting of a single distinct character) of length 2, so the answer is 0

Question 2:

Kitansh Mayathwal recently got a pet dog and asked you to help him name it. You want to suggest a good name but do not have time as you have to finish your CPro assignment, so you ask your friend for help. Your friend gives you an initial string S and asks you to perform M actions on the string.
In the i^{th} action, your friend will say Q_i, a number either 1 or 2 and will order you the following.
If he says the number 1 (Q_i = 1) - You are ordered to reverse the current string
If he says the number 2 (Q_i = 2) - He will give you one more number, L_i, a number either 1 or 2 and a character J.

- If he now said the number 1 (L_i= 1), then you are ordered to add J to the start of the string.
- If he now said the number 2 (L_i = 2), you are ordered to add J to the end of the string.

What is the final string after completing all actions?

Input Constraints:

- The length of the string S is ≤ 10^5

- *S* is a string of length N consisting of lowercase English letters
- M is the number of actions (M ≤ 2x 10^5)
- Q_i is either 1 or 2.
- L_i is either 1 or 2
- J is a lowercase English letter

Input:

```
S
M
Q_1
…
Q_M
```

Where `Q_i` is the `i^th` query which is either

`1`

(This implies Q_i = 1)

`2 L_i J`

(This implies Q_i = 2)

Output:

The Final String

Sample Input 1:

```
ab
4
1
2 1 c
2 2 d
1
```

Sample Output 1:

```
dabc
```

Sample Input 2:

```
a
3
1
2 2 z
1
```

Sample Output 1:

```
za
```

Sample Input 1:

Note:

> 💡 In the first example, the initial string is `ab`. After the first operation, the string becomes `ba`. After second operation string becomes `cba`. After third operation string becomes `cbad`. After fourth operation string becomes `dabc`.
>
> In the second example, the initial string is `a`. After the first operation, the string remains `a`. After second operation string becomes `az`. After third operation string becomes `za`.

Non-OJ

> To be submitted on Moodle

A . Matrix Operations

You are tasked with implementing matrix operations in C, organized into separate files. Your code will consist of three files: `matrix.h`, `matrix.c`, and `main.c`.

- `matrix.h`: Create a header file named `matrix.h`, containing the function prototypes for various matrix operations.
- `matrix.c`: Implement the functions defined in `matrix.h`.
- `main.c`: Read the input, perform the operation (*addition, multiplication, scalar multiplication, determinant, transpose, etc)* and output it as asked.

A. Write code to perform the following operations with these function names:

1. destroy `destroy_matrix`
2. add `add_matrix`
3. multiply `mult_matrix`
4. scalar multiply `scalar_mult_matrix`
5. transpose `transpose_matrix`
6. determinant `determinant` (This link can help)

You can refer to the homework code on the website: [Commandline Args and Multifile Programming Lecture Homework]

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

typedef struct Matrix {
    int        num_rows;
    int        num_cols;
    long long int**     data;
} Matrix;


Matrix* create_matrix(int r, int c) {
    Matrix* m = (Matrix*) malloc(sizeof(Matrix));
    m->num_rows = r;
    m->num_cols = c;
    m->data = (long long int**) calloc(r, sizeof(long long int*));
    for (int i =0; i < r; i++) {
        m->data[i] = (long long int*) calloc(c, sizeof(long long int));
    }
    return m;
}

void destroy_matrix(Matrix* m) {
    // 1: Write code here to free all memory used by the matrix stored in m
}

Matrix* add_matrix(Matrix* A, Matrix* B) {
   // 2: write code here to add the matrices A and B and return a new matrix with the results.
   // A, B should remain unmodified. If dimensions don't match, it should return NULL.
}

Matrix* mult_matrix(Matrix* A, Matrix* B) {
   // 3: write code here to multiply the matrices A and B and return a new matrix with the results.
   // A, B should remain unmodified. If the dimensions don't match, it should return NULL.
}

Matrix* scalar_mult_matrix(long long int s, Matrix* M) {
    // 4: write code here to multiply the matrix A with a scalar s and return a new matrix with the results.
   // M should remain unmodified.
}

Matrix* transpose_matrix(Matrix* A) {
   // 5: write code here to find the transpose of given matrix A and return a new matrix with the results.
   // A should remain unmodified.
}

long long int determinant(Matrix* M) {
    // 6: Write code here to calculate the determinant of the given matrix M (if not a square matrix, return -1).
    // Return the determinant value.
}

// DO NOT MODIFY THE OUTPUT FORMAT of this function. Will be used for grading
void print_matrix(Matrix* m) {
    printf("%d %d\n", m->num_rows, m->num_cols);
    for (int i = 0; i < m->num_rows; i++) {
        for (int j = 0; j < m->num_cols; j++) {
            printf("%lld ", m->data[i][j]);
        }
        printf("\n");
    }
}

int main() {
    /**
      Matrix* m = create_matrix(3,3);
      print_matrix(m);
    */
    return 0;
```

```
}
```

Input Format:

The first line of input would contain a single integer `q`, denoting the number of queries. Then, `q` queries follow. Each query could be one of the following.

In each of the following queries, a matrix would be specified in the following way:

The first line would contain two integers `n m` denoting the dimensions of the matrix (having `n` rows and `m` columns). Then, `n` lines follow:

The `i^th` line describes the `i^th` row of the matrix. It would contain `m` space-separated integers where the `j^th` integer describes the element in the `j^th` column.

In other words, the format is given by:

```
n m
A_11 A_12 A_13 ... A_1m
A_21 ...
...
A_n1 A_n2 A_n3 ... A_nm
```

We shall refer to this format as `<matrix>` in the following descriptions.

Add matrices:

The first line of the query would contain the string `add_matrix` and the integer `0` (space-separated)

> 💡 The `0` here is put in place to make the queries from part B cleaner.

Next, the first operand is specified by the above format, followed by the second in the same format.

Formally, it would look like:

```
add_matrix 0
<matrix>
<matrix>
```

Multiply matrices

The format is similar to the above command, except the first line now reads `mult_matrix 0`

```
mult_matrix 0
<matrix>
<matrix>
```

Scalar Multiply with matrix

The first line of the query reads `scalar_mult_matrix 0`

The second line would contain a single integer `s` that denotes the scalar to be multiplied with the matrix.

Next, the matrix is described in the above-mentioned format.

```
scalar_mult_matrix 0
s
<matrix>
```

Transpose matrix

The first line of the query reads `transpose_matrix 0`

Next, the matrix is described in the above format.

```
transpose_matrix 0
<matrix>
```

Determinant

The first line of the query reads `determinant 0`

Next, the matrix is described in the above format.

```
determinant 0
<matrix>
```

Output Format

For each of the above queries (except the determinant), the code needs to output the resultant matrix using the `print_matrix` function **already provided**. **For the determinant query, the code needs to output the resulting integer output on a single line**.

Handling Errors

In case of an argument mismatch (for example, you were asked to add two matrices of different dimensions), the code needs to output a single line for the query saying `ERROR: INVALID ARGUMENT`

Example Run

Input:

```
4
add_matrix 0
2 3
1 2 3
4 5 6
2 3
1 1 1
1 1 1
add_matrix 0
2 2
2 5
3 5
1 1
3
transpose_matrix 0
3 2
3 5
6 1
3 5
determinant 0
```

```
1 3
3 5 1
```

Output:

```
2 3
2 3 4
5 6 7
ERROR: INVALID ARGUMENT
2 3
3 6 3
5 1 5
ERROR: INVALID ARGUMENT
```

B. Matrix File Handling

Implement two more functions `read_matrix_from_file` and `write_matrix_to_file`

Read Matrix From File:

The function should take a file name as an argument and read the matrix present in the file. If the file is present, then it would contain the matrix in the format specified above. If not, print the error `ERROR: INVALID ARGUMENT` similar to part A.

Write Matrix To File:

This function should take a file name and a matrix as its argument and write the given matrix in the specified file. If there are issues while opening the file, then it should print the error message specified earlier. If a file with the name does exist, overwrite its contents. The matrix should be written to the file in the format specified in part A (which is also followed by the `print_matrix` command)

Input format

The queries are similar to the ones specified in part A, except here, you would read the input matrix/matrices from files and write the output to a file.

For each of the above queries, the first line would now end with the integer `1` (instead of the previous `0`). Further, when describing matrices as `<matrix>` we will now say `<file_name_x>` indicating the file name to be used for reading the corresponding matrix. Additionally, the queries in this section would have an additional line at the end which specifies the output file name. It is guaranteed that all of the file names would have only alpha-numeric characters without any white-spaces in them.

Add matrices:

```
add_matrix 1
<file_name_1>
<file_name_2>
<output_file_name>
```

Multiply matrices

```
mult_matrix 1
<file_name_1>
<file_name_2>
<output_file_name>
```

Scalar Multiply with matrix

```
scalar_mult_matrix 1
s
<file_name_1>
<output_name_1>
```

Transpose matrix

```
transpose_matrix 1
<file_name_1>
<output_file_name>
```

Determinant

```
determinant 1
<file_name_1>
```

Output Format

For each of the above queries (except the determinant), the code needs to output the resultant matrix using the `write_matrix_to_file` function, which is to be implemented. **For the determinant query, the code needs to output the resulting integer output on a single line to the terminal and not to an output file**.

Handling Errors

In case of an argument mismatch (for example, you were asked to add two matrices of different dimensions), the code needs to output a single line for the query saying `ERROR: INVALID ARGUMENT`. Additionally, if there are any errors related to file opening/closing/reading/writing output the same error message. While writing to files, if the file already exists, overwrite it. This should not counted as an error.

Example Run

Input:

```
4
add_matrix 1
helloWorldMatrix1
worldHelloMatrix2
out1
add_matrix 1
fileContainingA2By3Matrix
fileContainingA1By2Matrix
out2
add_matrix 0
2 3
1 2 3
4 5 6
2 3
1 1 1
1 1 1
determinant 1
byeMatrix
```

Output:

```
//query 1 writes the result to file 'out1'
//query 2 failed say, because the matrices were incompatible
ERROR: INVALID ARGUMENT
2 3
2 3 4
5 6 7
//The determinant output is written to the terminal
25
```

C. History Command

This command is similar to the history command in your terminal. Keep track of all the operations that were done in part A and part B. And provide a new command `history`, which should print all the operations done so far. This should work across different runs of your program. For example, if you do the operations `add_matrix 0`, `mult_matrix 1` in one program run. Close the program. Run the program again and do one more `add_matrix 0`; then run history, it should show the following:

```
LOG::add_matrix 0
LOG::mult_matrix 1
LOG::add_matrix 0
LOG::history
```

Note that the `history` command counts as an operation, so you should also store it. To achieve this, you should write the operations done so far in a file. Name it `mx_history` (do not have any extensions for this file).

The lines written to the file should be of the format `LOG::` followed by the first line of the query.

The following describes the output of two sample runs:

```
> ./main
3
add_matrix 0
...
...
mult_matrix 1
...
...
history
LOG::add_matrix 0
LOG::mult_matrix 1
LOG::history

//Ctrl+C to end the program

> ./main
2
transpose 1
...
...
history
LOG::add_matrix 0
LOG::mult_matrix 1
LOG::history
LOG::transpose 1
LOG::history
```

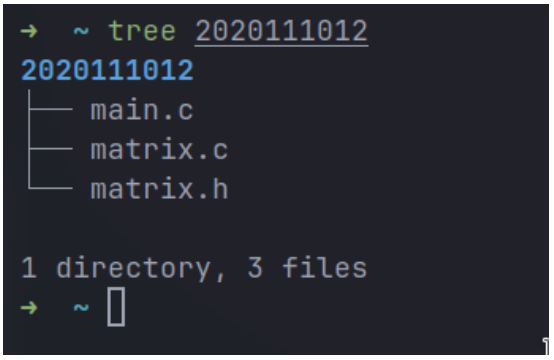
Note that the history file should not store the matrices specified in the output and instead store only the first line of the query. Each of the following commands is eligible to be written to the history file and should be

output when the `history` command is called.

- `add_matrix 0` `add_matrix 1`
- `mult_matrix 0` `mult_matrix 1`
- `scalar_mult_matrix 0` `scalar_mult_matrix 1`
- `transpose_matrix 0` `transpose_matrix 1`
- `determinant 0` `determinant 1`
- `history`

Submission details

Have the implementations structured in `main.c`, `matrix.c` and `matrix.h`, as explained at the beginning of the section. Keep the files in a single directory with your **roll number as the name** and zip the directory. Submit only the **zip file** (note that the extensions `.tar`, `.tar.gz` etc would not be accepted)



The program will be compiled as `gcc rollnumber/main.c rollnumber/matrix.c -o main`. Ensure there are no other files in your submission (especially those with `.c` or `.h` extensions). **Strictly** follow the I/O format and the file structure specified above.