[CENG 315 ALL Sections] Algorithms

Navigation

- Dashboard
  - Site home
  - Site pages
  - My courses
    - 571 - Computer Engineering
      - CENG 223 All Sections
      - CENG 223 Section 2
      - CENG 315 ALL Sections
        - Participants
        - Competencies
        - Grades
        - General
        - October 2 - October 8
        - October 9 - October 15
        - October 15 - October 22
        - October 23 - October 29
        - October 30 - November 5
        - November 6 - November 12
        - November 13 - November 19
          - THE3
            - Description
            - Submission view
        - November 20 - November 26
        - November 27 - December 3
        - December 3 - December 10
        - December 11 - December 17
        - December 18 - December 24
        - December 25 - December 31
        - January 1 - January 7
        - January 8 - January 14
      - CENG 315 Section 1

Description | Submission view

THE3

Available from: Saturday, November 18, 2023, 12:00 PM
Due date: Sunday, November 19, 2023, 11:59 PM
Requested files: the3.cpp, test.cpp (Download)
Type of work: Individual work

Your archeologist friend found a scroll during her latest excavation, but unfortunately, the writing on the scroll is corrupted. She believes that the writing on the scroll may be a duplicate of an existing text. You offer to help her by writing a computer program that compares strings and tries to align them according to their similarities. You ask her for a scoring system for character comparison, and she gives you the following function:

```
alignment score = (# of matching characters * match score)  - (# of gaps * gap score) - (# of mismatching characters * mismatch score)
```

For example, if the readable sequence on the scroll is "CC", the possible match string is "CD", the match score is 4, the gap score is 1, and the mismatch score is 2, then the possible alignments between the strings and their scores are as follows:

(The character "_" denotes a gap in the first string, "." denotes a gap in the second string, and "!" denotes a mismatch.)

```
CC__
..CD
aligned:
..__ => alignment does not have any matches or mismatches, only gaps
gap, gap, gap, gap => (0 * 4)-(4 * 1)- (0 * 2) = 0 - 4 - 0 = -4

CC_
.CD
aligned:
.C_  => alignment has two gaps, one match, no mismatch
gap, match, gap => (1 * 4) - (2 * 1) - (0 * 2) = 4 - 2 - 0 = 2

CC
CD
aligned:
C!   => alignment has one match and one mismatch
match, mismatch => (1 * 4) - (0 * 1) - (1 * 2) = 4 - 0 - 2 = 2

_CC
CD.
aligned:
_!.  => alignment has one mismatch and two gaps
gap, mismatch, gap => (0 * 4) - (2 * 1) - (1 * 2) = 0 - 2 - 2 = -4

__CC
CD..
aligned:
__.. => alignment does not have any matches or mismatches, only gaps
gap, gap, gap, gap => (0 * 4) - (4 * 1) - (0 * 2) = 0 - 4 - 0 = -4
```

Problem

In this exam, you are asked to compare the two given sequences and find one possible alignment with the highest match score with two different approaches by completing the ***recursive_alignment()*** and ***dp_table_alignment()*** functions defined below.

```
int recursive_alignment(std::string sequence_A, std::string sequence_B, int gap, int mismatch, int match, std::string &possible_alignment, int call_count);

int dp_table_alignment(std::string sequence_A, std::string sequence_B, int gap, int mismatch, int match, std::string &possible_alignment);
```

You should **return the highest alignment score of the two sequences** as the return value for both functions. The arguments of the function are defined as follows with their limits:

- ***sequence_A*** and ***sequence_B*** are strings to be compared, with lengths <= 10000
- ***gap*** is the integer penalty point for leaving a gap in the alignment, where 0 <= ***gap*** <= 10
- ***mismatch*** is the integer penalty point for a mismatch in the alignment, where 0 <= ***mismatch*** <= 10
- ***match*** is the integer point for a successful match in the alignment, where 0 <= ***match*** <= 10
- ***possible_alignment*** is a string that includes one of the possible alignments for two sequences with the highest alignment score, where ***the character "_" denotes a gap in sequence_A, "." denotes a gap in sequence_B, and "!" denotes a mismatch.***
- ***call_count*** is an integer used to keep track of the number of recursive calls.

Let **F(i,j)** denote the best alignment score when looking at index **i** of ***sequence_A*** and index **j** of ***sequence_B***. Then, the recurrence relation of this problem can be defined as follows:

```
F(i,j) = max(F(i-1, j-1) + c(i,j),
             F(i-1, j) - gap,
             F(i, j-1) - gap)
```

where **c(i,j)** stands for the comparison between the characters ***sequence_A[i]*** and ***sequence_B[j]***, and returns ***match*** or ***-mismatch*** accordingly.

Constraints and Hints:

- For the ***recursive_alignment()*** function you are expected to implement the given recurrence relation directly. Since this will be an inefficient solution, you need to **keep track of the recursive calls**. You should increment the ***call_count*** variable by 1 every time you make a recursive call, and **stop the calculation if the *call_count* reaches 1000000**. This means that you will not be able to find the optimal alignment for longer strings. **If you terminate the calculation because of the *call_count* limit, you should update the *possible_alignment* string as "*STACK LIMIT REACHED*". In that case, the returned alignment score is not important, but you can return INT_MIN to keep track** (<climits> is included in the3.h for you)**.**
- For the ***dp_table_alignment()*** function, you are expected to implement a bottom-up dynamic programming solution. When implemented correctly with a dynamic programming table, this solution should work with longer strings.
- When returning/reconstructing the alignment, it is **not** important which alignment you return as ***possible_alignment*** as long as it has the highest alignment score (i.e. if there are multiple solutions, you can return **any** of them).
- It is guaranteed that **length(*sequence_A*) <= length(*sequence_B*).**
- ***sequence_A*** and ***sequence_B*** will consist of **uppercase English characters [A-Z]**.

Evaluation:

After your exam, black-box evaluation will be carried out. Grading is as follows:

- **30 points** from the ***recursive_alignment()*** function if both the return value is the correct highest alignment score, and the possible_alignment string is a valid solution (an alignment that has the highest alignment score),
- **40 points** from the ***dp_table_alignment()*** function if the **return value** is the correct highest alignment score,
- **30 points** from the ***dp_table_alignment()*** function if the ***possible_alignment*** string is a valid solution (an alignment that has the highest alignment score).

Your program will be tested with additional inputs in the final evaluation phase. Please make sure that your program works correctly in longer inputs to guarantee a full grade.

Example IO:

1)

- ***sequence_A:*** "ACD"
- ***sequence_B:*** "ABCD"
- ***gap:*** 1
- ***mismatch:*** 2
- ***match:*** 4
  for both ***recursive_alignment***() and ***dp_table_alignment***():
- **possible_alignment:** "A_CD"
- **highest *alignment* score (return value):** 11

2)

- ***sequence_A:*** "THY"
- ***sequence_B:*** "BETH"
- ***gap***: 1
- ***mismatch***: 2
- ***match***: 4
  for both ***recursive_alignment***() and ***dp_table_alignment***():
- ***possible_alignment***: "__TH."
- ***highest alignment score (return value)***: 5

3)

- ***sequence_A:*** "AAAAAAAAAABBBBABABB"
- ***sequence_B***: "AAAAAACCAAACCABBBBABABB"
- ***gap***: 1
- ***mismatch***: 2
- ***match***: 4
  for ***recursive_alignment***():
- ***possible_alignment***: "STACK LIMIT REACHED"
- ***highest alignment score (return value)***: INT_MIN (not important)
  for ***dp_table_alignment***():
- ***possible_alignment***: "AAAAAA__AAA__ABBBBABABB"
- ***highest alignment score (return value)***: 72

4)

- ***sequence_A:*** "ELPROBLEMAESELCAPITALISMO"
- ***sequence_B***: "THEREISNOETHICALCONSUMPTIONUNDERCAPITALISM"
- ***gap***: 0
- ***mismatch***: 2
- ***match***: 10
  for ***recursive_alignment***():
- ***possible_alignment***: "STACK LIMIT REACHED"
- ***highest alignment score (return value)***: INT_MIN (not important)
  for ***dp_table_alignment***():
- ***possible_alignment***: "__E..R___O..E___.A___.S_____E_.CAPITALISM."
- ***highest alignment score (return value)***: 170

Specifications:

- There are 2 tasks to be solved in **36 hours** in this take-home exam, with partial grading.
- You will implement your solutions in ***the3.cpp*** file.
- You are free to add other functions to *the3.cpp*
- **Do not change** the first line of the3.cpp, which is #include "the3.h"
- Some libraries are included in "the3.h" for your convenience, you can use them freely.
- **Do not change** the arguments and the return value of the functions ***recursive_alignment()*** and ***dp_table_alignment()*** in the file the3.cpp
- **Do not include** any other library or write include anywhere in your the3.cpp file (not even in comments).
- You are given ***test.cpp*** file to test your work on **ODTUClass** or your **locale**. You can, and you are, encouraged to modify this file to add different test cases.
- If you want to test your work and see your outputs you can compile your work on your locale as:

```
>g++ test.cpp the3.cpp -Wall -std=c++11 -o test

> ./test
```

- You can test your the3.cpp on the virtual lab environment. If you click **run**, your function will be compiled and **executed with test.cpp**. If you click **evaluate**, you will get **feedback** for your current work and your work will be **temporarily graded** for a limited number of inputs.
- The grade you see in lab is not your final grade, **your code will be reevaluated with different inputs** after the exam.

The system has the following limits:
(EDIT: the limits were set like this from the beginning and shown in the upper side of the VPL activity, but this place had some errors)

- a maximum execution time of 120 seconds
- a 1 GB maximum memory limit
- an execution file size of 4MB.
- Solutions with longer running times will not be graded.
- If you are sure that your solution works in the expected complexity, but your evaluation fails due to limits in the lab environment, the constant factors may be the problem.

Requested files

the3.cpp

```cpp
#include "the3.h"

// do not add extra libraries, but you can define helper functions below.




/*
PART 1
you are expected to call recursive_alignment (as the name suggests) recursively to find an alignment.
initial call_count value given to you will be 0.
you should check if call_count >= 1000000, if so, set possible_alignment string to "STACK LIMIT REACHED", return INT_MIN (or anything - it will not be check
*/
int recursive_alignment(std::string sequence_A, std::string sequence_B, int gap, int mismatch, int match, std::string &possible_alignment, int call_count){
    int highest_alignment_score;



    return highest_alignment_score;
}

/*
PART 2
you are expected to create a dynamic programming table to find the highest alignment score.
then you will need to reconstruct a possible alignment string from the table.
*/
int dp_table_alignment(std::string sequence_A, std::string sequence_B, int gap, int mismatch, int match, std::string &possible_alignment){
    int highest_alignment_score;


    return highest_alignment_score;
}

```

test.cpp

```cpp
// this file is for you for testing purposes, it will not be included in evaluation.

#include <iostream>
#include <fstream>
#include "the3.h"

void file_input(std::string& sequence_A, std::string& sequence_B, int& gap, int& mismatch, int& match){
    std::string file_name = "inp05.txt"; // inp01-inp10 are available.
    std::ifstream infile (file_name);
    if(!infile.is_open()){
        std::cout << "Input file cannot be opened" << std::endl;
        std::cout << "File name: " << file_name << std::endl;
        return;
    }
    infile >> sequence_A;
    infile >> sequence_B;
    infile >> gap;
    infile >> mismatch;
    infile >> match;
    return;
}

void test(){
    std::string sequence_A;
    std::string sequence_B;
    int gap, mismatch, match, highest_alignment_score_p1, highest_alignment_score_p2;
    std::string  possible_alignment_p1;
    std::string  possible_alignment_p2;
    int call_count = 0;

    file_input(sequence_A,sequence_B,gap,mismatch,match);
    std::cout << "Sequence A: " << sequence_A << std::endl <<
                 "Sequence B: " << sequence_B << std::endl <<
                 "gap: " << gap << std::endl <<
                 "mismatch: " << mismatch << std::endl <<
                 "match: " << match << std::endl;
```

```
    std::cout << "PART 1:" << std::endl;
    highest_alignment_score_p1 = recursive_alignment(sequence_A,sequence_B,gap,mismatch,match,possible_alignment_p1,call_count);
    std::cout << "Highest match score: " << highest_alignment_score_p1 << std::endl <<
                 "Possible alignment: " << std::endl;
    std::cout << possible_alignment_p1 << std::endl;

    std::cout << "PART 2:" << std::endl;
    highest_alignment_score_p2 = dp_table_alignment(sequence_A,sequence_B,gap,mismatch,match,possible_alignment_p2);
    std::cout << "Highest match score: " << highest_alignment_score_p2 << std::endl <<
                 "Possible alignment: " << std::endl;
    std::cout << possible_alignment_p2 << std::endl;
    return;
}

int main(){
    test();
    return 0;
}

```

VPL