CS6013: Advanced Data Structures and Algorithms IIT Hyderabad, Fall 2019

Programming Assignment 3 (DNA sequence alignment)

- Out on: **04 Nov 2019**. Due Date: **14th Nov 2019, 11:59:59 pm**.
- The total marks for this assignment is 50, split across three questions. You may refer to internet sources like Wikipedia for clarifications (since the problems are well-known) but the written explanation in the report and implementation in C++ code should be your own.
- For submission and upload instructions, see the section after the questions.
- As before, standard rules regarding plagiarism apply. If in doubt, ask before assuming.

Problem Description: Global DNA sequence alignment

In class, we had discussed the RNA sequence alignment problem, where a single strand of RNA pairs with itself. Here, we will look at DNA sequences. In the *global DNA sequence alignment* problem, you are given as input *two* DNA sequences S_1, S_2, which are strings using the four characters `A, C, T, G`. The strings are of lengths m, n, where possibly $m \neq n$. Given two DNA sequences, the goal is to compute the best possible (minimum) *alignment cost* between them. We will soon specify how to compute the alignment cost between two sequences.

Application:
Why find the alignment cost? Suppose you want to find the role played by a DNA sequence you find in an organism. Instead of carrying out experiment after experiment trying to determine the function, we can use prior information to our advantage: there is usually a good chance that it is a variant of a known gene in a previously studied organism. DNA sequences and their functions in many species have been studied and published in publicly available databases. Thus, a more efficient way would be to compare our sequence for a match with a sequence in an already studied organism. However, DNA strands are subject to minor random mutations from organism to organism, so we cannot expect that we will find an exact match. Nevertheless, two sequences that are more or less similar can be expected to perform the same function. We try to align our (query) sequence with these target sequences, and compute an alignment cost; finding a low alignment cost with a target sequence would give us a reasonable guess about the role played by the query sequence in our organism.

Alignment and scoring:
Consider the query and target sequences: `AGGGCCT`, `TGGCA` respectively. Note that these are of different lengths, so one natural way to pair them would be:

```
A G G G C C T
T G G - C - T
```

Here, `-` is called a *gap*, that can be inserted anywhere in either string, to make the rest of sequences align better. There is a penalty $s(x, y)$ for matching character x to character y. Consider the following scoring function:

$$s(x,y) = \begin{cases} 0, & \text{if } x = y \\ 1, & \text{if } x \neq y \\ 2, & \text{if } x = - \text{ or } y = - \end{cases}$$

Note that a smaller cost means better alignment. The cost of aligning a character with the gap character is usually denoted by γ, and is called the **gap-penalty**.

According to the above scoring rules, the pairing given above has an alignment cost of $1 + 2 + 2 = 5$. Consider another example: two possible alignments of `AACAGTTACC` and `TAAGGTCA` and corresponding alignment costs are:

```
Target Sequence: A A C A G T T A C C       Target Sequence: A A C A G T T A C C
Query Sequence:  T A A G G T C A - -       Query Sequence:  T A - A G G T - C A
Penalty:         1 0 1 1 0 0 1 0 2 2       Penalty:         1 0 2 0 0 1 0 2 0 1
Total Penalty = 8                          Total Penalty = 7
```

Thus, the second alignment is better than the first. Note we may decide to insert gap characters in either target or query string to minimize the alignment scores, and not just in the string with smaller length.

Because of the nature of sequence matching, a no-crossing rule is always there: suppose $S_1 = \ldots x_1 \ldots y_1 \ldots$ and $S_2 = \ldots x_2 \ldots y_2 \ldots$. Then, it cannot be that x_1 is matched to y_2 and y_1 is matched to x_2.

Problem 1: (20 points)
Given two sequences, write C++ code to compute the cost, and alignment pattern of the optimal alignment given by the $s(x, y)$ function specified above. The first sequence (target) will be specified first, without spaces, followed by a newline, followed by the second sequence, followed by a newline. The output should be (each on a new line):

1. The optimal alignment score (a single integer)
2. The pattern with gaps of the target sequence that is matched (without spaces)
3. The pattern with gaps of the query sequence that is matched (without spaces)

Output: There may be multiple output patterns possible with the same alignment score. You are to output the alignment patterns in which the query DNA sequence pattern is lexicographically the least, when read from *right to left.* You should consider the character ordering: `_ < A < C < G < T`. The gap character is considered the smallest.

e.g. `ACCG_TG` is larger then `ACCT_G`, since read from the right, `_` is smaller than `T`. This ordering makes sense even for strings of unequal length: `AA_CGG` is larger than `T_CG_G`.

For the above example, the input would be:

```
AACAGTTACC
TAAGGTCA
```

The output would be:

```
7
AACAGTTACC
TA_AGGT_CA
```

Above, the gap character is denoted by an underscore "_". You may assume that the input will always be valid (i.e. need not check for invalid inputs).

Hint: The approach is to have the subproblem $M[i, j]$ to denote the optimal cost of aligning $S_1[1, \dots i]$ with $S_2[1, \dots, j]$.

Problem 2: (10 points)
The penalties given by the previous function are quite simplistic. Suppose that the matching penalties are specified by the following matrix (actually used in DNA sequence alignment), and a gap-penalty of $\gamma = 30$ applies for matching against the gap character.

	A	C	G	T
A	-91	114	31	123
C	114	-100	125	31
G	31	125	-100	114
T	123	31	114	-91

Output the optimal alignment cost and patterns, in a format similar to the previous problem. You may reuse the code from Problem 1, appropriately modified.

Problem 3: (20 marks)
Often, it is simplistic to assume that all gaps are scored equally. Consider the two strings `AAAGAATTCA` and `AAATCA`, and two possible alignments:

```
AAAGAATTCA                    AAAGAATTCA
A-A-A-T-CA                    AAA----TCA
```

The second one seems more natural: inserting `GAAT` into a single point in the query string gives the target string, and the second alignment captures this better. However, under our previous cost functions, both of the above alignments have the same final cost (8), because all gaps are counted alike. Instead, we would like to have that a sequence of k consecutive gaps is penalized less than k gaps spread all over. So, to do this, we generalize the gap penalty γ to considering a continuous sequence of k gaps:

$$\gamma(k) = \delta + (k - 1) \times \beta$$

Above, δ is called the *gap-opening* penalty, and β is called the *gap-extension* penalty. That is, you pay δ for opening a gap sequence, but for further gaps attached to this, you pay only β. If, for example, $\delta = 2$ and $\beta = 1$, the first alignment has a cost of $2 \times 4 = 8$, but the second one has a cost of $2 + 3 = 5$. Such a gap penalty model is called the *affine gap-penalty model.*

Note that the previous dynamic programming approach suggested will fail in this setting, as it does not take into consideration the number of gaps happening together.

You have to write C++ code to find the optimal alignment (cost and pattern) under this new cost function. The input and output specifications remain the same as before.

Use the character alignment costs from Problem 2. The gap-opening penalty is $\delta = 400$, and gap-extension penalty is $\beta = 20$.

Hint: You will need two more arrays besides $M[i, j]$:

$$A[i, j] : \text{cost of best alignment of } S_1[1..i] \text{ and } S_2[1..j] \text{ ending with a space in } S_1$$
$$B[i, j] : \text{cost of best alignment of } S_1[1..i] \text{ and } S_2[1..j] \text{ ending with a space in } S_2$$

Write out the appropriate recursion using these, and use it to solve the problem.

Submission Guidelines

1. Your submission will be one zip file (or `.tar.gz` file) named `<roll-number>.zip` (or `<roll-number>.tar.gz`), where you replace `roll-number` by your roll number (e.g. `cs19mtech11003.zip`), all in small letters. The compressed file should contain the below mentioned files:
   (a) One file for each problem: `<roll-number>_<problem-number>.cpp` (e.g. `cs19mtech11003_1.cpp`).
   (b) A brief description of your solution in **pdf** format named `report.pdf`. This file should describe the algorithms you use for implementing the functions, and briefly state what each function in your code does. Also, state and argue about the asymptotic time complexity and space complexity of your algorithms.
2. The preferred way to write your report is using LaTeX(Latex typesetting). [Not mandatory].
3. **Follow the output format strictly. Do not print a single extra character space over what is needed.**
4. In evaluations, programs will be compiled using `g++`, with `-std=c++14` option (similar to Assignment-2). Preferably use a gcc version greater than 8 for testing your code. Mention the gcc version you compiled and tested your code with in your report.
5. Upload your submission to the Google classroom link. You may reupload multiple versions till the deadline; only the final version will be considered.
6. Failure to comply with the above instructions will result in your submission not being evaluated (and you being awarded 0 for the assignment).

7. **Late submission policy**: For every day (24 hours) late: -10 from your score on the assignment.

8. **Plagiarism policy:** If we find a case of plagiarism in your assignment (i.e. copying of code, either from the internet, or from each other, in part or whole), you will be awarded a **zero**. Note that we will not distinguish between a person who has copied, or has allowed his/her code to be copied; both will be equally awarded a zero for the submission.

1 Evaluation

You will be marked for the following aspects:

1. Correctness and Efficiency of the implementation.
2. Code clarity and comments.
3. Report presentation.