4470/6470 Project:
Text Search via Local Alignment

Spring 2019

Due Monday April 8, 2019

This programming assignment is to code and test a text searching method based on an idea of alignment between two text sequences. This alignment problem, called **Local Alignment**, is a variant of the original alignment problem (called **Global Alignment**) studied in this course. The program you are to develop is expected to take as input two sequences: a potentially very large text sequence, called *target* stored in a text file, and a potentially very short text sequence called *query* entered at the prompt. Your program will output some most significant hits (not necessarily exact matches) between the short query sequence and parts of the large target text file. Obviously the program can be extended to suit the need of searching for a short query text sequence on a large database that may contain many target text files.

1 Local Alignment

The idea of local alignment originated from research in computational biology where identification of motifs conserved across biological sequences of different species is highly desirable. The significance of the idea does not diminish when it is applied to text search from databases or even on the internet. You may even think of this program to be a primitive online search engine where output hits are not yet to be semantically structured or sorted based on the contents.

Recall that **Global Alignment** problem is to align two text sequences X and Y to obtain the highest sum of the column-wise scores based on a predefined scoring scheme. It permits exact match, substitution, deletion and insertion and rewards them with respective positive or negative scores. The objective function that facilities dynamic programming is $f(i, j)$, the maximum score of an alignment between the first i letters in X and the first j letters in Y. It has the following recurrence:

$$f(i,j) = \max \begin{cases} f(i-1,j-1) & + S(x_i, y_j) \\ f(i-1,j) & + S(x_i, '-') \\ f(i,j-1) & + S('-', y_j) \end{cases} \tag{1}$$

For $i \geq 1,\ f(i,0) = f(i-1,0) + S(x_i, '-')$,
for $j \geq 1,\ f(0,j) = f(0,j-1) + S('-', y_j)$,
and $f(0,0) = 0$.

where S gives a score between two letters, including the gap $'-'$.

Therefore, a global alignment between two prefixes can be built upon a global alignment between two sub-prefixes, regardless if the latter has low (e.g., negative) scores or not.

Local alignment, however, builds a local alignment between two prefixes upon a local alignments between sub-prefixes but only if the latter has positive score. If the latter has a negative score, it will not be the part of the desired solution. The following figure shows the difference between a global alignment and a local alignment, where $'|'$ represents a match, and $'.'$ represents a substitution. For the local alignment, the alignment within the red rectangle box is a local alignment.

```
global alignment                local alignment
  r-unning                         running
  | | ||..                          | ||
  reu-nion                        reu-nion
```

So local alignment needs a step to abandon negatively scored alignments of prefixes. The recurrence is thus modified to

$$f(i,j) = \max \begin{cases} f(i-1,j-1) & + S(x_i, y_j) \\ f(i-1,j) & + S(x_i, '-') \\ f(i,j-1) & + S('-', y_j) \\ 0 \end{cases} \tag{2}$$

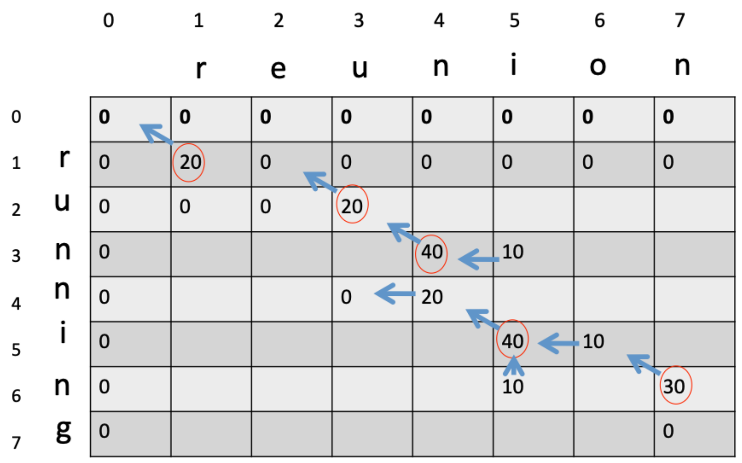
And for base cases, accordingly, $f(i,0) = f(0,j) = 0$.

The addition of the case of value 0 in recurrence (2) is to make sure that a subcase alignment of a negative score is not part of the result.

2 DP table and local alignment trace back

Much like for the **Global Alignment**, dynamic programming table can be set up and calculated for the **Local Alignment**, using the recurrence (2) and the corresponding base cases. Arrows can also be added to remember the option taken during the calculation of every cell.

However, traceback of **the best** local alignment does not necessarily start from the cell at the right bottom corner. The traceback should start from the cell of the highest score. In the example of local alignment between `running` and `reunion`, if we assume that a match has reward score 20, an insertion/deletion has penalty score -30 (and substitution has a much bigger penalty score), we may obtain the following dynamic programming table.



		0	1	2	3	4	5	6	7
			r	e	u	n	i	o	n
0		0	0	0	0	0	0	0	0
1	r	0	20	0	0	0	0	0	0
2	u	0	0	0	20				
3	n	0				40	10		
4	n	0			0	20			
5	i	0					40	10	
6	n	0					10		30
7	g	0							0

The highest score is in the cell $f(3,4) = 40$. Beginning from cell $(3,4)$, traceback of alignment will follow the recorded arrow to the next cell until a cell of value 0 is encountered. For this example, it will be cell $f(1,2) = 0$. Note that cell $f(5,5) = 40$ is also of the highest score. So there may be more than one most plausible local alignments.

In addition, We may also be interested in tracing back all local alignments whose scores may not be the highest but are significant enough. In this example, we may also want to traceback local alignments from the cells $f(6,7) = 30$, $f(2,3) = 20$, and $f(1,1) = 20$, respectively.

You probably already notice that some of the traced local alignments overlap. For example the alignment obtained by traceback from cell $(6,7)$ contains the alignment obtained by trackback from cell $(5,5)$. To avoiding tracing back local alignments already traced, it suffices to place value 0 in every cell after it has been visited in a traceback. But this may not be necessary, depending on applications.

3 Requirements

3.1 The program

The basic requirement for this project is to implement the **Local Alignment** algorithm into a standalone program. It should take two data inputs: one large target text sequence stored in a text file and one short query text sequence entered at the prompt. The program should also accept a number k from the prompt to indicate that local alignments of the k top highest scores will be traced and outputted from the program.

The output format: for each traced local alignment, the program outputs the corresponding start and end indexes of the segment in the target text file as well as the local alignment between the target segment and the query. For example, for the example given earlier, for $k = 3$, the program may output the following 3 hits

```
hit 1: score = 40
       [2]un[3]
          ||
        reunion

hit 2: score = 40
       [4]ni[5]
          ||
       reunion

hit 3: score = 30
       [4]ni-n[6]
          || |
       reunion
```

You should imagine that both the target segment and query sequences can be much longer. They may also contain the blank symbol ' ' as in English articles and sentences.

3.2 Scoring scheme

I leave the choice of a scoring scheme for you to decide since it is highly dependent on applications and other factors such as type of texts, what to search, the length of query sequence, etc.. You may have to tune the scoring gradually until it reaches some satisfactory level. You can hard-code the scoring scheme within the program but a more flexible way is to put it in another small text file the program reads it when it begins to run.

3.3 Demo

Upon the completion of your program, you should demo it in my office. You will need to prepare a large text file before the demo. Your program will be tested on this text file to search for some query sequences given by me and entered at the prompt.

3.4 Space efficiency issue (*Optional*)

For a large text file (e.g, of size $10,000$), the dynamic programming table would be of size $10,000q$, where q is the length of query text. To save memory, one may consider to use a sliding window on the large target text file one position per frame and to compute local alignment between the short query sequence and each segment of the target text covered each window frame. The size of window can be just several times larger than q, the length of query sequence. Different window frames can reuse the dynamic programming table so the memory usage is reduced from $10,000q$ to $O(q^2)$.

However, if this method to save space is adopted, we will need to resolve two other issues.

First, the time complexity will become higher. Without using a sliding window, the time complexity is $\theta(100,000q)$ proportional to the size of table. With the sliding window, since at each window position, the time $\theta(q^2)$ is used and the total time is $\theta(100,000q^2)$. This problem can be solved by reuse most of the table data computed for in the previous window frame but make the table computation a slightly more complicated.

Second, for each window frame computed, while all significant local alignments can be extracted, we do not know how significant they are compared to those to be obtained from other window frames. Therefore, some statistics based method may have to be used. For this part, please contact me to discuss some details if you are interested.