Lab 6: Dynamic Programming: Part 3. String Comparison Algorithms

Due date: Thursday, May 25, beginning of the lab.

Lab Assignment

Assignment Preparation

This is a **pair programming** lab. The goal of this lab is to let you implement more dynamic programming algorithms.

Each pair will develop, validate, test and submit one copy of the assignment. Each partner in a pair will get the same grade for this lab.

The Task

Each team will implement three algorithms for comparing two strings of text:

1. LCS: Longest Common Subsequence;
2. Edit Distance;
3. Alignment;

Brief notes concerning each of the three algorithms are given below.

Longest Common Subsequence

Your implementation of the LCS (Longest Common Subsequence) algorithm shall take as input two strings and shall compute all necessary data structures. A separate method shall be implemented to print the discovered longest common subsequence and its length out.

You can implement `LCS` by creating a Java class (say `LCS`) whose instance variables include the two tables used in the dynamic programming algorithm for finding the `LCS`. Then, you will need to implement two methods:

- `void findLCS(String x, String y)` (or, alternatively `int findLCS(String x, String y)`): this method takes two strings and runs the dynamic programming algorithm that discovers their LCS by filling out the contents of the two tables (see Section 15-4 of the textbook). The method can return the length of the found LCS.
- `String getLCS():` this method shall assume that `findLCS()` has successfully run. It will traverse the created tables to recover the actual longest common subsequence and return it.

(as usual, you may implement any other methods in this class)

Program for validation. Your submission shall include a program `LCSTest.java`. The program will take as input either one or two file names.

- If only one file name is provided, it shall contain two strings separated by a line break in it.
- If two file names are provided, each file shall contain one string. The string shall be assembled from the entire content of the file by removing all line-feed and carriage return characters from it

The `LCSTest.java` program shall read the input strings from either one or two files, shall run the `LCS` algorithm on them, and shall output (print) the longest common subsequence and its length.

Example. Consider the following two files `in1` and `in2`:

```
in1:
ABCABC
ACCACC

in2:
ABCABBCCA
```

Then:

```
java LCSTest in1
```

shall find the LCS of `"ABCABC"` and `"ACCACC"` and output something like:

```
4: ACAC
```

(note, you may choose a somewhat different output format as long as both the LCS and its size a clear)

At the same time,

```
java LCSTest in1 in2
```

shall find the LCS of `"ABCABCACCACC"` and `"ABCABBCCA"` and shall output something like

```
8: ABCABCCA
```

Program for experimental study. Each team shall conduct a short experimental study using the LCS implementation. You will measure the running time of your implementation of the LCS algorithm (only the part that fills the data structures with data, *not the recovery part*), as it depends on the size of the input.

The study shall consist of two parts. In one part you will compare a string to itself. In the other part you will compare randomly generated strings. In both parts you will generate strings of various increasing sizes. For simplicity, use the alphabet[1]

$$\{A, T, C, G\}$$

For the first part, for each string size you consider, generate 10 random strings of that size and run the LCS algorithm on each string compared to itself.

For the second part, for each string size you consider, generate 10 pairs of random strings and run the LCS algorithm on each pair.

In both studies, for each size considered, collect the average of the 10 runs.

You can choose string sizes on which you run your algorithm in any way you want, provided that **your study accomplishes the following:**

- You report results for at least 10 different sizes.
- The study *pushes the limits* of productivity of your algorithm, i.e., at the 2-3 longest string sizes, your implementation starts taking noticable time to complete.

Name the program running your study (`LCSStudy.java`.

Based on the results reported by the program, prepare a report that contains the following information:

- graphs/plots documenting the results of your study.
- your observations on how far your implementation goes (i.e., when your implementation starts taking considerable time to finish).
- your observations on whether it appears to be easier or harder to find an LCS of a string with itself than it is to find an LCS of two random strings, or whether there is no difference.

[1]Note, while only this alphabet is used in all your studies, all your algorithm implementations shall be alphabet-independent.

Edit Distance

The version of the Edit Distance problem you will be implementing is the one we covered in class (not the one found in the textbook)[2]

To remind you, the Edit Distance problem is stated as follows: given two strings x and y, find the minimal number of edits required to change x into y. The edits we consider for this assignment are:

- Insertion. A character y_i is inserted in position i of the string.
- Deletion. A character x_i is deleted from position i.
- Replacement. A character x_i is replaced with character y_i.

Alternatively, we can cast the edit distance problem as follows. An *alignment* between two strings x and y, is a mapping that pits each character x_i of x against either some character y_j of y or against an empty slot _, and which respectively pits every character y_i of y against some character x_i of x or _, such that if $i < j$ and both x_i and x_j are mapped to characters y_k and y_l respectively of y, then $k < l$ (i.e., the mapping follows the order of characters in the words).

Example. Consider two strings $x =$"STAND" and $y =$"SNEEZE". Below we show two possible alignments of these two strings:

```
STAND_                      STAND___
SNEEZE                      S__NEEZE
```

The *cost of alignment* is the number of positions in the alignment where the two characters do not match. For example, the cost of the first alignment above is 5, and the cost of the second alignment is 6.

The **edit distance** between x and y is the smallest alignmnet cost.

Implementation

Your implementation of the Edit Distance algorithm shall take as input two strings, compute all necessary data structures and return the edit distance. A separate method shall be implemented to retrieve an alignmnet corresponding to the computed edit distnce.

You can implement the edit distance algorithm by creating a Java class (say `EditDistance`) whose instance variables include the tables used in the dynamic programming algorithm for finding the the edit distance. Then, you will need to implement two methods:

[2] The textbook (Section 15-5) describes a more general problem, with more different types of possible edits.

- `int findEditDistance(String x, String y)`: this method takes two strings and runs the dynamic programming algorithm that discovers their edit distance by filling out the contents of the two tables. The method returns the edit distance.
- `void getAlignment()`: this method shall assume that `findEditDistance()` has successfully run. It will traverse the created tables to recover the alignment associated with the edit distance and print it out. To print out the alignment use the convention as above: print each string on a line of its own, and indicate insertions and deletions using the "_" character.

(as usual, you may implement any other methods in this class)

Program for validation. Your submission shall include a program `EDTest.java`. The program will take as input either one or two file names.

- If only one file name is provided, it shall contain two strings separated by a line break in it.
- If two file names are provided, each file shall contain one string. The string shall be assembled from the entire content of the file by removing all line-feed and carriage return characters from it

The `EDTest.java` program shall read the input strings from either one or two files, shall run the edit distance algorithm on them, and shall output (print) the edit distance between the two strings and the corresponding alignment.

Example. Consider the following two files `in1` and `in2`:

```
in1:
ABCA
ACA

in2:
ABCAACCA
```

Then:

```
java EDTest in1
```

shall find the edit distance of `"ABCA"` and `"ACA"` and output something like:

```
Edit distance: 1
ABCA
A_CA
```

(note, you may choose a somewhat different output format as long as both the edit distance and the alignmnet are clear)

At the same time,

```
java ATest in1 in2
```

shall find the edit distance of `"ABCAACA"` and `"ABCAACCA"` and shall output something like

```
Edit distance: 1
ABCAAC_A
ABCAACCA
```

Program for experimental study. Each team shall conduct a short experimental study using the edit distance implementation. You will measure the running time of your implementation of the edit distance algorithm (only the part that finds the edit distance, *not the alignment recovery part*), as it depends on the size of the input.

The study shall consist of two parts. In one part you will compare a string to itself. In the other part you will compare randomly generated strings. In both parts you will generate strings of various increasing sizes. For simplicity, use the alphabet

$$\{A, T, C, G\}$$

For the first part, for each string size you consider, generate 10 random strings of that size and run the edit distance algorithm on each string compared to itself.

For the second part, for each string size you consider, generate 30 pairs of random strings and run the edit distance algorithm on each pair.

In both studies, for each size considered, collect the average of the 10/30 runs. In the second study, for each size, collect the average edit distance observed.

You can choose string sizes on which you run your algorithm in any way you want, provided that **your study accomplishes the following:**

- You report results for at least 10 different sizes.
- The study *pushes the limits* of productivity of your algorithm, i.e., at the 2-3 longest string sizes, your implementation starts taking noticable time to complete.

Name the program running your study (`EDStudy.java`.

Based on the results reported by the program, prepare a report that contains the following information:

- graphs/plots documenting the results of your study.
- your observations on how far your implementation goes (i.e., when your implementation starts taking considerable time to finish).
- your observations on whether it appears to be easier or harder to find the edit distance of a string with itself than it is to find the edit distance of two random strings, or whether there is no difference.
- a plot documenting the observed behavior of the average edit distance between two random strings on the size of the strings.
- your observations concerning the behavior of the average edit distance as the size of the strings increases.

Note: If you are getting weird results with only 30 repetitions, try 100, 200, 500 or 1000 repetitions for each size of the strings you include in your study and take an average. As you increase the number of repetitions, you should at some point start seeing *some* stabilized behavior[3].

Alignment

The string alignment problem is a generalization of the edit distance problem.

An alignment between two strings is defined in the same way as for the edit distance problem.

What differs is the notion of the *cost of alignment.* We are given a matching function M which gives a cost of aligning two characters. The cost of the alignment is the sum of costs of aligning each pair of characters.

Example. Consider two strings, $x =$`"START"` and $y =$`"ARTS"`. Our alphabet is $\{A, T, S, R\}$. Consider the matching cost function M defined in the table below:

	A	T	R	S	_
A	0	1	5	5	2
T	1	0	5	5	2
R	5	5	0	1	2
S	5	5	1	0	2
_	2	2	2	2	∞

According to this table, $M(A, T) = 1$, while $M(A, R) = 5$, which essentially means that it is much better to align A with T than it is to align it with R. The function M in the example above is symmetric and $M(c, c) = 0$ for any character c, but these two properties, while common in real-life situations, are not requirements.

Cosider the following two alignments of our strings x and y:

[3]This is a true exploration task. I actually don't know how this dependency would look like and am curious to find out.

```
S T A R T _         S T A R T _
_ _ A R T S         _ A _ R T S
```

The cost of the first alignment is computed as follows:

$cost_1 = M(S, \_) + M(T, \_) + M(A, A) + M(R, R) + M(T, T) + M(\_, S) = 2 + 2 + 0 + 0 + 0 + 2 = 6.$

The cost of the second alignmnet is:

$cost_2 = M(S, \_) + M(T, \_A) + M(A, \_) + M(R, R) + M(T, T) + M(\_, S) = 2 + 1 + 2 + 0 + 0 + 2 = 7.$

We see that the first alignment has a lower cost.

The alignmnet problem is to find an alignment of two given strings x and y which *minimizes the alignment cost* under a given matching function M.

Note: The edit distance problem is essentially an instance of the alignmnet problem, where the matching cost function M is $M(c, c) = 0$ and $M(c, d) = M(\_, d) = M(c, \_) = 1$ for $c \neq d$.

Implementation

Modify your edit distance algorithm to solve the alignment problem. Implement the solution as described below.

Your implementation of the Alignment algorithm will be a Java class Alignment which will have the following methods:

- `void setMatchingCost(..)`: this method shall take as input a data structure representing a matching cost function, and shall set it as the matching cost function for the alignment cost computations. The choice of how to represent the matching cost function internally is left to you. The `Alignment` class should have an instance variable (or variables) representing the matching cost function.
- `int findAlignment(String x, String y)`: this method takes two strings and runs the dynamic programming algorithm that discovers their best by filling out the contents of the necessary tables. The method returns the cost of the best alignment.
- `void getAlignment():` this method shall assume that `findAlignment()` has successfully run. It will traverse the created tables to recover the alignment associated with the best alignment cost and print it out. To print out the alignment use the convention as above: print each string on a line of its own, and indicate insertions and deletions using the "_" character. (you may also, optionally indicate the cost of aligning each pair of characters)

To help your algorithm, you may implement the following two methods:

- `Match(char x, char y)`: returns $M(x, y)$.

- `getCost(String x, String y)`: x, y are two strings of the same length representing an alignment (i.e., some characters can be "_"). The method returns the cost of the given alignment between x and y.

  Note: your implementation may find it more convenient for the parameters of this method to be something else (depending on the data structures you use in the algorithm).

(as usual, you may implement any other methods in this class)

Program for validation. Your submission shall include a program `ATest.java`. The program will take as input either two or three file names.

- If two file names are provided, the first file will contain two input strings separated by a line break, while the second file will contain the description of the matching function (see below).
- If three file names are provided, the first two files shall contain the two input strings. Each string shall be assembled from the entire content of the file by removing all line-feed and carriage return characters from it. The third file shall contain the desription of the matching function.

The `ADTest.java` program shall read the matching function and the input strings from either one or two files, shall run the `alignment` algorithm on them, and shall output (print) the best alignment cost and the best alignment between the two strings.

Representing matching function. The matching function file shall have a simple format. The first line of the file is a comma-separated list of alphabet characters, followed by _, representing the insertion/deletion in the alignment. The subsequent rows of the file store the values of the cost function. If $A_1, A_2, \ldots, A_K, \_$ is the order of the characters in the first row, then the second row represents the costs

$M(A_1, A_1), M(A_1, A_2), \ldots, M(A_1, A_K), M(A_1, \_)$.

Subsequent rows continue the pattern. $M(\_, \_)$ will always be $+\infty$, but for the sake of ease of parsing, it will be listed in the file as 0. (you just need to make sure that you use the right value when computing). For example, a file `match` representing the matching cost function M from above will look as follows:

```
A,T,R,S,_
0,1,5,5,2
1,0,5,5,2
5,5,0,1,2
5,5,1,0,2
2,2,2,2,0
```

Example. Consider the following two files `in1` and `in2`:

```
in1:
START
ARTS


in2:
STARTSTART
```

Then:

```
java ADTest in1 match
```

shall find the best alignment of `"START"` and `"ARTS"` and output something like:

```
Alignment cost: 6
START_
__ARTS
```

(note, you may choose a somewhat different output format as long as both the alignment cost and the alignmnet are clear)

At the same time,

```
java ATest in1 in2 match
```

shall find the best alignment of `"STARTARTS"` and `"STARTSTART"` and shall output something like

```
Alignment cost: 6
START__ARTS
STARTSTART_
```

Program for experimental study. Each team shall conduct a short experimental study using the alignment implementation. You will measure the running time of your implementation of the alignment algorithm (only the part that finds the best alignment cost, *not the alignment recovery part*), as it depends on the size of the input.

The study shall consist of two parts. In one part you will compare a string to itself. In the other part you will compare randomly generated strings. In both parts you will generate strings of various increasing sizes. For simplicity, use the alphabet

$$\{A, T, C, G\}$$

and the matching function M:

```
A,T,C,G,_
0,1,3,3,2
1,0,3,3,2
3,3,0,1,2
3,3,1,0,2
2,2,2,2,0
```

For the first part, for each string size you consider, generate 10 random strings of that size and run the alignment algorithm on each string compared to itself.

For the second part, for each string size you consider, generate 30 pairs of random strings and run the alignment algorithm on each pair.

In both studies, for each size considered, collect the average of the 10/30 runs. For the second study, also collect the average best alignment cost.

You can choose string sizes on which you run your algorithm in any way you want, provided that **your study accomplishes the following:**

- You report results for at least 10 different sizes.
- The study *pushes the limits* of productivity of your algorithm, i.e., at the 2-3 longest string sizes, your implementation starts taking noticable time to complete.

Name the program running your study (`AStudy.java`.

Based on the results reported by the program, prepare a report that contains the following information:

- graphs/plots documenting the results of your study.
- your observations on how far your implementation goes (i.e., when your implementation starts taking considerable time to finish).
- your observations on whether it appears to be easier or harder to find the alignment of a string with itself than it is to find the alignment of two random strings, or whether there is no difference.
- a plot documenting the observed behavior of the average best alignment cost for two random strings as the size of the strings increases.
- your observations concerning the behavior of the best alignment cost as the size of the strings increases.

Note: If you are getting weird results with only 30 repetitions, try 100, 200, 500 or 1000 repetitions for each size of the strings you include in your study and take an average. As you increase the number of repetitions, you should at some point start seeing *some* stabilized behavior[4].

[4]This is a true exploration task. I actually don't know how this dependency would look like and am curious to find out.

Report

Your written report shall contain the results of the studies of all three algorithm implementations as specified above.

As most of you use MS Excel to generate graphs and plots for your studies, please follow the instructions below.

- **Legend.** Make certain that all your axes are properly labeled. Each series needs to be properly named. The legend shall be placed *at the bottom of the graph*: this makes it possible to extend the coordinate plane in the X direction.
- **Size.** Please, do not hesitate to include **large** graphs/plots in your report. The larger the graph, the more visible your results are.
- **Series.** Please, edit the plot/graph to make sure that the size of MS Excel elements representing the points in the series is appropriate. Usually, this means *reducing* the size of the "points".
- **Coordinate plane.** Please edit the properties of the graph/plot to make sure that your coordinate plane looks appropriate. The X-axis and Y-axis ranges should be appropriate (sometimes Excel's best guess can be substantially improved).
- **Overall look.** Please make certain that the graph/plot *looks good to you.* Do no hesitate to edit the properties of the plot to make it look better and to make the data more visible.

In addition, before creating a graph/plot, please take a pause and think about the best way in which you can represent the data. Pick the representation that would make the data more understandable to *you.*

In this lab, poor choices for information visualization will be subject to penalties.

Deliverables

This part of the lab has both electronic devliverables and a hardcopy deliverable.

All electronic deliverables shall be submitted by the assignment deadline using the following `handin` command:

Use `handin` to submit:

```
$ handin dekhtyar-grader lab06-349 <files>
```

Electronic Deliverables. Submit all the Java files you created for this lab. This should include your `LCSTest.java, LCSStudy.java, EDTest.java, EDStudy.java, ATest.java, AStudy.java` test programs in addition to the Java classes implementing the dynamic programming algorithms and any supplemental classes you have. Submit a README file with the list of people on the team and any notes/comments/instructions. Finally, submit an electronic copy of your report in PDF format.

Hardcopy Deliverables. At the beginning of the lab period on May 25 hand in the hardcopy of your report.

Good Luck!