ITCS-6114/8114: Algorithms and Data Structures
Project 1 — Sequence Comparison

Due Date

The due date is **Tuesday, March 10, 2015 by 11:59:59 pm.** See the syllabus for late policies. See below for submission guidelines.

Introduction

A lot of practical data is stored in the form of a sequence. A sequence is a set of items that are arranged in a specific order. For example, a strand of DNA consists of four different bases: adenine, cytosine, guanine, and thymine. These bases are represented by their first letters and a strand of DNA can be expressed as a string consisting of the following set of characters: $\{a, c, g, t\}$. For example, the DNA for one organism may be $S_1 = agttgtagct$ while the DNA for another organism may be $S_2 = agtgctact$. (In practice, the DNA for an organism has millions of bases.) In computational biology, it is useful to compare DNA for similarity. When comparing two DNA strings, exact matching is not always important. An exact matching algorithm can only tell you if two DNA strings are equal; it is useful to have a measure of similarity that is not binary.

In this project, we will study two measures of similarity for comparing sequences: **normalized edit distance** and **longest common subsequence**. Your objective is to compute the normalized edit distance and the longest common subsequence between two sequences. Note that while the term string suggests that the set of items are characters, the term sequence implies that the items can be anything. In this project description, we will use the term string to describe the problems and the algorithms.

Normalized Edit Distance

One of the simplest measures of similarity is the **edit distance** between two strings. The edit distance between string S_1 and S_2 is the minimum number of symbol deletions required to transform S_1 and S_2 into a common string.

Example 1

Given:
$S_1 = agttgtagct \quad S_2 = agtgctact$

To transform S_1 and S_2 into a common string we can apply three deletions.

1. delete $S_1[3]$ which is a t
2. delete $S_1[7]$ which is a g
3. delete $S_2[4]$ which is a c

The result is the string *agtgtact*. The absolute minimum number of deletions required to transform S_1 and S_2 into a common string is three. See for yourself: Try to transform the strings using only two deletions. You will see that it is impossible.

A good measure of similarity between two strings is the **normalized edit distance (d_N)**, which is computed using the formula below.

$$d_N = \frac{|S_1|+|S_2|-d}{|S_1|+|S_2|}$$

where $|S_1|$ is the length of the input string S_1, $|S_2|$ is the length of the input string S_2, and d is the edit distance between string S_1 and S_2. In Example 1, the normalized edit distance is computed as follows:

$$|S_1| = 10,\ |S_2| = 9,\ d = 3.$$

$$d_N = \frac{10+9-3}{10+9} = \frac{16}{19} = 0.842$$

Two strings that are identical would have a normalized score of 1.0 and two strings that are completely unrelated would have a score of 0.0.

Longest Common Subsequence

Normalized edit distance is only a numerical measure of similarity. In computational biology, it is often useful to see the similarity in a more meaningful way. One of the simplest ways to represent the similarity between two strings is to identify the **longest common subsequence (LCS)**. A subsequence of a given sequence is just the given sequence with zero or more elements deleted. In simple terms, the LCS is the string that is left over after you have applied the minimum number of deletions to transform two strings into a common subsequence. In more formal terms, the LCS of S_1 and S_2 is the longest subsequence that is a subsequence of S_1 and a subsequence of S_2.

Example 2

Given: $S_1 = agttgtagct$ $\quad S_2 = agtgctact$

$LCS(S_1, S_2) = agtgtact$

Note that $LCS(S_1, S_2)$ appears in both S_1 and S_2 in order.

$S_1 =$ **agt** t **gta** g **ct** $\quad S_2 =$ **agtg** c **tact**

Note that while a common substring must appear in the two strings exactly, a common subsequence can skip some characters so long as the relative ordering of the characters is always preserved.

Computing Edit Distance

Consider the following naive approach to computing the edit distance between two strings:

1. Given two strings, try deleting each symbol of the two strings (one at a time) and see if the two strings are ever equal.
2. If the strings with a single deletion are never equal, try deleting every possible combination of two symbols and see if the two strings are ever equal.
3. If the strings with two deletions are still never equal, allow for exactly three symbol deletions and repeat.

4. Repeat the process, stopping when the strings with deletions are equal. Return the number of deletions that were required.

This naive approach is not only tedious to implement, but is also very inefficient. The implementation involves a process that redundantly computes many sub-problems. To make the algorithm more efficient, we will use a technique called dynamic programming, which systematically records the answers to sub-problems in a table. Dynamic programming eliminates the redundant computation of sub-problems through the use of additional memory.

The dynamic programming approach is recursive. We compute the edit distance of two strings incrementally by first computing the edit distance of the prefixes of the two strings. In turn, the edit distance of each prefix can be computed from the edit distance of even smaller prefixes. This process will eventually continue until the prefix size is one. At this point, the problem of determining edit distance is as simple as comparing the first two symbols in the strings.

To perform this recursive computation, we build a table where each table cell corresponds to the edit distance of a pair of prefixes. The initial table is shown in Figure 1*(a)*. One string Y is aligned across top and the other X is aligned down the left. Let X have m symbols and let Y have n symbols. Each cell of the table T can be identified using indices. For example, $T[i][j]$ would correspond to the cell in row i and column j. Row zero and column zero are always initialized as shown in the figure. The initialization actually has meaning. For example, the value zero in $T[0][0]$ indicates that the edit distance of two empty strings is zero. The value five in $T[0][5]$ indicates that the edit distance between $Y[0,\ldots,4] = agtac$ and the empty string is five. And the value nine in $T[9][0]$ indicates that the edit distance between $X[0,\ldots,8] = gtatcgtat$ and the empty string is nine. In general, the value $T[i][j]$ represents the edit distance between the strings $Y[0,\ldots,j\text{-}1]$ and $X[0,\ldots,i\text{-}1]$.

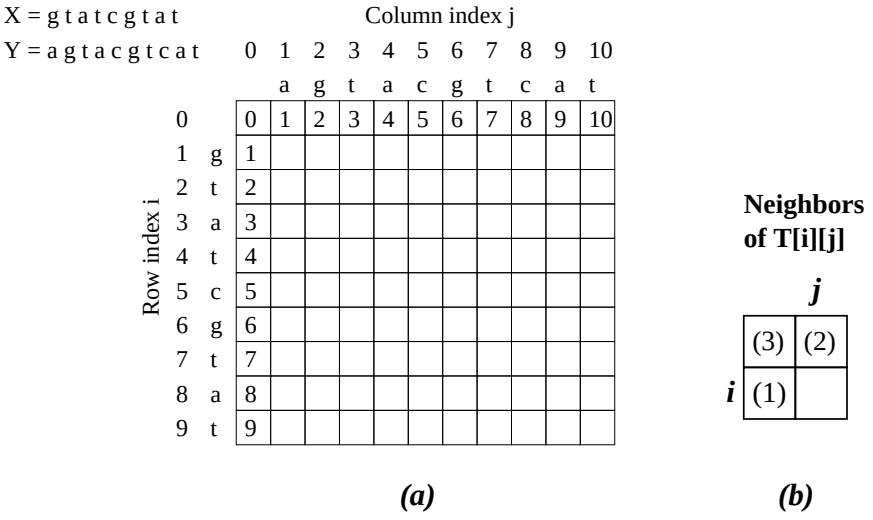


Figure 1: The initial dynamic programming table.

Computing the Table

- The table values should be computed row by row starting from the first row and moving down to the bottom row.
- Each row is computed from left to right.

- To compute a new cell, three neighbors must already be computed (see Figure 1*(b)*): (1) the left neighbor, (2) the top neighbor, and (3) the diagonal top-left neighbor.

Recall that $T[i][j]$ is the edit distance between $X[0, \ldots, i\text{-}1]$ and $Y[0, \ldots, j\text{-}1]$. If the two symbols that correspond to $T[i][j]$ are equal, then no deletion is needed. Thus, the new edit distance is equal to the edit distance between $X[0, \ldots, i\text{-}2]$ and $Y[0, \ldots, j\text{-}2]$, which was previously computed and stored in $T[i\text{-}1][j\text{-}1]$.

If the two symbols are not equal then a deletion must be performed. To determine which symbol to delete, we must examine the edit distance stored in the top and left neighbors. If the top neighbor has a smaller edit distance then the symbol along the left should be deleted. If the left neighbor has a smaller edit distance then the symbol along the top should be deleted. This decision ensures that we choose the proper deletions to minimize the edit distance.

Computing T[i][j]

- Each cell in the table corresponds to two symbols: the symbol along the top row and the symbol along the left column.
- If the two symbols are equal, we copy the diagonal top-left value to the new cell.
- If the two symbols are not equal, we examine the top cell and the left cell. We choose the cell with the smaller value, we add one to that value, and we copy the incremented value to the new cell.

Figure 2 shows the entire computed table. The purpose of the entire table is to arrive at the value $T[m][n]$ in the bottom right corner, which represents the edit distance between the two strings.

		Column index j										
		0	1	2	3	4	5	6	7	8	9	10
			a	g	t	a	c	g	t	c	a	t
0		0	1	2	3	4	5	6	7	8	9	10
1	g	1	2	1	2	3	4	5	6	7	8	9
2	t	2	3	2	1	2	3	4	5	6	7	8
3	a	3	2	3	2	1	2	3	4	5	6	7
4	t	4	3	4	3	2	3	4	3	4	5	6
5	c	5	4	5	4	3	2	3	4	3	4	5
6	g	6	5	4	5	4	3	2	3	4	5	6
7	t	7	6	5	4	5	4	3	2	3	4	5
8	a	8	7	6	5	4	5	4	3	4	3	4
9	t	9	8	7	6	5	6	5	4	5	4	3

Row index i

Figure 2: The computed dynamic programming table.

Dynamic programming saves an incredible amount of computation by recording the edit distance between the prefixes of the strings, rather than recomputing them as needed. The naive algorithm we first described implicitly recomputes this information over and over again.

The problem with dynamic programming is that the table requires a lot of memory. Given two input strings of length n and m, the size of the table would be $(n+1)(m+1)$. For example, if you were given two input strings that were each one million symbols long, the table would contain 10^{12} cells, which is too large to store in memory.

However, storing the entire table is not necessary. In fact, we can compute the final edit distance by storing only two rows of the table. If we align a string of length n along the top of the table, the amount of memory required is $2(n+1)$. This is described as a **linear memory algorithm** because the amount of memory is linearly proportional to the length of the input strings, which is a very nice feature.

Computing the LCS

After computing the edit distance, we can backtrack over the table to determine the exact deletions required to compute the LCS.

- Start from the bottom right cell of the table.
- If the symbols are equal, push the symbol on to a stack and move to the diagonal top-left neighbor.
- If the symbols do not match, compare the top and left neighbor and move to the one with the smaller value. **If there is a tie then you should always choose the left neighbor.** This is to ensure consistency with our convention, so that your algorithm's solution will match ours.
- Continue backtracking through the table and always push symbols on to the stack if they are equal.
- The backtracking will stop when you reach the top or left boundary of the table. After you are done, pop the symbols off the stack to produce the LCS.

Figure 3 shows the backtracking needed to produce the LCS. The highlighted symbols represent the LCS, which is *gtacgtat*. The backtracking needed to produce the LCS requires that the entire table be stored in memory since the algorithm will likely backtrack over every row.

		Column index j										
		0	1	2	3	4	5	6	7	8	9	10
			a	**g**	**t**	**a**	**c**	**g**	**t**	c	**a**	**t**
0		0	(1)	2	3	4	5	6	7	8	9	10
1	**g**	1	2	(1)	2	3	4	5	6	7	8	9
2	**t**	2	3	2	(1)	2	3	4	5	6	7	8
3	**a**	3	2	3	2	(1)	2	3	4	5	6	7
4	t	4	3	4	3	(2)	3	4	3	4	5	6
5	**c**	5	4	5	4	3	(2)	3	4	3	4	5
6	**g**	6	5	4	5	4	3	(2)	3	4	5	6
7	**t**	7	6	5	4	5	4	3	(2)	(3)	4	5
8	**a**	8	7	6	5	4	5	4	3	4	(3)	4
9	**t**	9	8	7	6	5	6	5	4	5	4	(3)

Row index i

Figure 3: The computed dynamic programming table.

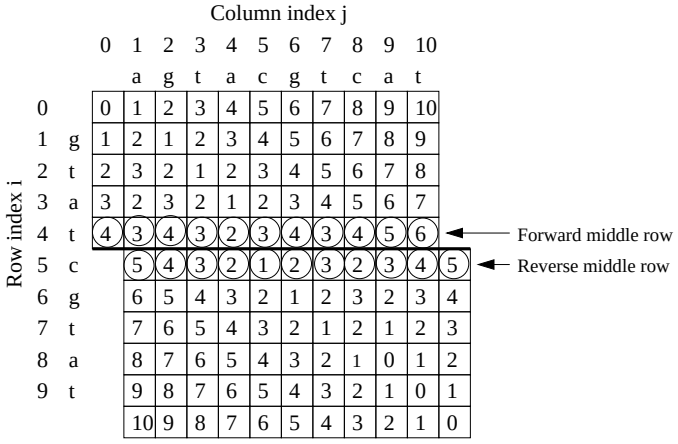
Computing the LCS using Linear Memory

We can use a clever trick to actually compute the LCS without storing the entire table by using a divide and conquer approach. Rather than compute the LCS by backtracking over the entire table, we can actually break up the table into very small chunks and compute the LCS recursively. Throughout this process, we never have to store more than two rows of the table.

- First, we start by computing rows from top to bottom. Each row is computed left to right as we did in the previous algorithms for computing edit distance. Again, we can use only two rows, so that the entire table does not need to be stored. However, instead of computing all the rows, we stop at the middle row, *i.e.* row $\lfloor \frac{m}{2} \rfloor$ where m is the length of the string aligned along the left. This row is called the **forward middle row** because it is computed in the **forward direction**.

- Second, we compute the other rows from bottom to top. Each of these rows is computed right to left (**reverse direction**). Note that the cells three neighbors are now inverted. Before a new cell can be computed, its right neighbor, bottom neighbor, and diagonal bottom-right neighbor must be computed. Again, instead of computing all the rows, we stop at the bottom middle row, *i.e.* row $\lfloor \frac{m}{2} \rfloor + 1$. This row is called the **reverse middle row**.

Figure 4 shows the computation of the two **middle rows**. Note how the bottom rows are computed in the **reverse direction** and note that the rightmost column and bottom row are initialized in reverse order.



		0	1	2	3	4	5	6	7	8	9	10
			a	g	t	a	c	g	t	c	a	t
0		0	1	2	3	4	5	6	7	8	9	10
1	g	1	2	1	2	3	4	5	6	7	8	9
2	t	2	3	2	1	2	3	4	5	6	7	8
3	a	3	2	3	2	1	2	3	4	5	6	7
4	t	4	3	4	3	2	3	4	3	4	5	6
5	c		5	4	3	2	1	2	3	2	3	4	5
6	g		6	5	4	3	2	1	2	3	2	3	4
7	t		7	6	5	4	3	2	1	2	1	2	3
8	a		8	7	6	5	4	3	2	1	0	1	2
9	t		9	8	7	6	5	4	3	2	1	0	1
			10	9	8	7	6	5	4	3	2	1	0

Figure 4: Computing the table in two directions.

After computing the middle rows, the next step is to scan the middle rows cell by cell and figure out exactly where the LCS would have backtracked. We can do this by adding the cell value in the forward middle row to the value in the diagonal bottom-right cell of the reverse middle row. We scan from left to right and remember the **last** location of the minimum edit distance. Figure 5 shows how to use the middle rows to compute the edit distance.

Note that it is important to scan the middle rows cell by cell from left to right and record the LAST location of the minimum edit distance. Otherwise your algorithm will not produce a correct LCS. This location actually tells us where the backtracking will occur. We can use this location to split the table into four quarters and we can discard two of the quarters

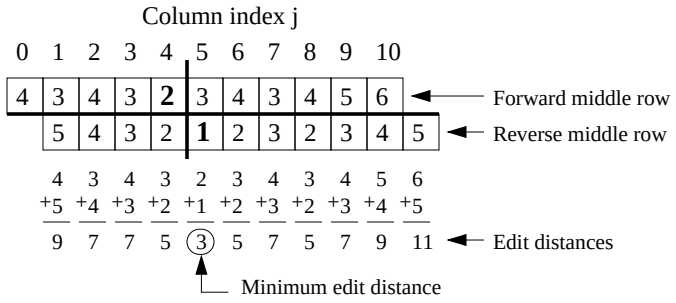


Figure 5: Computing the edit distance from the middle rows.

because we know that the LCS will not backtrack over these cells. Figure 6 shows how the table is split and shows the discarded entries. Highlighted is the actual backtracking path of the LCS.

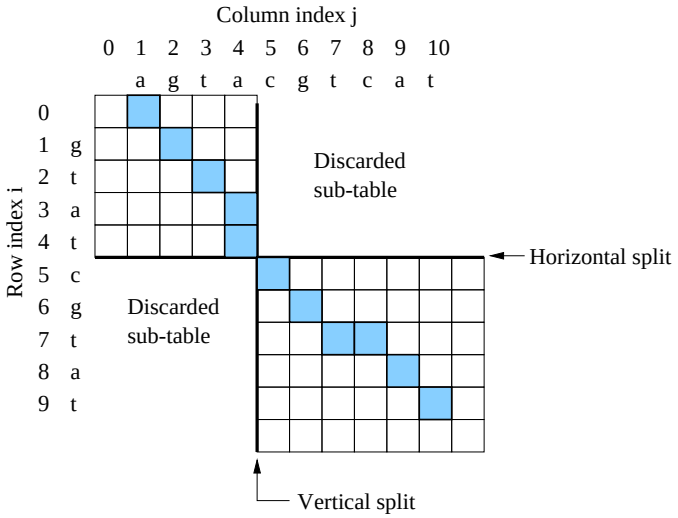


Figure 6: Splitting the table into two quarter tables.

To compute the LCS we only need to examine the two quarter tables shown above. Note that the table is always horizontally split in the middle ($\lfloor \frac{m}{2} \rfloor$), but the vertical split depends on where the minimum edit distance appears. This allows us to determine where the LCS will backtrack.

After we have split the table into two quarter tables, we can apply the same algorithm to the two quarter tables and break them into even smaller quarters. We can recursively apply this process until all the sub-tables are broken into tables which have either one row or one column. Figure 7 illustrates how the table might be recursively split into sub-tables with either one row or one column.

Finally, to compute the LCS we need to report if the symbol in the one remaining column matches any of the symbols in the remaining rows, or if the symbol in the one remaining row matches any of the symbols in the remaining columns. This is the **base case** of the recursive algorithm. The purpose of the divide and conquer recursive algorithm is to break the table into these base cases.

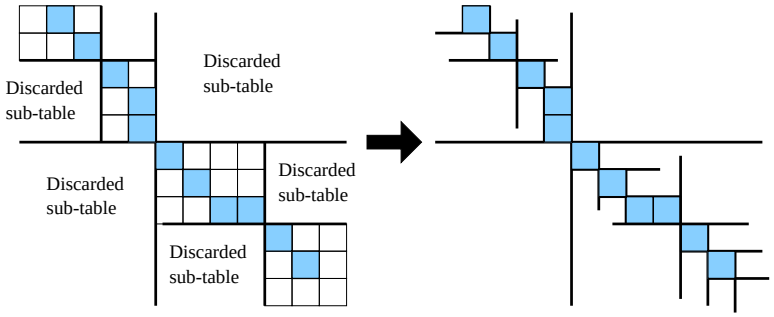


Figure 7: Splitting the table into single rows and single columns.

This complicated approach has a very simple recursive algorithm that ensures that the LCS is computed in the proper order.

```
void lcs_recursive(const Sequence & X, const Sequence & Y, Sequence & LCS) {
   if (X.size() == 1) {
      Compare X[0] to each symbol in Y[0, ..., Y.size()-1]
      If there is a symbol match push_back X[0] on to LCS
   }
   else if (Y.size() == 1) {
      Compare Y[0] to each symbol in X[0, ..., X.size()-1]
      If there is a symbol match push_back Y[0] on to LCS
   }
   else {
      Compute the middle rows as described above.
      Find the horizontal (x) and vertical (y) split indices for the table.
      Generate Sequence X_front = X[0, ..., x]
      Generate Sequence Y_front = Y[0, ..., y]
      Generate Sequence X_back = X[x+1, ..., X.size()-1]
      Generate Sequence Y_back = Y[y+1, ..., Y.size()-1]
      lcs_recursive(X_front, Y_front, LCS);
      lcs_recursive(X_back, Y_back, LCS);
   }
}
```

- The horizontal split index (x) corresponds to the index of string X where the table is split horizontally. In the example shown in Figure 6 the horizontal split index would be three (NOT four, four would be the row index). This corresponds to symbol t. This should always be equal to X.size()/2 - 1. The minus one is necessary because our arrays start with index zero.

- The vertical split index (y) corresponds to the index of string Y where the table is split vertically. In the example shown in Figure 6 the horizontal split index would be three (NOT four, four would be the column index). This corresponds to symbol a.

Objectives: What you must do

1. Implement the algorithm for normalized edit distance.
   This function takes two sequences and computes the normalized edit distance between them. **This function should use only two rows so that the table does not need to be stored in memory.**

2. Implement the LCS algorithm that stores the entire table.
   This function computes and returns the longest common subsequence of two sequences. **This function should be implemented using the basic dynamic programming algorithm that stores the entire table in memory.**

3. Implement the LCS algorithm that uses linear memory.
   This function computes and returns the longest common subsequence of two sequences. **This function should be implemented using the recursive linear memory algorithm.**

Note: Follow the convention that the sequence specified as the first argument in the above functions is placed at the left of the dynamic programming table and the second argument is placed at the top of the dynamic programming table.

Grading

- **Normalized Edit Distance Algorithm (20 points)**
- **LCS Algorithm – stores the entire table (20 points)**
- **LCS Algorithm – linear memory recursive version (40 points)**
- **Design, documentation, and compilation (20 points):** Your algorithms must be well-designed and structured. Your code must be well-commented, otherwise you will receive a 5 point penalty. Your algorithms should be carefully and efficiently implemented.

Submission Guidelines

Your submission must include all your source code and a brief report as a `README` file. Your submission should NOT include any IDE-specific project files, any compiled files, or any executables. Every file should have your name in a comment line at the top. Your `README` file should have a brief description of your program design, the breakdown of the files, which compiler you used, a summary of what you think works and fails in your program, **and** a short description of your data structure design.

You will submit your project on Moodle. You should submit a single zip or tar file containing your source code files and README file. You can submit your project multiple times; only the most recent project submission will be graded. The most recent project submission will also be used to compute late days, if any.

Final Warning

A project that does not follow the submission guidelines will receive a 10 point deduction. Proper submission is entirely **your responsibility**. Contact the TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to

keep copies of your files and **do not change them after submitting**. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.

Please observe the academic integrity guidelines in the syllabus, and submit your own work.

Acknowledgments: This programming assignment was originally designed by Eric Breimer.