PS5: DNA Sequence Alignment

Global Sequence Alignment

We will:

- Write a program to compute the optimal sequence alignment of two DNA strings. This program will introduce you to the field of computational biology in which computers are used to do research on biological systems. Further, you will be introduced to a powerful algorithmic design paradigm known as dynamic programming.
- install **`Valgrind`**, a memory analysis tool
- measure and report the space and time performance of the implementation

Understanding the Problem

Biology review

A genetic sequence is a string formed from a four-letter alphabet {Adenine (A), Thymine (T), Guanine (G), Cytosine (C)} of biological macromolecules referred to together as the DNA bases. A gene is a genetic sequence that contains the information needed to construct a protein. All of your genes taken together are referred to as the human genome, a blueprint for the parts needed to construct the proteins that form your cells. Each new cell produced by your body receives a copy of the genome. This copying process, as well as natural wear and tear, introduces a small number of changes into the sequences of many genes. Among the most common changes are the substitution of one base for another and the deletion of a substring of bases; such changes are generally referred to as point mutations. As a result of these point mutations, the same gene sequenced from closely related organisms will have slight differences.

The problem

Through your research you have found the following sequence of a gene in a previously unstudied organism.

A A C A G T T A C C

What is the function of the protein that this gene encodes? You could begin a series of uninformed experiments in the lab to determine what role this gene plays. However, there is a good chance that it is a variant of a known gene in a previously studied organism. Since biologists and computer scientists have laboriously determined (and published) the genetic sequence of many organisms (including humans), you would like to leverage this information to your advantage. We'll compare the above genetic sequence with one which has already been sequenced and whose function is well understood.

T A A G G T C A

If the two genetic sequences are similar enough, we might expect them to have similar functions. We would like a way to quantify "similar enough."

Edit-distance

In this assignment we will measure the similarity of two genetic sequences by their edit distance, a concept first introduced in the context of coding theory, but which is now widely used in spell checking, speech recognition, plagiarism detection, file revisioning, and computational linguistics. We align the two sequences, but we are permitted to insert gaps in either sequence (e.g., to make them have the same length). We pay a penalty for each gap that we insert and

also for each pair of characters that mismatch in the final alignment. Intuitively, these penalties model the relative likeliness of point mutations arising from deletion/insertion and substitution. We produce a numerical score according to the following table, which is widely used in biological applications:

operation	cost
insert a gap	2
align two characters that mismatch	1
align two characters that match	0

Here are two possible alignments of the strings **x =** "**AACAGTTACC**" and **y =** "**TAAGGTCA**":

x	**y**	**cost**
A	T	1
A	A	0
C	A	1
A	G	1
G	G	0
T	T	0
T	C	1
A	A	0
C	–	2
C	–	2

		8

x	**y**	**cost**
A	T	1
A	A	0
C	–	2
A	A	0
G	G	0
T	G	1
T	T	0
A	–	2
C	C	0
C	A	1

		7

The first alignment has a score of 8, while the second one has a score of 7. The edit-distance is the score of the best possible alignment between the two genetic sequences over all possible alignments. In this example, the second alignment is in fact optimal, so the edit-distance between the two strings is 7. Computing the edit-distance is a nontrivial computational problem because we must find the best alignment among exponentially many possibilities. For example, if both strings are 100 characters long, then there are more than 10^{75} possible alignments.

A recursive solution is an elegant approach. However, it is far too inefficient because it recalculates each subproblem over and over. Once we have defined the recursive definition we can redefine the solution using a dynamic programming approach which calculates each subproblem once.

A recursive solution

We will calculate the edit-distance between the two original strings x and y by solving many edit-distance problems on smaller suffixes of the two strings. We use the notation **`x[i]`** to refer to character **`i`** of the string. We also use the notation **`x[i..M]`** to refer to the suffix of **`x`** consisting of the characters **`x[i], x[i+1], ..., x[M-1]`**. Finally, we use the notation **`opt[i][j]`** to denote the edit distance of **`x[i..M]`** and **`y[j..N]`**. For example, consider the two strings **`x =`** "**AACAGTTACC**" and **`y`** = "**TAAGGTCA**" of length **`M = 10`** and **`N = 8`**, respectively. Then, **`x[2]`** is '**C**', **`x[2..M]`** is "**CAGTTACC**", and **`y[8..N]`** is the empty string. The edit distance of **`x`** and **`y`** is **`opt[0][0]`**.

Now we describe a recursive scheme for computing the edit distance of **`x[i..M]`** and **`y[j..N]`**. Consider the first pair of characters in an optimal alignment of **`x[i..M]`** with **`y[j..N]`**. There are three possibilities:

1. The optimal alignment matches **`x[i]`** up with **`y[j]`**. In this case, we pay a penalty of either **`0`** or **`1`**, depending on whether **`x[i]`** equals **`y[j]`**, plus we still need to align **`x[i+1..M]`** with **`y[j+1..N]`**. What is the best way to do this? This subproblem is exactly the same as

the original sequence alignment problem, except that the two inputs are each suffixes of the original inputs. Using our notation, this quantity is **`opt[i+1][j+1]`**.

2. The optimal alignment matches the **`x[i]`** up with a gap. In this case, we pay a penalty of **`2`** for a gap and still need to align **`x[i+1..M]`** with **`y[j..N]`**. This *subproblem* is identical to the original sequence alignment problem, except that the first input is a proper suffix of the original input.
3. The optimal alignment matches the **`y[j]`** up with a gap. In this case, we pay a penalty of **`2`** for a gap and still need to align **`x[i..M]`** with **`y[j+1..N]`**. This *subproblem* is identical to the original sequence alignment problem, except that the second input is a proper suffix of the original input.

The key observation is that all of the resulting *subproblems* are sequence alignment problems on suffixes of the original inputs. To summarize, we can compute **`opt[i][j]`** by taking the minimum of three quantities:

```
opt[i][j] = min { opt[i+1][j+1] + 0/1, opt[i+1][j] + 2, opt[i][j+1] + 2 }
```

This equation works assuming **`i < M`** and **`j < N`**. Aligning an empty string with another string of length **`k`** requires inserting k gaps, for a total cost of **`2k`**. Thus, in general we should set **`opt[M][j] = 2(N-j)`** and **`opt[i][N] = 2(M-i)`**. For our example, the final matrix is:

x\y		0	1	2	3	4	5	6	7	8
		T	A	A	G	G	T	C	A	–
0	A	7	8	10	12	13	15	16	18	20
1	A	6	6	8	10	11	13	14	16	18
2	C	6	5	6	8	9	11	12	14	16
3	A	7	5	4	6	7	9	11	12	14
4	G	9	7	5	4	5	7	9	10	12
5	T	8	8	6	4	4	5	7	8	10
6	T	9	8	7	5	3	3	5	6	8
7	A	11	9	7	6	4	2	3	4	6
8	C	13	11	9	7	5	3	1	3	4
9	C	14	12	10	8	6	4	2	1	2
10	–	16	14	12	10	8	6	4	2	0

By examining **`opt[0][0]`**, we conclude that the edit distance of **`x`** and **`y`** is **`7`**.

A dynamic programming approach

A direct implementation of the above recursive scheme will work, but it is spectacularly inefficient. If both input strings have ***N*** characters, then the number of recursive calls will exceed 2^N. To overcome this performance bug, we use ***dynamic programming***. Dynamic programming is a powerful algorithmic paradigm, first introduced by **Bellman** in the context of operations research, and then applied to the alignment of biological sequences by **Needleman** and **Wunsch**. Dynamic programming now plays the leading role in many computational problems, including control theory, financial engineering, and bioinformatics, including BLAST (the sequence alignment program almost universally used by molecular biologists in their experimental work). The key idea of dynamic programming is to break up a large computational problem into smaller subproblems, store the answers to those smaller subproblems, and, eventually, use the stored answers to solve the original problem. This avoids recomputing the same quantity over and over again. Instead of using recursion, use a nested loop that calculates **`opt[i][j]`** in the right order so that **`opt[i+1][j+1]`**, **`opt[i+1][j]`**, and **`opt[i][j+1]`** are all computed before we try to compute **`opt[i][j]`**.

Recovering the alignment itself

The above procedure describes how to compute the edit distance between two strings. We now outline how to recover the optimal alignment itself. The key idea is to retrace the steps of the dynamic programming algorithm backwards, re-discovering the path of choices (highlighted in red in the table above) from **`opt[0][0]`** to **`opt[M][N]`**. To determine the choice that led to **`opt[i][j]`**, we consider the three possibilities:

Case 1. The optimal alignment matches **`x[i]`** up with **`y[j]`**. In this case, we must have **`opt[i][j] = opt[i+1][j+1]`** if **`x[i]`** equals **`y[j]`**, or **`opt[i][j] = opt[i+1][j+1] + 1`** otherwise.

Case 2. The optimal alignment matches **`x[i]`** up with a gap. In this case, we must have **`opt[i][j] = opt[i+1][j] + 2`**.

Case 3. The optimal alignment matches **`y[j]`** up with a gap. In this case, we must have **`opt[i][j] = opt[i][j+1] + 2`**.

Depending on which of the three cases apply, we move diagonally, down, or right towards **`opt[M][N]`**, printing out **`x[i]`** aligned with **`y[j]`** (case 1), **`x[i]`** aligned with a gap (case 2), or **`y[j]`** aligned with a gap (case 3). In the example above, we know that the first **`T`** aligns with the first **`A`** because **`opt[0][0] = opt[1][1] + 1`**, but **`opt[0][0] ≠ opt[1][0] + 2`** and **`opt[0][0] ≠ opt[0][1] + 2`**. The optimal alignment is:

x	y	cost
A	T	1
A	A	0
C	–	2
A	A	0
G	G	0
T	G	1
T	T	0
A	–	2
C	C	0
C	A	1

Implementation

You may elect to implement any of four solution approaches:

- Recursive without memoization (too slow to be practical, but order N in space)
- Recursive using memoization
- Dynamic programming using an **N×M** matrix, the Needleman-Wunsch method
- Using Hirschberg's algorithm, which is linear in space

If you implement the dynamic programming with a matrix approach, remember that the solution is in two parts:

1. Filling out the **N×M** matrix per the min-of-three-options formula, bottom to top, right to left (this gives you the optimal edit distance in the upper-left, [0][0] cell of the matrix);
2. Traversing the matrix from top-to-bottom, left-to-right (i.e., [0][0] to [N][M]) to recover the choices you made in filling it, and thereby also recovering the actual edit sequence.

API specification

You should create a class file **ED** (for "Edit Distance") with the following methods:

- A **constructor** that accepts the two strings to be compared, and allocates any data structures necessary into order to do the work (e.g., the **N×M** matrix).
- A **static** method **`int penalty(char a, char b)`** that returns the penalty for aligning chars **a** and **b** (this will be a **0** or a **1**).
- A **static** method **`int min(int a, int b, int c)`** which returns the minimum of the three arguments.
- A method **`int OptDistance()`** which populates the matrix based on having the two strings, and returns the optimal distance (from the **[0][0]** cell of the matrix when done).
- A method **`string Alignment()`** which traces the matrix and returns a string that can be printed to display the actual alignment. In general, this will be a multi-line string — i.e., with embedded **\n**'s.

You should have a **main** routine that accepts the two strings from **stdin**, uses your **ED** (Edit Distance) class to do the work, and then prints the result to **stdout**. Remember that your final output should look like this:

```
% ./ED < example10.txt
Edit distance = 7
A T 1
A A 0
C - 2
A A 0
G G 0
T G 1
T T 0
A - 2
C C 0
C A 1
```

Implementation

You have to allocate the memory for the opt matrix dynamically, after you read in the two strings and figure out how long they are. There are a few different ways to do this:

- vector of columns, each containing a row vector

- one long vector, with internal calculations to treat it as a matrix
- one big block of memory, similarly with calculations to treat it as a matrix
- others?

NOTE: See http://stackoverflow.com/questions/936687/how-do-i-declare-a-2d-array-in-c-using-new for some ideas.

If you use `new`, make sure to de-allocate memory in your class destructor; e.g., if your array pointer is named `_opt`, then `delete [] _opt;`

- The dynamic programming solution we discussed requires filling the whole matrix with values (step 1 of the two-part solution), so it's **N**-squared in space (or more precisely, **N×M**).

You shouldn't expect to have your code work for the test case with two `500,000` `char` strings. Assuming you use a 32-bit int to hold edit distance values in your matrix, that's 2 MB of data squared, or `4,000` GB. Probably your computer can't allocate this much RAM.

The largest problem you should be able to handle is in `ecoli28284.txt`. This should cause you to allocate an array of approx. 800 million values. Assuming you're using 4-byte `int`s, that's 3.2 GB of data. Don't worry about larger cases unless you want to explore alternate approaches. If so, there is a solution which computes the optimal alignment in *linear space* (and quadratic time). This is known as *Hirschberg's algorithm* (1975).

After you have things working, add code to calculate and print execution time. You may use SFML's `sf::Clock` and `sf::Time` classes, as follows:

- To your `main` routine, `#include <SFML/System.hpp>` at the top.
- Then define the following two objects in your `main`:

```
sf::Clock clock;
sf::Time t;
```

- At the end of `main`, after computing the solution, capture the running time:

```
t = clock.getElapsedTime();
```

- Then after printing out the solution, display the running time:

```
cout << "Execution time is " << t.asSeconds() << " seconds \n";
```

Valgrind

Verify your algorithm's space usage with the Valgrind runtime analysis tool.

Install Valgrind:

```
sudo apt-get install valgrind
```

- Try different compiler optimization flags to see how execution time is affected. E.g., `-O1` or `-O2` could halve running time over using no optimization.
- Make sure your code is compiled and linked with debugging information (`-g` flag).
- Use **Valgrind** to run your code with the "massif" heap analysis tool; e.g.:
  `valgrind --tool=massif ./ED < ../sequence/ecoli28284.txt`

- Then Valgrind will produce a log file named `massif.out.XXXXX`, where `XXXXX` is the process ID from your run. View with file with the `ms_print` utility that's part of the Valgrind distribution; e.g.: `ms_print massif.out.11515 | less`
- By examining the `massif.out` file, confirm that the amount of memory used matches your expectations.
- Also, you can use the program `massif-visualizer` to view the logs (see http://milianw.de/tag/massif-visualizer) -- install with `sudo apt-get install massif-visualizer`.
- More documentation on **Valgrind** and massif is here: http://valgrind.org/docs/manual/ms-manual.html.

Making the autograder happy

The autograder will compile your code and check it with the following input files:

`bothgaps20.txt` `ecoli2500.txt` `ecoli7000.txt` `example10.txt` `fli8.txt`

`ecoli10000.txt` `ecoli5000.txt` `endgaps7.txt` `fli10.txt` `fli9.txt`

To make sure this works:

- Remember to add `-lsfml-system` to your compile and link commands in your `Makefile`.
- Your `Makefile` must produce an executable named `ED`.
- Your program must accept input from `stdin` (e.g., using the `<` redirect).
- Your program must print out **first** the edit distance and **then** the edit path.
- You must follow the format (see example above) exactly, and you can't have trailing spaces at the ends of the lines.
- You should print the elapsed time after all this. These lines will be ignored.
- Everything must be in a directory named ps5, and per usual, make sure to remove object files before tarring up your work.

Reporting your results

Please fill out your data in the **`ps5-readme.txt`** file.

Notes:

- CPU speed in MHz
- input-size is the length of the problem string, for these test problems:
  - ecoli2500.txt
  - ecoli5000.txt
  - ecoli7000.txt
  - ecoli10000.txt
  - ecoli20000.txt
  - ecoli28284.txt
- time-to-solution is in seconds
- memory-used is in MB
- Implementation method is one of: recursive, recursive-with-memoization, Needleman-Wunsch, Hirschberg, other
- array-method is one of: vectors, c-arrays, hash-table, other
- operating-system is one of: x86-unix-native, mac-os-x, lubuntu, other
- cpu-type — e.g., core i3, core i7, AMD (and model), etc.

Submitting

Put all of your work into a directory named `ps5: all` of your code files, `Makefile`, and the completed `ps5-readme.txt` file. ***Make sure to run `make clean` before archiving*.**

- Submit on `Blackboard`

Grading rubric

Feature	Value	Comment
core implementation	6	3 pts for filling the matrix and getting the edit distance 3 pts for traversing the matrix to retrieve the path
Makefile	1	Makefile included
		targets `all` and `clean` must exist `all` should build `ED`
`readme`	7	1pt Discussion of algorithm 2 pts Performance time & space 2 pts Valgrind analysis discussion 1 pt largest N mem calc 1 pt largest N time calc
Total	**14**	
extra credit	2	Use of the `lambda` expression