

Introduction.

The purpose of this project is to write a program to compute the optimal sequence alignment of two DNA strings. This program will introduce you to the emerging 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*.

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.

Problem.

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

A A C A G T T A 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:

Penalty	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 $x = \text{'AACAGTTACC'}$ and $y = \text{'TAAGGTCA'}$:

Sequence 1	A	A	C	A	G	T	T	A	C	C
Sequence 2	T	A	A	G	G	T	C	A	-	-
Penalty	1	0	1	1	0	0	1	0	2	2

Sequence 1	A	A	C	A	G	T	T	A	C	C
Sequence 2	T	A	-	A	G	G	T	-	C	A
Penalty	1	0	2	0	0	1	0	2	0	1

The first alignment has a score of 8 (total penalty), 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.

We will explain a recursive solution that is an elegant approach. However, it is far too inefficient because it recalculates each sub-problem over and over. Once we have defined the recursive definition we can redefine the solution using a dynamic programming approach which calculates each sub-problem 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 $x[i]$, $x[i + 1]$, \dots , $x[M - 1]$. Finally, we use the notation $\text{opt}[i][j]$ to denote the edit distance of $x[i..M]$ and $y[j..N]$. For example, consider the two strings $x = \text{'AACAGTTACC'}$ and $y = \text{'TAAGGTCA'}$ 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 $\text{opt}[0][0]$.

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 sub-problem 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 $\text{opt}[i + 1][j + 1]$.
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 sub-problem 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 sub-problem 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 sub-problems are sequence alignment problems on suffixes of the original inputs. To summarize, we can compute $\text{opt}[i][j]$ by taking the minimum of three quantities:

$$\text{opt}[i][j] = \min\{\text{opt}[i+1][j+1] + 0/1, \text{opt}[i+1][j] + 2, \text{opt}[i][j+1] + 2\}$$

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 $\text{opt}[M][j] = 2(N-j)$ and $\text{opt}[i][N] = 2(M-i)$. For our example, the final matrix is:

			0	1	2	3	4	5	6	7	8
x\y		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 $\text{opt}[0][0]$, we conclude that the edit distance of x and y is 7.

Problem 1. (Calculating Edit Distance Using Dynamic Programming)

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 biologist in their experimental work). The key idea of dynamic programming is to break up a large computational problem into smaller sub-problems, *store* the answers to those smaller sub-problems, and, eventually, use the stored answers to solve the original problem. This avoids re-computing 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]`.

Your Task.

Write a program `edit_distance.py` that reads strings `x` and `y` from standard input and computes the edit-distance matrix `opt` as described above. The program should output `x`, `y`, the dimensions (number of rows and columns) of `opt`, and `opt` itself, using the following format:

- The first and second lines should contain the strings `x` and `y`.
- The third line should contain the dimensions of the `opt` matrix, separated by a space.
- The following lines should contain the rows of the `opt` matrix, each ending in a newline character.

Use `stdio.write()` with the format string `'%3d'` to write out the elements of the matrix.

```
$ python2.7 edit_distance.py < data/example10.txt
AACAGTTACC
TAAGGTCA
11 9
 7  8 10 12 13 15 16 18 20
6  6  8 10 11 13 14 16 18
6  5  6  8  9 11 12 14 16
7  5  4  6  7  9 11 12 14
9  7  5  4  5  7  9 10 12
8  8  6  4  4  5  7  8 10
9  8  7  5  3  3  5  6  8
11 9  7  6  4  2  3  4  6
13 11 9  7  5  3  1  3  4
14 12 10 8  6  4  2  1  2
16 14 12 10 8  6  4  2  0
```

Hints for Problem 1.

- Read the sequences x and y from standard input, as strings
- Create an $(M + 1) * (N + 1)$ edit-distance matrix opt with all elements initialized to 0, where M and N are the lengths of x and y respectively
- Set the bottom row of opt to $2 * (N - j)$ and its right column to $2 * (M - i)$, where $0 \leq j \leq N - 1$ and $0 \leq i \leq M - 1$
- For example, if $x = \text{'HAM'}$ ($M = 3$) and $y = \text{'SPAM'}$ ($N = 4$), then the corresponding opt matrix after the above step is

			0	1	2	3	4
	$x \backslash y$		S	P	A	M	-
0	H		0	0	0	0	6
1	A		0	0	0	0	4
2	M		0	0	0	0	2
3	-		8	6	4	2	0

- Fill in the rest of the opt matrix, starting at $opt[M - 1][N - 1]$ and ending at $opt[0][0]$, as follows: if $x[i]$ and $y[j]$ are the same, where $0 \leq i \leq M-1$ and $0 \leq j \leq N-1$, then
 $opt[i][j] = \min(opt[i+1][j+1], opt[i+1][j]+2, opt[i][j+1]+2)$ and
 $opt[i][j] = \min(opt[i+1][j+1]+1, opt[i+1][j]+2, opt[i][j+1]+2)$ otherwise
- The opt matrix for the above example after the preceding step is

			0	1	2	3	4
	$x \backslash y$		S	P	A	M	-
0	H		3	1	2	4	6
1	A		4	2	0	2	4
3	M		6	4	2	0	2
3	-		8	6	4	2	0

- Write the following output, each starting on a new line
 - String x
 - String y
 - Dimensions of the opt matrix separated by a space
 - Elements of opt ; use format string `'%3d'` for elements not on the last column, and `'%3d\n'` for the last-column elements

Problem 2. (Recovering the Alignment)

Now that we know how to compute the edit distance between two strings, we next want 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 $\text{opt}[0][0]$ to $\text{opt}[M][N]$. To determine the choice that led to $\text{opt}[i][j]$, we consider the three possibilities:

- The optimal alignment matches $x[i]$ up with $y[j]$. In this case, we must have
 $\text{opt}[i][j] = \text{opt}[i+1][j+1]$ if $x[i]$ equals $y[j]$, or
 $\text{opt}[i][j] = \text{opt}[i+1][j+1] + 1$ otherwise.
- The optimal alignment matches $x[i]$ up with a gap. In this case, we must have
 $\text{opt}[i][j] = \text{opt}[i+1][j] + 2$.
- The optimal alignment matches $y[j]$ up with a gap. In this case, we must have
 $\text{opt}[i][j] = \text{opt}[i][j+1] + 2$.

Depending on which of the three cases apply, we move diagonally, down, or right towards $\text{opt}[M][N]$, 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 $\text{opt}[0][0] = \text{opt}[1][1] + 1$, but $\text{opt}[0][0] \neq \text{opt}[1][0] + 2$ and $\text{opt}[0][0] \neq \text{opt}[0][1] + 2$.

The optimal alignment is:

```
A A C A G T T A C C
T A - A G G T - C A
1 0 2 0 0 1 0 2 0 1
```

Your Task.

Write a program `alignment.py` that reads from standard input, the output produced by `edit_distance.py`, i.e., input strings x and y , and the `opt` matrix. The program should then recover an optimal alignment using the procedure described above, and write to standard output the edit distance between x and y and the alignment itself, using the following format:

- The first line should contain the edit distance, preceded by the text 'Edit distance = '.
- Each subsequent line should contain a character from the first string, followed by the paired character from the second string, followed by the associated penalty. Use the character '-' to indicate a gap in either string.

```
$ python2.7 edit_distance.py < data/example10.txt | python2.7 alignment.py
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
```

Hints for Problem 2.

- Read from standard input the sequences x and y as strings, and the matrix opt as a 2D array of integers
- Write the edit distance between x and y , i.e., the value of $opt[0][0]$
- Recover and output the alignment, starting at $opt[0][0]$ and ending at $opt[M - 1][N - 1]$, as follows:
 - if $opt[i][j]$ equals $opt[i + 1][j] + 2$, then align $x[i]$ with a gap and penalty of 2, and increment i by 1;
 - if $opt[i][j]$ equals $opt[i][j + 1] + 2$, then align a gap with $y[j]$ and penalty of 2, and increment j by 1;
 - otherwise, align $x[i]$ with $y[j]$ and penalty of 0/1 based on whether $x[i]$ and $y[j]$ match or not, and increment both i and j by 1

Note: if one of the sequences is exhausted before the other, align a character from the other with a gap and penalty of 2

- For our running example, the optimal alignment produced by the previous step is

```

-  H  A  M
S  P  A  M
2  1  0  0 (edit distance = 3)

```


Analysis and Data.

The data directory contains short test data files and actual genomic data files. Be sure to test your programs thoroughly using the short test files and the longer actual data files. Here are the optimal edit distances of several of the supplied files:

ecoli2500.txt	118
ecoli5000.txt	160
fli8.txt	6
fli9.txt	4
fli10.txt	2
fts1272.txt	758
gene57.txt	8
stx1230.txt	521
stx19.txt	10
stx26.txt	17
stx27.txt	19

The test cases with unique optimal alignments:

```
$ python2.7 edit_distance.py < data/endgaps7.txt | python2.7 alignment.py
Edit distance = 4
a - 2
t t 0
a a 0
t t 0
t t 0
a a 0
t t 0
- a 2
```

```
$ python2.7 edit_distance.py < data/fli10.txt | python2.7 alignment.py
Edit distance = 2
T T 0
G G 0
G G 0
C T 1
G G 0
G G 0
A T 1
A A 0
C C 0
T T 0
```

Acknowledgement:

This assignment was adopted of Global Sequence Alignment by Thomas Clarke, Robert Sedgewick, Scott Vafai and Kevin Wayne. Princeton University