**Edit Distance**

***DNA Sequence Alignment***

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 using a powerful algorithmic design paradigm known as *dynamic programming*. **Read this entire document before proceeding, it will save you time in the long run!**

***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 revision control, 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.

We will explain a recursive solution which is an elegant approach. However it is far too inefficient because it recalculates each sub-problem over and over (you *could* now memorize this approach for much faster results). 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 the *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 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 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 opt[i][j] by taking the minimum of three quantities:

opt[i][j] = min { opt[i+1][j+1] + (0 or 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 2*k*. 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:

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

To get a better understanding about the above matrix, let's look at a few sample entries. First, we initialize the last row and the last column in the above matrix:

* The entry opt[10][8] contains the "minimum" costs (i.e., the edit distance) required to convert string x[10..10] to y[8..8]. The string x[10..10] is an empty string; the string y[8..8] is also an empty string; the minimum cost for converting from an empty string to another empty string is 0.
* The entry opt[10][7] contains the "minimum" costs (i.e., the edit distance) required to convert string x[10..10] to y[8..8]. The string x[10..10] is an empty string; the string y[7..8] is a single-character string A; the minimum cost for converting from an empty string to A is 2 (i.e., by inserting a gap in x[10..10])
* The entry opt[10][4] contains the "minimum" costs (i.e., the edit distance) required to convert string x[10..10] to y[4..8]. The string x[10..10] is an empty string; the string y[4..8] is the string GTCA; the minimum cost for converting from an empty string to the string GTCAis 8 (i.e., by inserting 4 gaps in x[10..10]).
* The entry opt[10][0] contains the "minimum" costs (i.e., the edit distance) required to convert string x[10..10] to y[0..8]. The string x[10..10] is an empty string; the string y[0..8] is the string TAAGGTCA; the minimum cost for converting from an empty string to the string TAAGGTCA is 16 (i.e., by inserting 8 gaps in x[10..10]).
* The entry opt[9][8] contains the "minimum" costs (i.e., the edit distance) required to convert string x[9..10] to y[8..8]. The string x[9..10] is a single-character string C the string y[8..8] is also an empty string; the minimum cost for converting from C to an empty string is 2 (i.e., by inserting a gap in y[8..8])
* The entry opt[4][8] contains the "minimum" costs (i.e., the edit distance) required to convert string x[4..10] to y[8..8]. The string x[4..10] is the string GTTACC the string y[8..8] is the empty string; the minimum cost for converting from GTTACC to an empty string is 12 (i.e., by inserting 6 gaps in y[8..8])
* The entry opt[0][8] contains the "minimum" costs (i.e., the edit distance) required to convert string x[0..10] to y[8..8]. The string x[0..10] is the string AACAGTTACC the string y[8..8] is the empty string; the minimum cost for converting from AACAGTTACC to an empty string is 20 (i.e., by inserting 10 gaps in y[8..8])

Next, we can see how we calculate each entry in the above matrix:

* The entry opt[9][7] contains the "minimum" cost (i.e., the edit distance) required to convert string x[9..10] to y[7..8]. The string x[9..10] is a single character string C; the string y[7..8] is a single character string A; to calculate the minimum costs, there are only 3 possibilities:
* We decide to align x[9] with y[7]; the "minimum" cost for converting x[10..10] to y[8..8] is opt[10][8] which is 0; since x[9] is C which is different from y[7] (i.e., A), the cost of pursuing this path will be 0 + 1 which is 1.
* We decide to insert a gap at the beginning of y[7..8], i.e., we match x[9] with a gap which costs 2; the "minimum" cost for converting x[10..10] to y[7..8] is opt[10][7] which is 2; the cost of pursuing this path will be 2 + 2 which is 4.
* We decide to insert a gap at the beginning of x[9..10], i.e., we match y[7] with a gap which costs 2; the "minimum" cost for converting x[9..10] to y[8..8] is opt[9][8] which is 2; the cost of pursuing this path will also be 2 + 2 which is 4.
  + Clearly, the best option (the minimum of the above three paths) is the first one which costs 1, so opt[9][7] should be 1. This is consistent with what we would get if we calculate such cost manually: the minimum cost for converting the string C to the string A is clearly 1.
* Following similar reasoning, we can calculate the rest of entries, e.g., opt[9][6] is 2; opt[8][7] is 3.
* The entry opt[8][6] contains the "minimum" cost (i.e., the edit distance) required to convert string x[8..10] to y[6..8]. The string x[8..10] is the string CC; the string y[6..8] is the string CA; to calculate the minimum costs, there are again only 3 possibilities:
* We decide to align x[8] with y[6]; the "minimum" cost for converting x[9..10] to y[7..8] is opt[9][7] which is 1; since x[8] is Cwhich is same as y[6] (i.e., C), the cost of pursuing this path will be 1 + 0 which is 1.
* We decide to insert a gap at the beginning of y[6..8], i.e., we match x[8] with a gap which costs 2; the "minimum" cost for converting x[9..10] to y[6..8] is opt[9][6] which is 2; the cost of pursuing this path will be 2 + 2 which is 4.
* We decide to insert a gap at the beginning of x[8..10], i.e., we match y[6] with a gap which costs 2; the "minimum" cost for converting x[8..10] to y[7..8] is opt[8][7] which is 3; the cost of pursuing this path will also be 3 + 2 which is 5.
  + Clearly, the best option (the minimum of the above three paths) is the first one which costs 1, so opt[8][6] should be 1. This is consistent with what we would get if we calculate such cost manually: the minimum cost for converting the string CC to the string CA is indeed 1.
* The entry opt[7][6] contains the "minimum" cost (i.e., the edit distance) required to convert string x[7..10] to y[6..8]. The string x[7..10] is the string ACC; the string y[6..8] is the string CA; to calculate the minimum costs, there are again only 3 possibilities:
* We decide to align x[7] with y[6]; the "minimum" cost for converting x[8..10] to y[7..8] is opt[8][7] which is 3; since x[7] is A which is different from y[6] (i.e., C), the cost of pursuing this path will be 3 + 1 which is 4.
* We decide to insert a gap at the beginning of y[6..8], i.e., we match x[7] with a gap which costs 2; the "minimum" cost for converting x[8..10] to y[6..8] is opt[8][6] which is 1; the cost of pursuing this path will be 1 + 2 which is 3.
* We decide to insert a gap at the beginning of x[7..10], i.e., we match y[6] with a gap which costs 2; the "minimum" cost for converting x[7..10] to y[7..8] is opt[7][7] which is 4; the cost of pursuing this path will also be 4 + 2 which is 6.
  + Clearly, the best option (the minimum of the above three paths) is the second one which costs 3, so opt[7][6] should be 3. This is consistent with what we would get if we calculate such cost manually: the minimum cost for converting the string ACC to the string CA is 3.

***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 2N. 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](http://www.ncbi.nlm.nih.gov/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 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:

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.
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.
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

***API specification.*** Your program **EditDistance.java** must be organized as a library of static methods with the following API:

|  |
| --- |
| **class EditDistance**  -----------------------------------------------------------------------------------------  static int penalty(char a, char b) //return the penalty for aligning char a and char b  static int min(int a, int b, int c) //return the min of 3 integers  static void main(String[] args) //read 2 strings from standard input.  //compute and print the edit distance between them.  //output optimal alignment and associated penalties. |

***Your program.*** Write a program **EditDistance.java** that reads, from standard input, two strings of characters. (Although, in the application described, the characters represent genetic sequences, your program should handle any sequence of alphanumeric characters.) Your program should then compute and print the edit distance between the two strings. Finally, it should recover the optimal alignment and print it out along with the individual penalties, 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.

Here is a sample execution:

% **java EditDistance < 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

There are small test data files (such as example10.txt) as well as many actual genomic data files in the lab folder. Download them as fast as humanly possible.

***Some hints in no particular order.*** These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. Write the main method in **EditDistance.java** to read in the two strings from standard input. For debugging purposes, print them to standard output.
2. Declare and initialize the (M+1) by (N+1) array opt[][]. Include the base cases. Print out the 2D array to check your work.

To print the matrix out in nicely formatted columns, use

System.out.printf("%3d", opt[i][j]);

with nested **for** loops. Remember to remove this debugging print section before submitting the final version of your program.

1. Now, it's time to implement the crucial dynamic programming part. Think carefully about the order in which you will perform the computation since this is critical. **Hint:** first fill in the base case of opt[i][j], e.g., when i = M or j = N. Now, fill in the remaining values using a nested loop. Test that it works by printing out the contents of the **opt** matrix.
2. Now, figure out how to recover the optimal alignment by backtracing.

This is an iterative process. At each step we look to see which path choice we should make. Using the example from above, we start at i = 0, j=0 where x[i] = 'A' and y[j] = 'T'. The choices are to print "A -" and move down with a gap cost of 2, "- T" and move right with a gap cost of 2, or "A T" and move diagonally with a mismatch cost of 1. We know to pick "A T" because 7 - 6 = 1. This is the only choice which matches the matrix. (It is possible to have more than one choice which matches the matrix. In that case, either choice will lead to the same optimal edit distance.)

***Testing.***   To help you check the part of your program that finds the edit distance, the edit distances of gene57.txt, stx1230.txt, and ftsa1272.txt are 8, 521, and 758, respectively. To help you check the part of your program that generates the alignment, there are several short test files in the sequence directory whose alignments are easy to generate manually.