Goal: Write programs to compute an optimal sequence alignment of two DNA strings. These programs will introduce you to the field of computational biology in which computers are used to conduct research on biological systems. Furthermore, you will be introduced to a powerful algorithmic design paradigm known as *dynamic programming*.

Part I: Warmup Problems

The problems in this part of the assignment are intended to give you solid practice on concepts (arrays and standard input) needed to solve the problems in Part II.

Problem 1. (*Reverse*) Write a program called `Reverse.java` that accepts strings from standard input, and writes them in reverse order to standard output.

```
>_ ~/workspace/global_sequence_alignment
$ javac -d out src/Reverse.java
$ java Reverse
b o l t o n
<ctrl-d>
n o t l o b
$ java Reverse
m a d a m
<ctrl-d>
m a d a m
```

Problem 2. (*Euclidean Distance*) Write a program called `Distance.java` that accepts n (int) as command-line argument, two size-n arrays x and y of doubles from standard input, and writes to standard output the Euclidean distance between the two vectors represented by x and y. The Euclidean distance is calculated as the square root of the sums of the squares of the differences between the corresponding entries.

```
>_ ~/workspace/global_sequence_alignment
$ javac -d out src/Distance.java
$ java Distance 2
1 0 0 1 <enter>
1.4142135623730951
$ java Distance 5
-9 1 10 -1 1 -5 9 6 7 4 <enter>
13.0
```

Problem 3. (*Transpose*) Write a program called `Transpose.java` that accepts m (int) and n (int) as command-line arguments, $m \times n$ doubles from standard input representing the elements of an $m \times n$ matrix a, and writes to standard output the transpose of a.

```
>_ ~/workspace/global_sequence_alignment
$ javac -d out src/Transpose.java
$ java Transpose 2 2
1 2 3 4 <enter>
1.0 3.0
2.0 4.0
$ java Transpose 2 3
1 2 3 4 5 6 <enter>
1.0 4.0
2.0 5.0
3.0 6.0
```

Problem 4. (*Strange Matrix*) Write a program called `StrangeMatrix.java` that accepts m (int) and n (int) as command-line arguments, creates an $m \times n$ matrix of integers a, updates its values as follows:

- sets $a(i, n-1)$ (ie, last column of a) to $m - i - 1$, for $0 \leq i < m$,
- sets $a(m-1, j)$ (ie, last row of a) to $n - j - 1$, for $0 \leq j < n$,
- sets $a(i, j)$ to $a(i, j+1) + a(i+1, j+1) + a(i+1, j)$, for $0 \leq i < m - 1$ and $0 \leq j < n - 1$,

and writes the matrix to standard output using `stdio.writef()` with `"%5d "` as the format string for elements *not* in the last column and `"%5d\n"` as the format string for the last-column elements.

```
>_ ~/workspace/global_sequence_alignment

$ javac -d out src/StrangeMatrix.java
$ java StrangeMatrix 4 5
  147    66    27    10     3
   54    27    12     5     2
   17    10     5     2     1
    4     3     2     1     0
```

Problem 5. (*Pascal's Triangle*) Pascal's triangle $\mathcal{P}_n$ is a triangular array with $n+1$ rows, each listing the coefficients of the binomial expansion $(x+y)^i$, where $0 \leq i \leq n$. For example, $\mathcal{P}_4$ is the triangular array:

$$\begin{array}{lllll} 1 \\ 1 & 1 \\ 1 & 2 & 1 \\ 1 & 3 & 3 & 1 \\ 1 & 4 & 6 & 4 & 1 \end{array}$$

The term $\mathcal{P}_n(i,j)$ is calculated as $\mathcal{P}_n(i-1,j-1) + \mathcal{P}_n(i-1,j)$, where $0 \leq i \leq n$ and $1 \leq j < i$, with $\mathcal{P}_n(i,0) = \mathcal{P}_n(i,i) = 1$ for all i. Write a program called `Pascal.java` that accepts n (int) as command-line argument, and writes $\mathcal{P}_n$ to standard output.

```
>_ ~/workspace/global_sequence_alignment

$ javac -d out src/Pascal.java
$ java Pascal 3
1
1 1
1 2 1
1 3 3 1
$ java Pascal 10
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

Part II: Global Sequence Alignment

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: 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 (eg, 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 do not match	1
Align two characters that do match	0

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

```
x     y     cost          x     y     cost
--------------           --------------
A     T      1            A     T      1
A     A      0            A     A      0
C     A      1            C     -      2
A     G      1            A     A      0
G     G      0            G     G      0
T     T      0            T     G      1
T     C      1            T     T      0
A     A      0            A     -      2
C     -      2            C     C      0
C     -      2            C     A      1
            ---                       ---
             8                         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 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.

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 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 `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:

```
        |   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.

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 issue, 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]`.

Problem 6. (*Compute Edit Distance*) Write a program called `EditDistance.java` 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. Write the elements of the matrix using `stdio.writef()` with `"%3d "` as the format string for elements *not* in the last column and `"%3d\n"` as the format string for the last-column elements.

```
>_ ~/workspace/global_sequence_alignment
$ javac -d out src/EditDistance.java
$ java EditDistance < 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
```

Problem 7. (*Recover 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 `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 matched `x[i]` up with a gap. In this case, we must have `opt[i][j] = opt[i + 1][j] + 2`.
2. The optimal alignment matched `y[j]` up with a gap. In this case, we must have `opt[i][j] = opt[i][j + 1] + 2`.
3. The optimal alignment matched `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.

Write a program `Alignment.java` that reads from standard input the output produced by `EditDistance.java`, ie, 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.
- 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.

```
>_ ~/workspace/global_sequence_alignment
$ javac -d out src/Alignment.java
$ java EditDistance < data/example10.txt | java Alignment
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
```

Data: The `data` directory contains some test input (`.txt`) files for the `EditDistance.java` program. The first line in each file is the genome sequence `x` and the second line is the sequence `y`. Some of these files have an associated output (`.mat`) file containing the expected output. For example, here is an input file:

```
>_ ~/workspace/global_sequence_alignment
$ cat data/endgaps7.txt
atattat
tattata
```

and here is the corresponding output file:

```
>_ ~/workspace/global_sequence_alignment
$ cat data/endgaps7.mat
atattat
tattata
8 8
  4   4   5   6   8  10  12  14
  2   4   4   4   6   8  10  12
  4   2   4   4   4   6   8  10
  6   4   2   3   3   4   6   8
  8   6   4   2   3   2   4   6
 10   8   6   4   2   2   2   4
 12  10   8   6   4   2   1   2
 14  12  10   8   6   4   2   0
```

The `solutions.txt` file lists the input file names along with the optimal alignment scores and the output file names (if one exists).

Files to Submit:

1. `Reverse.java`
2. `Distance.java`
3. `Transpose.java`
4. `StrangeMatrix.java`
5. `Pascal.java`
6. `EditDistance.java`
7. `Alignment.java`
8. `notes.txt`

Before you submit your files, make sure:

- You do not use concepts from sections beyond *Input and Output*.
- Your code is adequately commented, follows good programming principles, and meets any problem-specific requirements.
- You edit the sections (`#1` mandatory, `#2` if applicable, and `#3` optional) in the given `notes.txt` file as appropriate. Section `#1` must provide a clear high-level description of each problem in no more than 100 words.

Acknowledgement: Part II of this assignment is an adaptation of the Global Sequence Alignment assignment developed at Princeton University by Thomas Clarke, Robert Sedgewick, Scott Vafai, and Kevin Wayne.