Goals
This assignment aims to help you:

- Learn about dynamic programming.
- Character coding

Reading
K&T, Chapter 4 (greedy algorithms), Chapter 6 (dynamic programming) S&W, Chapter 5.5 (data compression)

Questions
1. Sequence alignment: The question is based on "Creative Programming assignments" by Sedgewick and Wayne:
   https://introcs.cs.princeton.edu/java/assignments/sequence.html
   Adaptations are taken from: http://www.cis.upenn.edu/ cis110/12su/bw/hw06/dynprog.shtml
   The background in the question provides the recursive brute force process, but you should only implement the dynamic programming part. You may test your program on the sequences suggested in the checklist (see link to original assignment).

   It is very important that you fully understand the dynamic programming sequence alignment algorithm before you start programming. The algorithm, once you understand it, is very straightforward to implement, but the understanding part requires careful consideration. What helps is to run an example on paper. For example, for the two sequences in the question build the dynamic programming matrix and fill it manually, making sure you’re getting what you suppose to get. Notice that the algorithm described in the PA is somewhat different than the one I described in class, but the principle is very similar. For example, the table is filled from bottom to top and right to left (which I find odd but whatever...), so the final edit distance can be obtained from \texttt{opt[0][0]}.

   Some guidelines:

   - Your program should get a file as a command line parameter. Usage: \texttt{java cs310.EditDistance input.txt} where \texttt{input.txt} contains two DNA sequences separated by a newline. This is different from the description in both links above.
   - Your program should consist of a class \texttt{Match.java} with one static method to compute the optimal match between a pair of objects. The optimal solution is returned as a linked list of \texttt{Path} nodes. Each \texttt{Path} node stores the row and column of the element it represents, the total cost of the match from that point to the end, and a pointer to the next path node. So for opt matrix above, the Path nodes would follow the red numbers: the row and col variables of the first node would both be 0 and cost would be 7. In the second node, row and col would be 1 and cost would be 6, and so on. The reason for this is that the DP formula gives you the edit distance, but you also need to recover the alignment itself.
• The classes `Match`, `Path` and `EditDistance` should follow the API described here: [http://www.cis.upenn.edu/~cis110/12su/hw/hw06/dynprog.shtml](http://www.cis.upenn.edu/~cis110/12su/hw/hw06/dynprog.shtml)

With the following differences:
- Make the variables private in `Path`. Failure to do so will result in style check error. You should write getters and setters as needed.
- As mentioned above, have `EditDistance` read from a file, rather than the standard input.

• The output should be in the same format as described in the link above.

• Use standard java, not `algs4`.

**Notice:** The link above provides a set of sequences for testing. I will use several of them, for example `example10.txt` (you can restrict yourselves to small files). The output format is exactly as shown in the link.

2. Remember the greedy algorithm for the change making problem we mentioned in class: Given coin denominations and an amount to be paid, devise a method to pay that amount using the fewest possible number of coins. We showed that the greedy algorithm always produces an optimal solution for US coins, that is – \{1, 5, 10, 25, 100\} cents. However, for other denominations, this greedy cashier’s strategy might not be optimal. For example, if we add a 7 cent coin, the optimal solution for 14 cents is two coins of 7, but the greedy algorithm gives the following: \{10, 1, 1, 1, 1\}.

In this problem you are asked to implement a dynamic programming solution to the coin changing problem that always produces an optimal solution (i.e., minimum number of coins) regardless of the coin denominations.

As a reminder, your input is:

• An array `denominations[1..n]` containing the n coin denominations \(d_1, ..., d_n\) that you can use (for example, \{1, 10, 21, 34, 70, 100\}, in this case n=6).

• The amount M that you need to pay (in the example above, M = 140 cents).

You can assume that all the coin denominations and M are positive integer numbers.

The command line parameter is a file, say `coins.txt`, where the first line is the amount of money you should count and the second line is the denominations, separated by spaces. For the example above, `coins.txt` looks like this:

```
140
1 10 21 34 70 100
```

You can assume that this array is already sorted in increasing order (with no repetitions, of course). You can also assume that you have an unlimited number of coins of each denomination.

Usage: `java Coins coins.txt`

Your output should be:

• The minimum number of coins needed to make change for M (in the example above where M=140, the optimal number of coins is 2).

• The number of coins of each denomination used in the optimal solution (in the example above where M=140, the optimal answer is 2 coins of 70 cents. Remember that in the general case more than one denomination can be used).

The exact output is for the example above should be:

```
140: 2 coins
70 * 2
```

**Hint:** You should keep track of the solution for any \(n \leq M\), so you should define an array of size M that keeps track of the number of coins for each \(n\), plus another array of size M that keeps track of the last coin used, so you can trace back the optimal combination of coins, just like the code shown in class. Use standard java.

In this assignment you only do the move-to-front part. I copy the relevant parts here, with some edits:

**Binary input and binary output:** To enable your programs to work with binary data, you will use BinaryStdIn and BinaryStdOut, which are described in Algorithms, 4th edition and part of `algs4.jar`. To display the binary output when debugging, you can use HexDump, which takes a command-line argument n, reads bytes from standard input and writes them to standard output in hexadecimal, n per line.

```bash
> more abra.txt
ABRACADABRA!
41 42 52 41 43 41 44 41 42 52 41 21
96 bits
```

Note that in ASCII, 'A' is 41 (hex) and '!' is 21 (hex).

**Move-to-front encoding and decoding:** The main idea of move-to-front encoding is to maintain an ordered sequence of the characters in the alphabet, and repeatedly read in a character from the input message, print out the position in which that character appears, and move that character to the front of the sequence. As a simple example, if the initial ordering over a 6-character alphabet is A B C D E F, and we want to encode the input CAAABCCCACCF, then we would update the move-to-front sequences as follows:

<table>
<thead>
<tr>
<th>move-to-front</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E F</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C A B D E F</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A C B D E F</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A C B D E F</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A C B D E F</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B A C D E F</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C B A D E F</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>C B A D E F</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>C B A D E F</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A C B D E F</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C A B D E F</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>C A B D E F</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>F C A B D E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the same character occurs next to each other many times in the input, then many of the output values will be small integers, such as 0, 1, and 2. The extremely high frequency of certain characters makes an ideal scenario for Huffman coding (which you don’t have to implement).

**Move-to-front encoding:** Your task is to maintain an ordered sequence of the 256 extended ASCII characters. Initialize the sequence by making the i\(^{th}\) character in the sequence equal to the \(i\)\(^{th}\) extended ASCII character. Now, read in each 8-bit character \(c\) from standard input one at a time, output the 8-bit index in the sequence where \(c\) appears, and move \(c\) to the front.

```bash
```

96 bits

**Move-to-front decoding:** Initialize an ordered sequence of 256 characters, where extended ASCII character \(i\) appears \(i\)\(^{th}\) in the sequence. Now, read in each 8-bit character \(i\) (but treat it as an integer between 0 and 255) from the standard input as binary one at a time, write the \(i\)\(^{th}\) character in the sequence, and move that character to the front. Check that the decoder recovers any encoded message.

```bash
> java -cp .:/lib/algs4.jar MoveToFront - < abra.txt | java -cp .:/lib/algs4.jar MoveToFront
```

ABRACADABRA!

Name your program MoveToFront.java and organize it using the following API:
public class MoveToFront {
    // apply move-to-front encoding,
    // reading from standard input and writing to standard output
    public static void encode()
    {
        // apply move-to-front decoding,
        // reading from standard input and writing to standard output
        public static void decode()
        {
            // if args[0] is '-', apply move-to-front encoding
            // if args[0] is '+', apply move-to-front decoding
            public static void main(String[] args)
        }
    }
}

Performance requirements: The running time of move-to-front encoding and decoding should be
proportional to $R \times n$ (or better) in the worst case and proportional to $n + R$ (or better) in practice
on inputs that arise when compressing typical English text, where $n$ is the number of characters in the
input and $R$ is the alphabet size.

Note about the file abra.txt: When you download or create the file, notice that most operating
systems will add a newline character to the end of the file. So when you try your program on it, you'll
see an additional character. To remove it, do the following: `tr -d \n < abra.txt > tmp` and then
`mv tmp abra.txt`. This simply deletes the newline.

Delivery

- Match.java, Path.java, EditDistance.java. All three files should be uploaded to Gradescope. Usage:
  `java EditDistance input.txt`, where `input.txt` contains two sequences, each in a separate line.
- Coins.java. Usage: `java Coins coins.txt` as described above.
- MoveToFront.java, usage: See above.
- memo.txt. Indicate any late days and answer the following questions:
  1. What is the runtime of the DNA alignment DP algorithm?
  2. What is the space required to implement the DNA alignment DP algorithm?