Goals

This assignment aims to help you:

- Learn about dynamic programming.
- Huffman 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](https://introcs.cs.princeton.edu/java/assignments/sequence.html)

Adaptations are taken from: [http://www.cis.upenn.edu/cis110/12su/hw/hw06/dynprog.shtml](http://www.cis.upenn.edu/cis110/12su/hw/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 matrix on the webpage is filled out from bottom to top and from right to left (which I find odd, I usually do it backwards but whatever). Therefore, the final edit distance can be obtained from opt[0][0].

Some guidelines:

- Your program should get a file from the standard input. Usage: `java cs310.EditDistance < input.txt`, where 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 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 Path nodes. Each 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)

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

• Use standard Java and place all your files (from this and the other questions in a cs310 package, as per the instructions in the previous PAs.

In your memo.txt answer the following questions:

(a) What is the output from matching the two sequences:
   x = "AACAGTTACC" and y = "TAAGGTCA"?

(b) What is the runtime of the DP algorithm?

(c) What is the space required to implement the algorithm?

2. The **maximum increasing subsequence** problem is defined as follows:
   Given a sequence of numbers, find an increasing subsequence of maximum length contains in it. The subsequence does not have to be contiguous (which means – the numbers don’t have to follow consecutively in the array). Also, we are looking for the sequence as the final answer. For example, if the input is:
   
   0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

   A maximum contiguous subsequence is:
   
   0, 2, 6, 9, 13, 15

   Notice that the solution is not necessarily unique, so other subsequences of size 6 may exist. I will accept any valid sequence.

Write a **dynamic programming** program to solve the problem. The idea is as follows: given an input sequence \(a[0] \ldots a[n-1]\) each index \(j\) either extends a previous maximum length subsequence or starts a new one if \(a[j]\) is not larger than any previous element in the sequence. Notice that since the sequence is not necessarily contiguous, it’s not sufficient to only check \(a[j-1]\) for each \(a[j]\), but possibly all the elements left of \(a[j]\). You can assume that no two elements in your input array are equal.

In addition to your input array keep also an array \(p[]\) where \(p[j]\) is the predecessor of \(a[j]\) in the maximum increasing subsequence ending in \(a[j]\), and an array \(s[]\) where \(s[j]\) is the size of the maximum subsequence ending in \(j\). These arrays will later be used to backtrack on the output and print the sequence.

   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.

   ```
   > 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>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>A</td>
<td>2</td>
</tr>
<tr>
<td>C A B D E F</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>F C A B D E</td>
<td>F</td>
<td>5</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.

```
41 42 52 02 44 01 45 01 04 04 02 26
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 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.

```
> java -cp :.../lib/algs4.jar cs310.MoveToFront - < abra.txt | java -cp :.../lib/algs4.jar cs310.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 $R^2n$ (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**: Usage “java cs310.EditDistance < input.txt”, where input.txt contains two sequences, each in a separate line.
- **DynamicSubsequence.java**, usage: “java cs310.DynamicSubsequence arr”,
- **MoveToFront.java**, usage: See above.
- **memo.txt**. Indicate any late days and answer the questions above. Also copy and paste the output of your programs.

Notice that you don’t need to use the algs4 library for the first two questions.