CS310: Advanced Data Structures and Algorithms

Fall 2021 Programming Assignment 3

Due: Monday, December 6, 2021 at midnight

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

   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:

   - You don't need the `EditDistance` class. The other two do all the work anyway, and `Match` can provide both the match and the distance.
   - Your program should get a file as a command line parameter. Usage: `java pa3.Match 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 public 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.

- You can take ideas for `Match` and `Path` from the API described here:
  http://www.cis.upenn.edu/ cis110/12su/hw/hw06/dynprog.shtml
  BUT
  - The `match` function inside the `Match` class should be public and not static.
  - The `Path` class variables should be private. The getters should be: `public int getRow()`, `public int getCol()`,`public int getCost()` and `public Path getNext()`.
  - You will also need setters.
- The output should be in the same format as described in the link above (I won't test it in the autograder but I'll ask you to paste it in your `memo.txt` file).
- Use standard java and place all your files (from this and the other questions in a `pa3` package, as per the instructions in the previous PAs.

2. Remember the greedy algorithm for the change making (cashier) 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 the 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. This problem was addressed in class.

   As a reminder, your input is:

   - A set of numbers 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).

   The input format 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 all the coin denominations and M are positive integer numbers. You can also assume that this array is already sorted in increasing order (with no repetitions, of course), even though it wouldn't hurt to check. You can assume that you have an unlimited number of coins of each denomination.

   Usage: `pa3.Coins coins.txt`

   Your class should be able to calculate:

   - 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 class `Coins` has to have the following two methods (and others as needed):

   - The amount is a class variable. Implement a getter: `int getAmount()`
   - `public int makeChange()`. This function returns the number of coins (not the coins themselves!). In the example above it should return 2. You may safely assume a solution exists.
   - `public int howMany(int coin)` which returns how many coins of type `coin` we use. For example, `howMany(70)` should return 2. `howMany(21)` should return 0. The function should accept any positive number, and if it's not used (or is not a denomination), simply return 0.

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. You also need to find an efficient way to trace back the optimal combination of coins in a way that won't force you to re-run the algorithm every time `howMany` is called. To do this you can save your combination in a data structure that can be queried easily.

3. Implement Move-To-Front. Loosely based on: http://coursera.cs.princeton.edu/algs4/assignments/burrows.html In this assignment you only do the move-to-front part with some changes to prevent relying too much on the standard input and output. 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 `BinaryIn` and `BinaryOut`, which are described in Algorithms, 4th edition and part of `algs4.jar` (as opposed to `BinaryStdIn` and `BinaryStdOut`. This means you have to work with files, and treat them like binary streams) 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!
```

```
> java -cp .:../lib/algs4.jar edu.princeton.cs.algs4.HexDump 16 < abra.txt
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:

move-to-front	in	out
A B C D E F	C	2
C A B D E F	A	1
A C B D E F	A	0
A C B D E F	A	0
A C B D E F	B	2
B A C D E F	C	2
C B A D E F	C	0
C B A D E F	C	0
C B A D E F	A	2
A C B D E F	C	1
C A B D E F	C	0
C A B D E F	F	5
F C A B D E		

   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 the input file one at a time, output the 8-bit index in the sequence where c appears, and move c to the front.

```
> java -cp .:../lib/algs4.jar pa3.MoveToFront - abra.txt | java -cp .:../lib/algs4.jar
edu.princeton.cs.algs4.HexDump 16
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 input file 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.
Notice: Your functions should print the encoding and decoding output to the standard output, but nothing else should be printed. So for example, the output of:

```
> java -cp .:../lib/algs4.jar pa3.MoveToFront + abra_dec.txt
```

Should be: ABRACADABRA!
(without the newline).

The file `abra_dec.txt` is the encoded result of `abra.txt`. If you open it with a test editor it will look weird because it has non-printable characters.

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(String f)

    // apply move-to-front decoding,
    // reading from standard input and writing to standard output
    public static void decode(String f)

    // if args[0] is '-', apply move-to-front encoding
    // if args[0] is '+', apply move-to-front decoding
    // args[1] is the input file name
    public static void main(String[] args)
}
```

The methods have to be static because you want to use encode and decode on a fresh copy. **Performance requirements:** The running time of move-to-front encoding and decoding should be proportional to R*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.

Delivery

- `Match.java, Path.java`: Usage `java pa3.Match input.txt`, where `input.txt` contains two sequences, each in a separate line.
- `Coins.java`. Usage: `java pa3.Coins coins.txt` as described above.
- `MoveToFront.java`. Usage: See above.
- `memo.txt`. Indicate any late days and answer the questions below.

In your `memo.txt` answer the following questions:

1. What is the output from matching the two sequences:
   x = "AACAGTTACC" and y = "TAAGGTCA"?
2. What is the runtime of the matching DP algorithm?
3. What is the space required to implement the algorithm?
4. Analyze the runtime and space usage of your Move to front algorithm