Homework 2 Writeup

In this homework, you will write programs for several greedy and dynamic programming algorithms.

Part 1: Understanding

There are two files that you will be using:

1) GreedyDynamicAlgorithms
2) Huffman

1) GreedyDynamic Algorithms contains the first part of the homework. In it, you will find two problems:

a) optimal intervals
b) path through a grid

Both of these questions are explained in more detail below. In one of them, a greedy approach can be used to solve the problem. In the other, you will have to use dynamic programming. Your first goal will be to figure out the algorithm that solves each problem!

2) Huffman is a class for encoding strings using the huffman algorithm. More hints for implementing the class are described later, but make sure to understand the algorithm first! An animated example of the algorithm can be found in the lecture slides.

Part 2: Greedy and Dynamic:

You have two problems to solve, using either greedy or dynamic programming algorithms. Your first job, before writing any code, is to figure out an algorithm that will solve the problem!

General tips for a greedy problem:

-What is a good top-level question to ask?

-Brainstorm some possible greedy choices that the algorithm could make. Can you eliminate any of them with some counter-example inputs?

-Can you answer the question right away, and be sure that this will lead to the optimal solution?

Remember general tips for a dynamic programming problem:

-What is a good top-level decision to make?

-What are the base cases for the problem?

-Can you come up with a recurrence relation? How can you solve larger problems using the solutions to subproblems?

-Will you implement the algorithm iteratively or recursively? If you implement it iteratively, in what order must the subproblems be solved?

-If you choose to implement your algorithm recursively, don't forget to memoize the solutions to subproblems.

<u>Question a: optimal intervals</u>

In this problem, you are given a list of intervals (similar to the activities problem from the lecture slides). Each interval can be defined by a pair of numbers (start, finish), where start is the start time of the interval, and finish is the finish time. However, there's a small twist.

Each interval can also be either red or blue. The goal of this problem is to select the fewest number of RED intervals, such that all of the BLUE intervals are "intersected". In order for a blue interval to be "intersected", it must overlap with one of the red intervals that you selected.

Note that with this problem, you will not have to output the actual optimal list of red intervals chosen. You only need to output the integer corresponding to the optimal number of red intervals.

For example, suppose we're given a list of blue intervals:
(0, 2) (5, 5) (7, 10) (11, 13)
and a list of red intervals:
(0, 4) (2, 5) (4, 8) (9, 10) (9, 11) (10, 12) (11, 12)

You may want to draw these intervals out to get a better mental picture.

We must choose the fewest number of red intervals such that all 4 blue intervals are intersected. In this case, the optimal answers is 2: we choose the intervals (2, 5) and (10, 12). With these two red intervals, all 4 blue intervals are intersected.

Finally, note that in the case where no solution is possible (it is impossible to cover all of the blue intervals), your solution should return -1.

Here are a few hints:

1) Should you use a greedy algorithm or a dynamic programming algorithm? Can you greedily choose a red interval based on some local decision without losing the optimal solution?

2) One idea that you might consider would be to greedily choose the longest red intervals, one at a time, until all blue intervals have been intersected. Surprisingly, this algorithm doesn't work. Can you come up with an example set of red and intervals that causes this algorithm to fail?

3) In the above example, we know that the blue interval (0, 2) must be intersected. The only two red intervals that intersect it are (0, 4) and (2, 5). The optimal solution here uses the interval (2, 5). What makes this interval a better one to use?

4) Pictures help! Try drawing out a few examples as you're coding to help with keeping track of any indices, etc.

Finally, implement the optimalIntervals function found in GreedyDynamicAlgorithms.java! One thing to note: we've given you an Interval class with two methods that you may find useful: sortByStartTime() and sortByFinishTime() both take in a list of intervals, and sort them by start and finish time, respectively.

<u>Question b: optimal path through a grid</u>
In this problem, you are given an mxn grid (a 2d array) of integers, where each value in the grid contains a "cost". You start at position (0, 0) in the grid in the top left corner, and must travel to position (m-1, n-1) in the grid (the bottom right corner). Each time you move in the grid, you are only allowed to move either down or to the right. There is one twist: the value at each spot in the grid denotes the cost of entering that position. Your goal is to figure out an of the optimal path through this grid.

For example, suppose we're given this grid:

```
5 1 1
2 4 7
2 4 5
5 6 3
```

The path of lowest cost, going from the top-left to the bottom-right portion of the grid, will be the sequence: DOWN DOWN RIGHT RIGHT DOWN. This optimal path has a cost of 5 + 2 + 2 + 4 + 5 + 3 = 21. Any other path through this grid will have a cost at least as large as 21 (note that there could be more than one path with the optimal cost, but in this example there isn't. In the case that there is more than one optimal path, you are free to output any one of these paths).

NOTE: You will see that the output of this function is a List<Direction>. We've provided you with a Direction enum, which consists of two possible values: DOWN or RIGHT. For example, the actual list you would output in the example above would look something like:

```
    List<Direction> output = new LinkedList<>();
    output.add(Direction.DOWN);
    output.add(Direction.DOWN);
    output.add(Direction.RIGHT);
    output.add(Direction.RIGHT);
```

```
        output.add(Direction.DOWN);
```

A few hints for this problem:

1) Should you use a greedy algorithm or a dynamic programming algorithm? Can you greedily choose whether or not to go down or to the right at each spot in the grid without losing the optimal solution?

2) One idea might be to look at the spots immediately down and to the right of your current position in the grid, and move to the spot with a lower cost. This algorithm will fail. Can you come up with a simple grid where this algorithm won't produce the optimal path?

Part 3: Huffman Encoding

The second part of the homework is to implement the huffman algorithm that you saw in the lecture slides. As with the last homework, you want to familiarize yourself with the algorithm, as well as the animation in the lecture slides, before you jump into the code.

Next, we'll walk through the Huffman class. This class takes in an input string in the constructor, and produces two pieces of data (which you should store as global field variables). The 3 field variables in this class that you must store (although you're free to add other field variables if you wish) are:

input - the original string to be encoded

huffmanTree - This will be your huffman binary tree, which you will create in the constructor. It is of type Node, which is a small class that we've given you (explained below). Remember, the way a huffman tree translates into an encoding is that the leaves of the tree represent characters. The encoding of each character can be expressed by the path taken from the root of the tree to that leaf: going left in the tree denotes adding a "0" to the encoding, while going right in the tree denotes a "1".

mapping - A mapping from Characters to binary Strings. You will also create this mapping in the constructor, using your huffmanTree.

We've given you two functions in the Huffman class:

1) getFreqs - No need to do anything here, we've used this function for your in the Huffman constructor. It will take in your input string, and convert it into a mapping from each character in the string to its frequency (the number of times that it appears)

2) compressionRatio - You won't be using this function either, but it tells you how well your huffman algorithm compressed the input string (note that we calculate this with the formula (encoded length) / (8 * original length). This is because ascii character require 8 bits to store, while 0s and 1s are only 1 bit each).

It also contains 3 main functions for you to implement:

1) Huffman constructor - Takes in the string to be encoded, and creates the huffmanTree as well as the encoding mapping. Remember to refer back to the lecture slides for tips on the algorithm. Your constructor should do a few things:

a) create nodes for each character in your string

b) add all the nodes to the priority queue

c) continue to merge the two nodes with lowest frequency until only one node remains in the priority queue (this will be the final huffman tree)

d) use this huffman tree to create the encoding mapping (from characters to binary strings)

2) encode - encodes the "input" string using your encoding mapping, and returns the encoded string (i.e. a string of 1s and 0s)

3) decode - Takes in a binary string, and decodes it. You will need to use the huffmanTree to do this. One important thing to remember is that your huffmanTree is prefix-free. Because of this, each sequence of 1s and 0s in your binary string will correspond to exactly one character in the huffman tree.

With all of these functions, here's an example of how you would use the Huffman class:

```
Huffman h = new Huffman("aabc");
String encoding = h.encode;
//With this string, we should have the mapping "a"->0, "b"->10, "c"->11.
//Thus the encoded string should look like "001011".
String decode = h.decode(encoding)
//decode should be the same as the original string, "aabc"
```

NOTE: We've given you some starting code in the constructor. First, we wrote a convenient function for you that creates a frequency mapping from a string. Next, we've instantiated a PriorityQueue for you. If you haven't seen this class before, it stores a list of elements (in this case, Nodes), and has two important functions that you will use: add(n) adds a node n to your priority queue. spoll() will extract the node with lowest frequency from your priority queue. These should be the only two functions that you need, but more information can be found by searching for PriorityQueue in the javadocs.

NOTE 2: We've also given you a helpful Node class that you will use in your implementation (see at the bottom of the Huffman.java file). Every node contains 4 pieces of data:

- -the character corresponding to that node (only applies to leaves)
- -the frequency of that node (remember, when you combine two nodes, the new node outputted should have the sum of their frequencies)
- -the left and right subtrees of this node.

One important thing to note is that only the leaves of your tree will have a character. If your node is not a leaf, its character should be Null. For example, to construct a leaf node you would write

```
Node myLeaf = new Node("a", 5, null, null)
//the character "a" shows up 5 times
```

To construct a nonLeaf, you would write

```
Node notLeaf = new Node(null, left.freq + right.freq, left, right)
//not a leaf, so it's character should be null.
```

You will also need to know whether or not a given node is a leaf when writing your program. To help with this, we've given you a small function, isLeaf().

As with the last homework, we won't be testing edge cases- the focus will be on implementing the algorithm correctly. Note that one other edge case that we won't test for is any encoding with a single repeated character (ex: "aaaa"). Can you think of why this presents an edge case in the implementation?

Now that you've read through the Huffman class, you can implement the 3 functions!