CS532: Homework 4

On the website, there is a preliminary unittest.

Min Priority Queue

In these questions, you will be implementing a min priority queue as closely as possible as the implementation in the textbook, in chapter 6. The only difference is that you will be using an array that starts at 0, rather than 1, and implementing a min version, rather than a max version.

Question 1: Parent and Children

For the class MinHeap, implement static methods for parent, left and right. We discussed static methods in lecture slides 2. These methods should take an index as their input and return an index.

Just as in the textbook, you do not need to make sure that the resultant index is in the array. Remember, your array is starting at 0, not 1, as in the textbook.

Hand in your static methods.

Question 2: Min_Heapify, Build_Min_Heap, Heap_Extract_Min

Implement these routines. Make your code as similar to the textbook code as possible. The textbook uses a number of variables in implementing a heap: a fixed length array (lst), the length of the list currently being used (length), and how much of the list is in a heap (heap_size). To keep all of these variables together, implement the MinHeap class, and make these 3 routines be methods of that class.

Since you are using a class, you need an initializer for it. Since we are following the textbook and using a fixed length array, pass in the maximum size of the array as the only argument. After you create the heap object, your code (outside of the class) should initialize the list, and set length to the appropriate value. None of your methods or your code outside of the class should alter the physical size of the list; so do not use append nor pop.

For extract-min, the textbook just decreases the heap-size variable by 1, but not the length. Your code should decrease heap-size and length by one.

Hand in your code. Also hand in code to demonstrate the use of your routines, including how you create the object, then initialize the list, and then call Build_Min_Heap put it in min-heap order.

Question 3: Iterative version of Min_Heapify

Change your code for Min_Heapify so that it uses iteration instead of recursion. Try to make as few changes to the code as possible.

Hand in your code.

Question 4: Min_Heap_Insert

The textbook uses Heap_Increase_Key in its Max_Heap_Insert. Make a method Min_Heap_Insert that instead does the insertion solely on its own, and uses iteration. Make your version as similar to your iterative version of Min_Heapify.
I deviated from the textbook code so that it increases both the length variable as well as the heap_size variable.

Hand in your code.

**Brute Force String Alignment**

The past homeworks had questions in a section called ‘Prelude to String Alignment’. Make sure you understand the code for those two questions as they have a similar control structure for what you need for the following questions. In the previous homeworks, you were printing and scoring all paths through a binary tree in which you can go one of two directions, here you are printing and scoring all paths through an array, in which you can go one of three directions: right, down, and diagonally right-down.

**Question 5: Tracing Paths**

Make a function that will trace out all paths through the 2 dimensional array. You do not actually have to create the two dimensional array for this problem, instead you will have the two one-dimensional arrays x_array and y_array that contain the two strings that need to be aligned.

The function will be passed the current location, x and y, from there it can go down, left, or diagonally down and left, making sure to not go outside the bounds of the array. When you reach the lower-right hand corner you should end.

Also, keep track of the alignment as you go, ‘-‘ for going right, ‘|‘ for going down, and ‘\’ for going diagonally right-down. To use ‘\’ in python, you might need to escape it with a ‘\\’.

Call your routine version1, and it should take 5 parameters, x_array, y_array, two integers, x and y, and a string with the current alignment. You should call it with:

```
version1(["A","B","C","C","D"], ["A","E","B","F","C","D"], -1, -1, "")
```

When the function reaches the lower-right hand corner, it should print the alignment and return.

Note that your routine will essentially be doing a depth-first search through paths through the array. Your code for version1 will be somewhat similar to PrintAllPaths.

Below are the first few lines that you should output.

```
-----\|||||
-----|-\|||||
-----|--\|||||
-----|--\||-|||
-----|--\|--\|\|
-----|--\|--\|--
-----|--\|--\|--
-----|--\|--\|--
-----|--\|--\|--
```
Hand in your code for the function version1 (and the code for any other functions that it might call).

**Question 6: Exponential Size**

Show that this routine runs in exponential time. Do this by having x_array and y_array the same length, and varying this length from size 1 to 10, and time how long the code takes to run. Feel free to comment out the print statement so that your code runs faster.

If this is exponential time, we should be able to approximate its running time with $c^n$, where $n$ is the input size and $c$ is some constant. In other words, with each successive $n$, the running time increases by some $c$.

Print out a table of your results, showing size of the arrays versus running time. Add a third column in which you divide the running time by the running time of the previous row.

Hand in your table, and discuss how stable the third column is and whether that suggests the running time is exponential.

**Question 7: Scoring Alignments**

Create function version2. This function will be similar to version1 so that it also computes the score of the alignment. Do this by adding a 6th parameter to the function, which will be the score so far. When version2 is initially called, this parameter should be given 0. For each down or left, 1 should be added to the score for the next recursive call. For the diagonal, 1 should be added only if the current character in the two arrays is not the same. When a full alignment is found, print out the score and the alignment before returning.

When called on the same arrays as in Question 5, the first part of my output looks as follows:

```
11 -----|--|||--
11 -----|--|--|--
11 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
9 -----|--|--|--|
10 -----|--|--|--|
10 -----|--|--|--|
10 -----|--|--|--|
10 -----|--|--|--|
10 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
11 -----|--|--|--|
9 -----|--|--|--|
```

Hand in a copy of your code.
Question 8: Finding the best alignment

As the final part of doing a brute-force string alignment, create version3 that will return the best alignment. On the arrays from Question 5, you should get the following result.

2 \|\\

Your code for version3 should be based on your code for version2. Do not use a global variable for this question to store the best score/alignment so far.

There are two ways you can do this. First, you can have version3 pass around the best/alignment seen so far, by adding two extra input variables, and each invocation determining if it has seen something better than the current best, and passing this new best back via the return statement.

Or, second, you can not add extra input variables to version 3 and have the parent invocation decide which of its children has seen the best score, and pass the best result up. (This version is closer to what you will be doing in the next question.)

Hand in your code for version3.

Question 9: An Alternate Approach

In Question 7, we added scoring by using a 6th argument to pass the score of the path so far into the recursion. When the recursion hits the base case (lower right-hand of the rectangle) it returns the final score, which is then passed all of the way out of the recursion. We follow the same approach for the alignment path as well, passing in the current path as the 5th argument.

For this question, you will create version4. This will compute the same the same results as version3 but will compute the path and score in an alternate way. We will not pass them into the recursion as the 5th and 6th arguments. Instead, you will compute these on the way out of the recursion by passing them back as the return value. Thus the base case of the recursion (the end of the recursion) will simply pass back a score of 0 and an empty string. After a return from a recursive call, you will take the score and path returned, and add on the score from the current step, and add the direction onto the beginning of the path. Note that this approach is similar to how we implemented Search for binary search trees. In fact, you might want to contrast the code for PrintAllPaths with Search to better understand how version3 and version4 are to function.

Make your code as close to version3 as possible.

Dynamic Programming for String Alignment

Question 10: Top-down with Memoization

Your code for version3 and version4 was very inefficient as it kept on recomputing the answers to the same subproblems over and over again.

To overcome this, you will be changing your code so that it saves partial answers so that they do not need to be recomputed every time. The question is, what partial answers should you save in order to make your code much more efficient (changing it from exponential to polynomial)?

Think about this question carefully. Think about the difference between version3 and version4.
Question 11: Code

Create version5 that is identical to version4 except it saves partial results, and then checks if partial results are available instead of recomputing them. For this question, you can use a global variable called cache to hold your partial results, and it can be a dictionary.

You will be given 0 marks if you turn in a version of the code that is bottom-up or that substantially differs in structure from version3 or version4.

Structure your code so that the first thing it does is checks if the answer is in the cache. If it is, you should be able to do a quick computation (no recursion) and return the answer. Otherwise, do the full computation (including recursive calls), and at the end of your code, save the answer to the subproblem in the cache.

Turn in your code. If you did this correctly, you should find the same answer as you did for version3 and version4, but substantially faster. It should now run in polynomial time, rather than exponential time.