Dynamic Programming for String Alignment

These questions continue on from the previous homework. If you did not have a correct working version of version5, use the one from the answers from the previous homework. Make sure that you have copy the indentation correctly!

Question 1: Big O, Theta, and Omega

For measuring the speed of your code, argue why it does not matter what input arrays you use. In order words, argue that best and worst case performance is the same. What does this mean in regard to Big-O, Theta and Omega?

Question 2: Running Time versus Input Size

Measure the speed of your code as the input size increases. For this, vary the array sizes by 10 from size 10 to 500. Your code should run in time \( \Theta(n^2) \). After each run, make sure you empty the cache.

Your algorithm should run in time \( O(n^2) \). As \( n \) increases, the actual time will get closer and closer to \( cn^2 \) for some \( c \). For each input size, compute the value of \( c \). Does it’s value seem to be stabilizing as \( n \) increases?

Hand in a table with the running time and value for \( c \) for all of the runs. Argue whether it \( c \) seems to be stabilizing or not.

Question 3: Analysis of the Code

In the previous question, you gave empirical evidence that the running time is \( \Theta(n^2) \). Justify this answer by analyzing your code.

Question 4: Top-down versus Bottom-Up

The code in the textbook is a bottom-up implementation. Explain how the data that is saved for your top-down version is different from the data saved for the bottom up version.

Binary Knapsack Problem

The textbook discussed the binary knapsack problem in Chapter 16. We also dicussed it in class.

Let \( w_i \) be an array of positive integer weights and \( v_i \) be an array of values, both indexed from 0 through \( n-1 \).

Find the combination of items (each can be used once or not at all) that has the greatest value, and weight at most \( w \).

\[
b(0, w) = \begin{cases} 
  v_0 & \text{if } w_0 \leq w \\
  0 & \text{otherwise}
\end{cases}
\]
For \( i > 0 \)

\[
b(i, w) = \max(b(i-1, w),\ b(i-1, w-w_i) + v_i)
\]

**Question 5: Meaning of the Function**

Explain what the function \( b(i, w) \) represents.

**Question 6: Base Case**

Explain the rational behind the mathematic definition of the base case: \( b(0, w) \). Make sure to address why there are two cases.

**Question 7: Recursive Case**

Explain the rational between the recursive case: \( b(i, w) \). Explain why there is a max, and what the two choices for the max are.

**Question 8: Subproblems**

At each step in the recursion, how many choices are there, and for each choice, how many subproblems are there.

**Question 9: Dynamic Programming**

Explain why dynamic programmin can be applied. Make sure you talk about the two core requirements of dynamic programming

**Question 10: Bottom-up**

The bottom-up approach does not use recursion; instead it uses iteration to go through the subproblems. To apply bottom-up dynamic programming to this problem, why must we iterate through the subproblems in a certain order? What is that order?

**Question 11: Storing Intermediate Values**

What kind of data structure can we use to store the intermediate results that we compute for bottom-up. The data structure should be as simple as possible and as small as possible. Unlike the previous homework, do not suggest using a dictionary. Give the python code to initialize the data structure to all 0’s.

**Question 12: Bottom-up Code**

Create the function knapsack1 that takes as input an array of weights, a array of values, an overall weight, and returns the best value of items whose weight is at most the input weight. Knapsack1 must run bottom-up using dynamic programming. Your code should be consistent with the equations given at the beginning of this section.
Hand in the code. Since the next two questions depend on this solution, you can turn in your answer to just this question into Sakai.

**Question 13: Bottom-up Code: Returning the Solution**

Create knapsack2 by augmenting knapsack1 so that it keeps track of what items are included. It should return both the best value and a list of the items selected. Again, your code should make minimum changes to the earlier version. To verify it works, you should construct some knapsack problems that are more difficult than the 3 item one given in the textbook.

Hand in the code.

**Question 14: Top-down Code**

Only attempt this question after you have finished the rest of the homework.

Create the function knapsack3 that takes the same inputs as knapsack1 but computes the answer using a top-down approach with dynamic programming. This code should mirror the equations given above. You do not have to determine the items to include, just the value of the knapsack. Rather than pass the index ranges (i) into the function calls, just simply pass in the part of the weight and value arrays that should be processed.

To save your intermediate values, as with the last homework, you can use a global variable called ‘cache’ and implement it as a dictionary.

Debugging recursive code can be difficult. Even using a good debugger, it can be difficult to trace through a lot of recursive calls. In debugging my version, I used print statements to show me what was happening. I indented the print statements by the depth of the recursion by passing a adding a parameter to knapsack3 for the amount of indentation. Here is what the first part of my output looked like. For me, I was able to use this output to find several errors in my code as I was developing the solution.

```
Called with [10, 20, 30, 35, 15, 5, 20, 10, 15] [60, 100, 120, 165, 80, 35, 90, 50, 75] 80
Called with [10, 20, 30, 35, 15, 5, 20, 10] [60, 100, 120, 165, 80, 35, 90, 50] 80
Called with [10, 20, 30, 35, 15, 5, 20] [60, 100, 120, 165, 80, 35] 80
Called with [10, 20, 30, 35, 15] [60, 100, 120, 165] 80
Called with [10, 20, 30] [60, 100] 80
Called with [10] [60] 80
returning for [10] [60] 80 => 60
Called with [10, 20] [60, 100] 80
returning for [10, 20] [60, 100] 80 => 160
Called with [10] [60] 50
returning for [10] [60] 50 => 60
Called with [10, 20] [60, 100] 50
returning for [10, 20] [60, 100] 50 => 160
returning for [10, 20, 30] [60, 100, 120] 80 => 280
Called with [10, 20, 30] [60, 100, 120] 45
Called with [10] [60] 45
returning for [10] [60] 45 => 60
Called with [10] [60] 25
returning for [10] [60] 25 => 60
```
returning for [10, 20] [60, 100] 45=> 160
Called with [10, 20] [60, 100] 15
    Called with [10] [60] 15
    returning for [10] [60] 15=> 60
    returning for [10, 20] [60, 100] 15=> 60
returning for [10, 20, 30] [60, 100, 120] 45=> 180
returning for [10, 20, 30, 35] [60, 100, 120, 165] 80=> 345

Hand in the code for knapsack3