CS532: Homework 4 (version 8)

Spencer Hann

Version 1 includes correction to the output of question 2.
Version 2 includes the suggestion to comment out the print statement in question 3.
Version 3 added question 4 and 5.
Version 4 added questions on dynamic programming and string alignment.
Version 5: added what should be handed in for question 5. Fixed up the names of version3/4/5.
Version 6: Added a new question on ‘Efficiency: Part 3’.
Version 7: Fixed a few typos. For question 9, changed this to just go up to 500 rather than 1000.
Version 8: Friday 2:51pm: Clarified question 8 to say not to use global variables.

Note:

A few of my functions make use of an Alignmt class, to keep track of various alignments and their scores:

```
class Alignmt:
     def __init__(self, v=inf, s=""):
          # initialize 'best' score to
          # infinity for first comparison
          self.score = v
          self.str = s

     def __gt__(self, op):
          return self.score > op

     def __lt__(self, op):
          return self.score < op

     def __str__(self):
          return "%i %s" % (self.score, self.str)
     __repr__ = __str__
```

Brute Force String Alignment

Question 1: Asymptotic Behavior

In the class lecture notes 04, I discussed viewing an alignment as finding paths through an array where you start in the upper left corner and can move left, down, or diagonally left-down until the lower right corner is reached.
I claimed that this brute force method was $(n+m)^3$. This is a mistake. Prove that it is actually $O(3^{n+m})$.

The algorithm is actually exponential because each recursive call will make an additional 3 recursive calls, until the base case is met. Consider, there are *nm* “cells” in the table. Most of which are making 3 recursive calls.

Question 2: 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. Wh
en you reach the lower-right hand corner you should end.
Also, keep track of the alignment as you go, ‘-’ for going left, ’|’ for going down, and ’\’ for going diagonally left-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.
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).

```
def version1(str1, str2, x=0, y=0, path=""):
     if x > len(str1) or y > len(str2):
          return
     if x == len(str1) and y == len(str2):
          #print(path)
          return

     version1(str1, str2, x+1, y, path + '-')
     version1(str1, str2, x, y+1, path + '|')
     version1(str1, str2, x+1, y+1, path + '\\')
```

Question 3: 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 timing how long the code takes to run. There is python module timeit that you should use (or you can use the time module and the command time.clock()). Feel free to comment out the print statement so that your code will run faster.

```
     from timeit import default_timer as timer
     start = timer()
     # do stuff
     end = timer()
```

Print out a table of your results, showing size of the arrays versus running time. Roughly argue why your results indicate that this is exponential time.

```
          def exp_test(v, lower=1, upper=11, inc=1):
               version = {
                    1 : version1,
                    2 : version2,
                    3 : version3,
                    4 : version4
```

```
                }
                lst = []; lst0 = []

                print("n:\ttime:")
                i = lower
                while i < upper:
                        lst.append(i)
                        lst0.append(i)
                        start = timer();
                        version[v](lst,lst0)
                        end = timer();
                        t = (end-start)*1000
                        print(str(i) +"\t"+ str(t) + " ms")
                        i += inc

        exp_test(1)
```

```
Output:
        python3 hw4.py
        n:      time:
        1       0.014160003047436476 ms
        2       0.04880800406681374 ms
        3       0.22089100093580782 ms
        4       1.2453380040824413 ms
        5       6.463691999670118 ms
        6       31.744273997901473 ms
        7       168.6761149976519 ms
        8       903.2575169985648 ms
        9       4926.613231000374 ms
        10      27514.747361004993 ms
```

A simple way of noting that runtime is increasing exponentially with respect to input size, is to note the change in coefficients from n to t as n increases. For example, $n/t = c,$ 27514/10 = 2751.4, 4926/9 ≈ 547.3, 903/8 ≈ 112.9, and 168/7 = 24. As n increases at a linear rate, there is no c that can show a linear increase in t as well. This can be taken a step further using the form $x^n = t$, solving for x using $e^{\ln(t)/n}$. So, $x^7 \approx 168$ | x=2.07, $x^8 \approx 903$ | x=2.34, $x^9 \approx 4926$ | x=2.57. We can now see that x stays relatively constant, meaning the efficiency of my algorithm with respect to time in milliseconds, is roughly 2.07^n.
The slight increase in x as n increases is probably due to the fact that Python's timeit.default_timer() measures "wall clock" time, not CPU time. Measuring CPU time would be more accurate as it counts how long a process has actually spent in the processor. However

with the current metric, as the execution time grows, so does the number of interrupts my process has to deal with, as the scheduler gives CPU time to other programs running concurrently on my laptop.

Question 4: Scoring Alignments

Create the function version2 so that it also computes the score of the alignment. Do this by adding a 6th parameter to the function, to get the score so far. When version 2 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 2, the first part of my output looks as follows:

```
11 -----||||||
11 ----|-|||||
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.

```
    def version2(str1, str2, x=0, y=0, path="", score=0):
        if x > len(str1) or y > len(str2):
            return
        if x == len(str1) and y == len(str2):
            print(score,end=" ")
            print(path)
            return

        version2(str1, str2, x+1, y, path + '-', score+1)
        version2(str1, str2, x, y+1, path + '|', score+1)
        if x < len(str1) and y < len(str2):
            if str1[x] == str2[y]:
                version2(str1, str2, x+1, y+1, path + '\\', score)
            else:
                version2(str1, str2, x+1, y+1, path + '\\', score+1)
```

Question 5: 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 2, 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. Make version3 pass around the best/alignment seen so far.

Hand in your code for version3.

```
def version3(input1, input2):
    output = Alignmt()
    _version3(output, input1, input2)
    print(output)

def _version3(best, str1, str2, x=0, y=0, path="", score=0):
    if x > len(str1) or y > len(str2):
    return
    if x == len(str1) and y == len(str2):
        if best > score:
            best.score = score
            best.str = path
        return

    _version3(best, str1, str2, x+1, y, path+'-', score+1)
    _version3(best, str1, str2, x, y+1, path+'|', score+1)
    if x < len(str1) and y < len(str2):
        if str1[x] == str2[y]:
            _version3(best, str1, str2, x+1, y+1, path+'\\', score)
        else:
            _version3(best, str1, str2, x+1, y+1, path+'\\', score+1)
```

Dynamic Programming for String Alignment

Question 6: Top-down with Memoization

Your code for version3 was very inefficient as it kept on recomputing the answers to the same subproblems over and over again. Instead, 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 version3 much more efficient?

The only partial answers needed at any given time are those above, left and diagonal from the cell currently being computed.

Question 7: Code

Create version4 that is identical to version3 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. So that you can rerun this code (as you will need to do for the next question), initialize cache when version4 is called with x=-1 and y=-1. 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.
Turn in your code.

Helper function:

```
    def display(arr):
        x = -1
        y = -1
        while arr.get((x,y)):
            while arr.get((x,y)):
                print(arr[(x,y)],end='\t')
                if len(arr[(x,y)].str) < 6:
                    print(end='\t')
                x += 1
            print()
            x = -1
            y += 1
        print()
```

Version4 functions:

```
def version4(input1, input2):
     # init the table's first row and column
     output = {(-1,-1):Alignmt(0)}
     for x in range(len(input1)):
          output[(x,-1)] = Alignmt(x+1,"-"*(x+1))
     for y in range(len(input2)):
          output[(-1,y)] = Alignmt(y+1,"|"*(y+1))

     _version4(output, input1, input2, len(input1)-1, len(input2)-1)
     display(output)
     print(output[(len(input1)-1,len(input2)-1)])

def _version4(arr, str1, str2, x=0, y=0):
     if x >= len(str1) or y >= len(str2):
          return

     above = arr.get((x,y-1))
     if not above:
          above = _version4(arr, str1, str2, x, y-1)
     left = arr.get((x-1,y))
     if not left:
          left = _version4(arr, str1, str2, x-1, y)
     diagon = arr.get((x-1,y-1))

     if above < diagon:
          if above < left:
               arr[(x,y)] = Alignmt(above.score+1, above.str + "|")
     elif left < diagon:
          arr[(x,y)] = Alignmt(left.score+1, left.str + "-")
     elif str1[x] == str2[y]:
          arr[(x,y)] = Alignmt(diagon.score, diagon.str + "\\")
     else:
          arr[(x,y)] = Alignmt(diagon.score+1, diagon.str + "\\")

     return arr[(x,y)]
```

Question 8: Efficiency

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.

The algorithm starts in the bottom-rightmost corner of the table. From there it must compute its above, left and diagonal cells. Continuing, the above cell must compute its above, diagonal, and left cells. When we return to the bottom-rightmost cell, the diagonal has already been computed but the left has not, and when the left is computed it must compute its own left. If if the best alignment has already been found up to the bottom-rightmost cell, before computing its left, the last cell can not be determined until its left has. Even input pairs that have very basic optimal alignments, that can, in theory, be found quickly, the whole table must be filled out to verify.

Question 9: Efficiency: Part 2

Measure the speed of your code. For this, vary the array sizes by 10 from size 10 to 500. Your code should run in time $\Theta(n^2)$. For each run, determine what the constant c
should assuming it runs in time cn^2. You should see that this constant is similar for arrays of size 10 to 500.

Function call:

```
exp_test(4,lower=10,upper=501,inc=10)
```

Output (every 10 lines):

```
n:     time:
100     0.5304750011418946 ms
200     1.8385579969617538 ms
300     3.411352001421619 ms
400     9.678307003923692 ms
500     15.613442999892868 ms
```

Modified table:

```
n:     time:                         |c (approx):
100     0.5304750011418946 ms     |0.000053
200     1.8385579969617538 ms     |0.000046
300     3.411352001421619 ms      |0.000038
400     9.678307003923692 ms      |0.000061
500     15.613442999892868 ms     |0.000062
```

Values of c vary only slightly, which is to be expected, ranging from roughly 0.00004 to 0.00006.

Question 10: Efficiency: Part 3

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

Simply, this algorithm must be n^2 (assuming both inputs are similar in size), because every cell in the n by n table is filled, and there is no place in the algorithm where a cell will ever be computed a second time.

Question 11: 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.

The algorithm in the text moves through the table column by column, row by row, filling each cell in order. This type of "traversal" makes sense, because there will never be a time when a cell does not have its above, left and diagonal cell populated.
The algorithm implemented in my assignment works because it starts at the bottom right cell, the solution, and then calculates other cells as need, moving backward through the table, only calculating a cell once it is needed.
In terms of how the data is literally stored, the textbook uses a multidimensional array, while my implementation uses a hash table (Python dictionary). A hash table has slightly more over head and, technically, a worse case access time of n, however both structures access time has an average case of constant time.