Mark Watson
CS 4720
Homework #4
12/05/2011

Problem 1 Edit Distance Problem

The dynamic programming method I used was the Levenshtein distance algorithm. The way the algorithm works is by building a graph with the one string on the columns and one on the rows. The pseudo code for this algorithm is as follows:

```
s = string 1, m = length
t = string 2, n = length
d = matrix of size m by n
for i = 0 to n: d[m, i] = i
for i = 0 to m: d[i, n] = i
for i = 0 to m:
    for j = 0 to n:
        if char at s[i] = char at t[j]:
            // No change needed
            d[i, j] = d[i - 1, j - 1]
        else:
            d[i, j] = min(
              d[i-1, j] + 1,
              d[i, j-1] + 1,
              d[i-1, j-1] + 1
            )
```

The time complexity for the algorithm can be found by adding up the loops, assuming all of the code in the loops are run at some constant time:

$$\begin{aligned} T = & \sum_{i=0}^{n} C_1 + \sum_{i=0}^{m} C_2 + \sum_{i=0}^{m} \sum_{j=0}^{n} C_3 \\ = & \quad nC_1 + mC_2 + mnC_3 \\ \in & \quad \theta(mn) \end{aligned}$$

The implemented algorithm is attached. The output looks like this:

```
> run sport sort
1
> run computer commuter
1
> run word word
0
> run something nothing
4
```

The first run matches the example in the homework problem; adding one letter to "sort" results in "sport" and giving an edit distance of 1. In "computer" and "commuter" we can change one letter. The third example has the same word and so the edit distance is 0. Finally, four edits can be applied to "something" to get the word "nothing". ■

Problem 2 Making Change

a) The greedy algorithm for the change problem is simple. It starts at the largest coin and goes to the smaller ones, dividing the amount by the value of the coin each time. For example to find the change for 12 cents it tries dividing by 25 and fails. Then it divides by 10 and succeeds, subtracting 10 from the running total. It then divides the resulting 2 by 5 and fails. Finally, it divides thee 2 by 1 and subtracts 2 x 1 from 2 which gives 0. This completes the algorithm run and we get "$12 = 1(10) + 2(1)$". The Pseudo code is shown below. It needs amount to be set to the amount we're getting change for.

```
workingAmount = amount
for each coin:
    if workingAmount / coin value > 0:

        // We got the number and value of the coin to
        // use for this iteration
        output workingAmount / coin + " * " + coin
    endif
    workingAmount = workingAmount % coin
```

The algorithm is simple, and it's time complexity is clearly linearly proportional to the number of coins in the currency.

b) A good example of where the greedy algorithm fails is the English currency system before 1971[1]. The currency consists of the values: $1, 3, 6, 12, 24, 30, 60$, and 240. The greedy algorithm fails in the case of finding change for 48 pence, which is represented as $30 + 12 + 6$, while the optimal choice is $24 + 24$. This problem would also happen for the US currency if we were to add a 7 cent piece. The currency would be $25, 10, 7, 5, 1$, and if we wanted to make change for 14 we could use the greedy algorithm and get $10 + 4(1)$, while the optimal solution would be $2(7)$. It's a good thing there are no 7 cent coins.

c) The dynamic programming solution to the problem consists of two phases. The first phase calculates the coins to use to have the minimum number of coins. We then loop over the selected coins and find out how many of each one we need to use. For my implementation I also have a third step that prints out the coins. I don't count that in the following pseudo code analysis, since it's not part of the core algorithm.

```
for j = 1 until amount + 1:

    D[j] = Infinity
    for each coin:
        if j > coin and 1 + D[j-coin] < D[j]:
            D[j] = 1 + D[j - 1]

i = amount
while i > 0:

    print i
    n = n - D[i]
```

The time complexity of the code for the first part is easy to find. It's simply proportional to the amount times the number of coins in the currency. The while loop depends on the makeup of the final solution, since the number of times that need to be iterated depend on the configuration. If we let "n" be the number of coins, and "a" be the amount, we can calculate the worst case time complexity as follows:

$$\begin{aligned} T = & \quad n * a * C + a * D \\ \in & \quad O(na) \end{aligned}$$

[1]D. Pearson, "A polynomial-time algorithm for the change-making problem," Operations Research Letters, vol. 33, no. 3, pp. 231-234, May 2005. [Online]. Available: http://dx.doi.org/10.1016/j.orl.2004.06.001

The last term comes from the fact that we will not loop more than the amount, assuming all coins are of value "1" in the worst case.

The implementation for the algorithm is in the same source file as for the greedy algorithm. Here are several examples where the new algorithm works, and the old one does not:

1. Currency: $1, 3, 6, 12, 24, 30, 60$, and 240, Amount: 48
2. Currency: 25,10,7,5,1, Amount: 14
3. Currency: 4,3,1, Amount: 6
4. Currency: 8,7,1, Amount: 21
5. Currency: 6,4,1, Amount: 8

All of these examples won't work with greedy algorithms, while the dynamic programming model will work. ∎

Problem 3 Exponential Time Problems

See attached sheet. ∎

Problem 4 Extra Credit - Branch-and-Bound

I implemented a branch and bound knapsack algorithm based on the Levitin book. I tested it on various input.

```
--> Size: 4, Items: {(weight=2, value=40),(weight=1, value=1)}
> run
Take item: w=2, v=40, v/w=20
Take item: w=1, v=1, v/w=1
--> Size: 10, Items: {
    (weight=4, value=40),
    (weight=7, value=42),
    (weight=5, value=25),
    (weight=3, value=12),
    }
```

```
> run
Take item: w=7, v=42, v/w=6
Take item: w=3, v=12, v/w=4
```

The algorithm finds the optimal items to take, assuming there is only one of each item. ∎