Lachlan Sinclair

Homework 3: CS 326

7/10/2019

Problem 1: Proof by counter example. Consider a rod of length 4 inches. Rods of length 1, 2, 3 and 4 will cost 1, 10, 18, 4 respectively. Therefor the densities price/length of rods of lengths 1, 2, 3 and 4 are 1, 5, 6, 1 respectively. In this scenario the greedy method will first cut the rod to length three since it has the highest density. The left-over rod of length one cannot be cut to a smaller size, therefor the net value from using the greedy method is 19. This is not the optimal solution, cutting the run into two length 2 rods nets a value of 20 which is higher than the result of the greedy method. Therefor the greedy method does not always produce the most optimal results.

Problem 2: Modifying the bottom-up-cut-rod method to account for a cost induced by cutting the rod. (pg.366)

```
cutRod(p, n, cost)

let r[0…n] be a new array

r[0] = 0

for j=1 to n

        q=p[j]

        for i=1 to j-1

                q=max(q, p[i]+r[j-i]-cost)

        r[j]=q

return r[n]
```

Problem 3:

a) I will use the bottom up approach to create a dynamic programing solution for this problem. To store the previously calculated subproblems I will use a 2D array. There will be a loop that calculates the least amount of coins required to make change for values from 1 to A, this will make up the rows of the 2D array. Inside this loop there will be another loop that iterates through the different denominations of coins. This loop will calculate the minimum number of coins required to make change. It does this by comparing the previous columns value to the sum of 1 plus the minimum number of coins of this denomination subtracted from the total value. We must make sure subtracting the denomination doesn't produce a negative number. As the code goes to the next rows it can use previous rows to calculate the minimum needed coins, i.e. 10-(coin of size 6) would then add 1 to the answer for least amount of coins to create a value of 4. Finally, there will be a loop used to iterate back through the table to determine which denomination of coins were used to create the final answer. It will know when a coin was used by comparing values column by column in reverse order looking for a difference, when one is found the denomination of the coin will be logged in the C array and the value of that coin will be used to find the next row to look at. Once complete the C array will be returned.

Pseudo code:

```
makingChange(V, A)

let C[0…A] be a new array

let B[0..A][0..nMax] (nMax being the length if the Vi array)

for i=1 to A

    B[i][0]=positive infinity

for i=0 to iMax

    B[0][i]=positive infinity

for p=1 to A

    for m=1 to nMax

        if V[m] < p

            if 1+B[p-V[m]][nMax] < B[p][m-1]

                B[p][m] = 1+B[p-V[m][nMax]

            Else

                B[p][m]=B[p][m-1]

        Else

            B[p][m]=B[p][m-1]


tempNum = B[A][nMax]

tempVal = A

while tempNum !=0

    for i=nMax to 1
```

```
        if B[i-1][tempVal] != tempNum
            C[i]++
            tempVal -= V[i]
            tempNum--
Return C
```

b) The theoretical runtime of my pseudo code is Θ(nA).

Problem 4:

a) For this program I will create an algorithm like that of the knapsack problem. First, I will read in the data, then create a table of size W N, where N is the number of items in the table and W is the maximum weight any of the family members can carry. I will then go through and fill out the table starting at 0, 0 increasing W first then N, filling out the most optimal solution for each cell of the table. Then it will loop through all the family members, using their max weight, and the total number of items to find the max value they can achieve. It will then back track through the table to determine which items were used to create that value, this is done by detecting changes in value from column to column. During the back track the items used will be written to the output file. This entire process will be repeated for each problem.

Pseudo code:

```
Main
    T = T value from        file
For z=1 to T
    N = N value from file
    let W[0…N] be a new array
    let P[0…N] be a new array
    let B[100][100] be a new 2d array
    for i=0 to N
        W[i] = weight of item from file
        P[i] = price of item from file
    let F = F value from file
    maxM = 0
```

```
let M[0…F] be a new array
for i=0 to F
     M[i] = family members max weight
     If M[i] > maxM          maxM=M[i]
//all data read in now
For h=0 to maxM
     B[0,h] = 0
for i=0 to N
     B[i,0] = 0
     for w=1 to maxM
          if W[w-1] <= w
               if (P[i-1] + B[i - 1][w - W[i - 1]] > B[i - 1][w])
                    B[i][w] = P[i - 1] + B[i - 1][w - W[i - 1]];
               else
                    B[i][w] = B[i - 1][w];
               else
                    B[i][w] = B[i - 1][w];
Sum =0
For i=0 to F
     Sum += B[N][M[i]]
Outfile << sum
For i=0 to F
     int tempVal
     int tempWeight
     For x = N down to 1
```

```
If B[x-1][maxM] != tempVal

    Out file << x

    memW -= W[x - 1];
    tempVal -= P[x - 1];
```

b) The theoretical runtime of my code is Θ(NM) or Θ(FN) or Θ(FM) depending on how large the arrays are relative to each other. The largest two arrays out of the three will dominate the runtime. However, no matter which arrays ends up being the largest, the runtime will always be polynomial. Based off the bounds provided from the problem the most likely variables to dominate are N and M so it will likely be Θ (NW) theoretical run time. The table for each family is made only once and requires N*m operations. FN and FM come from the back tracking, for each family member F the back tracking takes (N+M) operations. FN and FM are from factoring F(N+M).