CS 325 Spring 2019, Homework 3

April 20, 2019

0.1 Rod Cutting, Greedy Algorithm

Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.
Suppose a rod of length 4 and the values of different pieces are given as following:

```
 length  |  1   2   3   4
 price   |  1   5   8   9
```

The maximum obtainable value is 10 (by cutting in two pieces of lengths 2 and 2)

```
In [10]: def CutRodBottomUp(price, length):
             r = [0] * (length + 1)
             for j in range(length):
                 revenue = -1
                 for i in range(j+1):
                     revenue = max(revenue, price[i] + r[j-i])
                 r[j+1] = revenue
             return r[length]

         def CutRodGreedy(price, length):
             if length == 0:
                 return 0
             _, l = max((price[i]/(i + 1), i + 1) for i in range(length))
             revenue = price[l-1] + CutRodGreedy(price, length-l)
             return revenue

         p = [1, 5, 8, 9]
         print('Optimal = %d' % CutRodBottomUp(p, 4))
         print('Greedy  = %d' % CutRodGreedy(p, 4))
```

```
Optimal = 10
Greedy  = 9
```

Using the Greedy Algorithm, we get 9 which is not the optimum value.

0.2 Modified Rod Cutting, Dynamic Programming

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

```
MODIFIED-CUT-ROD(p, n, c)
    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] − c)
        r[j] = q
    return r[n]
```

The major modification required is in the body of the inner for loop, which now reads `q = max(q, p[i] + r[j−i] − c)`. This change reflects the fixed cost of making the cut, which is deducted from the revenue. We also have to handle the case in which we make no cuts (when i equals j); the total revenue in this case is simply $p[j]$. Thus, we modify the inner for loop to run from i to $j-1$ instead of to j. The assignment `q = p[j]` takes care of the case of no cuts. If we did not make these modifications, then even in the case of no cuts, we would be deducting c from the total revenue.

```
In [30]: def ModifiedCutRod(price, length, cost):
             r = [0] * (length + 1)
             for j in range(length):
                 revenue = price[j]
                 for i in range(j):
                     revenue = max(revenue, price[i] + r[j-i] - cost)
                 r[j+1] = revenue
             return r[length]

         p = [1, 5, 8, 9]
         ModifiedCutRod(p, 4, 1)

Out[30]: 9
```

0.3 Making Change

Given coins of denominations (value) $1 = v_1 < v_2 < ... < v_n$, we wish to make change for an amount A using as few coins as possible. Assume that v_i's and A are integers. Since $v_1 = 1$ there will always be a solution. Formally, an algorithm for this problem should take as input an array V where $V[i]$ is the value of the coin of the i^{th} denomination and a value A which is the amount of change we are asked to make. The algorithm should return an array C where $C[i]$ is the number of coins of value $V[i]$ to return as change and m the minimum number of coins it took. You must return exact change so

$$\sum_{i=1}^{n} V[i] \cdot C[i] = A$$

The objective is to minimize the number of coins returned or:

$$m = min \sum_{i=1}^{n} C[i]$$

a. Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins needed to make change for A.

   The following procedure implements the computation in a straightforward, top-down, recursive manner.

```
COIN-CHANGE(V[1..n], A)
    if A == 0
        return 0
    q = A
    i = 2
    while i <= n and A >= V[i]
        q = min(q, 1 + COIN-CHANGE(V, A - V[i]))
        i = i + 1
    return q
```

   Procedure COIN-CHANGE takes as input an array $V[1..n]$ of coins and an integer amount A, and it returns the minimum number of coins needed to make change for A. If $A = 0$, no coins are needed, and so COIN-CHANGE returns 0 in line 2. Line 3 initializes the minimum number of coins q to A, which is the number of coins needed if we were to use all 1 dollar coins. We're using a while loop because we don't need to continue the loop once the coin value exceeds our amount A, assuming that V is in sorted order.

The time complexity of above solution is exponential. We can reduce the Time Complexity significantly by using Dynamic programming.

```
BOTTOM-UP-COIN-CHANGE(V[1..n], A)
    let r[0..A] be a new array
    r[0] = 0
    for j = 1 to A
        q = j
        i = 2
        while i <= n and j >= V[i]
            q = min(q, 1 + r[j - V[i]])
            i = i + 1
        r[j] = q
    return r[A]
```

b. What is the theoretical running time of your algorithm?

The theoretical running time of the above solution is $O(nA)$.

```
In [55]: def BottomUpCoinChange(coins, amount):
             r = [[0 for _ in range(len(coins))] for _ in range(amount + 1)]
             for j in range(amount):
                 q = [j + 1]
                 i = 1
                 while i < len(coins) and j + 1 >= coins[i]:
                     if sum(q) > 1 + sum(r[j + 1 - coins[i]]):
                         q = list(r[j + 1 - coins[i]])
                         q[i] += 1
                     i += 1
                 r[j + 1] = q
             return r[amount], sum(r[amount])

         coins = [1, 5, 6, 9]
         A = 24
         C, m = CoinChangeBottomUp(coins, A)
         print('Amount = %d' % A)
         print(coins)
         print(C)
         print(m)
```

```
Amount = 24
[1, 5, 6, 9]
[0, 0, 1, 2]
3
```

0.4 Shopping Spree

Acme Super Store is having a contest to give away shopping sprees to lucky families. If a family wins a shopping spree each person in the family can take any items in the store that he or she can carry out, however each person can only take one of each type of item. For example, one family member can take one television, one watch and one toaster, while another family member can take one television, one camera and one pair of shoes. Each item has a price (in dollars) and a weight (in pounds) and each person in the family has a limit in the total weight they can carry. Two people cannot work together to carry an item. Your job is to help the families select items for each person to carry to maximize the total price of all items the family takes. Write an algorithm to determine the maximum total price of items for each family and the items that each family member should select.

a. Give a verbal description and pseudo-code for your algorithm. Try to create an algorithm that is efficient in both time and storage requirements.

   For every item, there can be two cases:

   1. The item is included in the optimal subset.
   2. The item is not included in the optimal set.

   Therefore, the maximum value that can be obtained from n items is the maximum of the following two values.

   1. Maximum value obtained by $n-1$ items and capacity C (excluding n^{th} item).
   2. Value of n^{th} item plus maximum value obtained by $n-1$ items and capacity C minus weight of the n^{th} item (including nth item).

   If the weight of n^{th} item is greater than capacity C, then the n^{th} item cannot be included and case 1 is the only option.

   Here's the pseudocode for the algorithm using Dynamic Programming:

```
KNAPSACK(p[1..n], w[1..n], C):
    let r[0..n][0..C] be a new array
    for i = 0 to n
        for j = 0 to C
            if i = 0 or j = 0
                r[i][j] = 0
            else if w[i] <= j
                r[i][j] = max(p[i] + r[i - 1][j - w[i]], r[i - 1][j])
            else
                r[i][j] = r[i - 1][j]
    return r[n][C]
```

   To obtain a list of items, we will need to back track through the completed table.

```
In [44]: def Knapsack(price, weights, cap):
             n = len(price)
             r = [[0 for _ in range(cap + 1)] for _ in range(n + 1)]

             # Build table r[][] in bottom up manner
             for i in range(n + 1):
                 for j in range(cap + 1):
                     if i == 0 or j == 0:
                         r[i][j] = 0
                     elif weights[i - 1] <= j:
                         r[i][j] = max(price[i - 1] + r[i - 1][j - weights[i - 1]],  r[i - 1][j]
                     else:
                         r[i][j] = r[i - 1][j]

             # Stores the result of Knapsack
             res = r[n][cap]

             items = [] # list to store the items
             w = cap
             for i in range(n, 0, -1):
                 if res <= 0:
                     break

                 # Either the result comes from the top (K[i-1][w]) or
                 # from (val[i-1] + K[i-1] [w-wt[i-1]]) as in Knapsack
                 # table. If it comes from the latter one, it means the
                 # item is included.
                 if res == r[i - 1][w]:
                     continue
                 else:
                     items.insert(0, i)
                     res -= price[i - 1]
                     w -= weights[i - 1]

             return items, r[n][cap]

         p = [32, 43, 26, 50, 20, 27]
         w = [16, 12, 4, 8, 3, 9]
         caps = [25, 23, 21, 19]
         sum = 0
         items = []
         for c in caps:
             item, res = Knapsack(p, w, c)
             sum += res
             items.append(item)
         print(sum)
         for item in items:
             print(item)
```

```
435
[3, 4, 5, 6]
[2, 4, 5]
[3, 4, 6]
[3, 4, 5]
```

b. What is the theoretical running time of your algorithm for one test case given N items, a family of size F, and family members who can carry at most M_i pounds for $1 \leq i \leq F$.

   For one test case, the theoretical running time of the above algorithm would be $O(NMF)$.