

Brian Allison

CS 325 - Homework Assignment 3

Problem 1: (2 points) Rod Cutting: (from the text CLRS) 15.1-2

Consider an array P that contains the price of a rod for a given length i at each index i .

length (i)	1	2	3	4	5	6	7	8
price ($P[i]$)	1	3	4	8	11	12	13	14
price/ i	1	1.5	1.3	2	2.2	2	1.9	1.8

Now, suppose we want to find the optimal number of cuts to maximize the price at length 8. Based on the greedy strategy, the first cut we make will be at the length that gives the highest price per inch. That means we would make a cut at 5.

Again, we make a cut that gives us the highest value per inch. Since we only have 3 inches left, we have to select 3, 2, or 1. We make a 2 inch cut for a value of 3. Then, we make a final 1 inch cut, which is our only option. We then have a combined value of $P[5] + P[2] + P[1] = 11 + 2 + 1 = 14$

If the greedy strategy always worked, 14 would be the optimal value for a rod of length 8. However, this is not the case. We could cut an 8 inch rod at 4 inches and keep the other 4 inch half for a combined value of $P[4] + P[4] = 8 + 8 = 16$

Therefore, the greedy strategy would not always work.

Problem 2: (3 points) Modified Rod Cutting: (from the text CLRS) 15.1-3

Code from textbook was used as a reference for this problem

Pseudocode:

c is cost per cut

```
rodcut (P, n, c)
    initialize array R
    rodcut2(p, n, c, R)
```

```
rodcut2(P, n, c, R)
    if  $R[n] \geq 0$ 
        return  $R[n]$ 
    if  $n == 0$ 
         $R[n] = 0$ 
        return  $R[n]$ 
    else
         $q = -\infty$ 
        for  $i = 1$  to  $n$ 
            if  $i == n$ ,  $c = 0$ 
                 $q = \max(q, p[i] + \text{rodcut2}(P, n - i, c, R) - c)$ 
         $R[n] = q$ 
        return  $q$ 
```

Explanation: Each cut now has a cost associated with it, so each time we call rodcut2, we have to subtract the cost of that cut. The only case where we don't subtract the cost is when $i == n$, which is just the price of the rod at that length.

Problem 3: (6 points) Product Sum

Given a list of n integers, v_1, \dots, v_n , the product-sum is the largest sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors.

For example, given the list 4, 3, 2, 8 the product sum is $28 = (4 \times 3) + (2 \times 8)$, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is $19 = (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2$.

a) Compute the product-sum of 2, 1, 3, 5, 1, 4, 2.
 $2 + 1 + 3 \times 5 + 1 + 4 \times 2 = 27$

b) Give the dynamic programming optimization formula $OPT[j]$ for computing the product-sum of the first j elements.

$OPT[j] = 0$ if j is 0

$OPT[j] = v_j$ if j is 1

$OPT[j] = \max(OPT[j-1] + v_j, OPT[j-2] + v_{j-1} \times v_j)$

c) What would be the asymptotic running time of a dynamic programming algorithm implemented using the formula in part b).

$T(n) = \# \text{ of sub-problems} \times \text{time per sub-problem}$

We have N sub-problems and the time per sub-problem is $\Theta(1)$.

Therefore $T(n) = \Theta(n)$. Without dynamic programming, it would be $O(n^2)$.

Problem 4: Making Change: Given coins of denominations (value) $1 = v_1 < v_2 < \dots < v_n$, 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:

- 1 An array V where $V[i]$ is the value of the coin of the i^{th} denomination.
- 2 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

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

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 to make change for A.

Description:

A 2d array is used to track the optimal number of coins at each amount and denomination. We go through each index of the 2d array, inserting the optimal number of coins, given the denominations available (any denomination equal to or less than the value at the current denomination index) and current amount of change to be given. This is done by choosing the minimum between two options: one added to the value at the index that is one denomination amount away from the current amount staying on the y axis of the 2d array, or the amount that is 1 denomination level above the current on the x axis of the 2d array. The smaller value is inserted at the current index. This continues until we get to the final index. The value inserted at the final index is the smallest possible amount of coins we can use to get the change for A.

Each index actually contains an object containing two integers rather than no object with one integer. One of the integers is the minimal amount of coins needed, and the other is the option used when determining the minimal amount of coins. Option 1 corresponds to getting the minimal amount from the index above on the y axis. This would also be the optimal solution when the denomination that corresponds to the current index cannot be used. Option 0 corresponds to getting the minimal amount from the index that is one current denomination away on the x axis (+ 1). By tracking the option chosen at each index, we can easily backtrack in order to produce an array that contains the amount of coins of each denomination used. Each time option 0 is selected, we add 1 for that denomination and continue moving up and left on the y and x axes until we hit 0 on the x axis.

Pseudocode:

```
min_coins (V, A)
    coins[][] is 2d array of size V.length, A + 1
    for j = 0 to A
        for i = 0 to V.length - 1
            if j == 0, coins[i, j].num = 0           init left col
            else if i == 0, coins[i, j].num = j       init top row
            else if V[i] <= j, \
                op0 = 1+coins[i, j - V[i]].num
                op1 = coins[i - 1, j].num
                if op0 >= op1
                    coins[i, j].num = op0
                    coins[i, j].op = 0
                else
                    coins[i, j].num = op1
                    coins[i, j].op = 1
            else, coins[i, j] = coins[i - 1, j]
```

below code is for creating array of number of denominations used

```
x = A
y = V.length - 1
m = coins[y, x]
C[] is array of length V.length
init all indexes in C to 0
while (x > 0)
    if coins[y, x].op == 1
        y = y - 1
    else
        C[y] = C[y] + 1
```

$x = x - V[y]$

return C and m

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

Based on the number of times we run through the two ‘for’ loops, worst case is $\Theta(MN)$ with M being the number of denominations and N being the amount of change that is needed.

1

Problem 5: (10 points) Making Change Implementation

Submit a copy of all your files including the txt files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named amount.txt.

You may use any language you choose to implement your DP change algorithm. The program should read input from a file named “amount.txt”. The file contains lists of denominations (V) followed on the next line by the amount A.

Example amount.txt:

```
1 2 5
10
1 3 7 12
29
1 2 4 8
15
```

In the above example the first line contains the denominations $V=(1, 2, 5)$ and the next line contains the amount $A = 10$ for which we need change. There are three different denomination sets and amounts in the above example. A denomination set will be on a single line and will always start with the 1 “coin”.

The results should be written to a file named change.txt and should contain the denomination set, the amount A, the change result array and the minimum number of coins used.

Example change.txt:

```
1 2 5
10 0 0
2 2

1 3 7 12
29
0 1 2 1
4
1 2 4 8
15
1 1 1 1
4
```

In the above example, to make 29 cents change from the denomination set (1, 3, 7, 12) you need 0: 1 cent coin, 1: 3 cent coin, 2: 7 cent coins and 1: 12 cent coin for a total of 4 coins.

Problem 6: (4 points) Making Change Experimental Running time

- a) Collect experimental running time data for your algorithm in Problem 4. Explain in detail how you collected the running times.

I did three separate tests of 10 trials. For the denomination test, I increased the number of denominations each time with a constant amount. For the amount test, I increased the amount each time with a constant number of denominations. For the denomination * amount test, I increased both each time. Denominations were chosen at random for each separate trial.

- b) On three separate graphs plot the running time as a function of A, running time as a function of n and running time as a function of nA. Fit trend lines to the data. How do these results compare to your theoretical running time? (Note: n is the number of denominations in the denomination set and A is the amount to make change)

Amount has a linear affect on time. Denomination has a linear effect on time. And, amount * denomination has a quadratic effect on time.



