CS352 (Winter 2018) - Homework 3

Marc Tibbs (tibbsm@oregonstate.edu)

Due Date: January 28, 2018

Problem 1: *(2 points)* **Rod Cutting:** (from the text CLRS) 15.1-2
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 $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

Counter example:
Let the length of a rod $n = 4$ with the following prices and densities:

length (i)	1	2	3	4
price (p_i)	1	5	8	10
density (p_i/i)	1.0	2.5	2.7	2.5

The greedy algorithm would make a first cut of length 3 since it has the maximum density of 2.7. The final price of the entire rod would then be p_3 + p_1 = 8 + 1 = **9**, which is not the maximum price you could get from the rod. For example, if the rod is cut into a to two lengths of 2 you would get p_2 + p_2 = 5 + 5 = **10** even though the density of length 2 is less than length 3. This shows that the greedy strategy does not always determine an optimal way to cut rods.

Problem 2: *(3 points)* 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.

A modified version of the Bottom-Up-Cut-Rod algorithm from page 166 from the textbook is shown down below. Line 5 is modified so that the for-loop checks the values i to j-1, so that the algorithm checks only the values where a cut is made. Then on line 7, after finding the maximum revenue where cuts are made (i to j-1), the algorithm checks whether the revenue of the rod where no cuts are made or the revenue where a cut is made (q-c) is greater. The maximum value is then stored in the array r.

Bottom-Up-Cut-Rod (p,n)

```
let r[0...n] be a new array
r[0] = 0
for j = 1 to n
    q = $-\infty$
    for i = 1 to j-1
        q = max(q, p[i]+r[j-i])
    q = max(q-c, p[i])
    r[j] = q
return r[n]
```

Problem 3: *(6 points)* Given a list of n integers, v_1, ... , 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 x 3) + (2 x 8), and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is 19 = (2 x 2) + 1 + (3 x 2) + 1 + (2 x 2) + 1 + 2.

(a) Compute the product-sum of 2, 1, 3, 5, 1, 4, 2.
The product sum is <u>27</u> = 2 + 1 + (3 x 5) + 1 + (2 x 4).

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

Given a sequence $X = [X_1, X_2, ..., X_j]$:

$$OPT[j] = \begin{cases} 0 & \text{if j = 0} \\ X_{j-1} & \text{if j = 1} \\ \max(OPT[j-1] + X_{j-1},\, OPT[j-2] + X_{j-1} * x_{j-2}) & \text{if j > 1} \end{cases}$$

(c) What would be the asymptotic running time of a dynamic programming algorithm implemented using the formula in part b.
$T(n) = \Theta(n)$

Problem 4: *(5 points)* 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 ith denomination.
- 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

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

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 to make change for A.

An algorithm that finds the minimum number of coins to make change using DP would use an array to keep track of values of previous results. The base case (0) would be placed in the first slot of the array. Then the algorithm would work up the different change amounts starting at one, since we know that $v_1 = 1$ the arr[1] = arr[0]+1 = 1. Next the algorithm would continue on to finding the minimum amount of change to make 2. If there is no denomination less than to other than 1, then arr[2] = arr[1]+1 = 2. On the other hand if there is a denomination that is equal to 2, then arr[2] = arr[0]+1. The algorithm would work in this fashion until it reached the searched for amount. Code for the algorithm is shown below.

```
std::vector<int> makeChange(int vals[], int size, int A){

    //return vector
    std::vector<int> final(size);

    //vector to track coin amts
    std::vector<int> track(A + 1);

    //array to store results
    std::vector<int> arr(A + 1);

    //base case
    arr[0] = 0;

    //arr[i] = inf
    for (int i = 1; i <= A; i++) {
        arr[i] = std::numeric_limits<int>::max();;
    }

    //for i = 1 to A
    for (int i = 1; i <= A; i++){

        //test vals < i
        for (int j = 0; j < size; j++){

            if (vals[j] <= i){
                int change = arr[i-vals[j]];

                //store result in arr
                if (change + 1 < arr[i] && change < std::
    numeric_limits<int>::max()){
```

```
                        arr[i] = change + 1;
                        track[i] = j;

                    }
                }
            }
        }

        int cnt = A;
        while(cnt > 0){
            final.at(track[cnt])++;
            cnt = cnt - vals[track[cnt]];
        }

        final.push_back(arr[A]);

        return final;
}
```

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

The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Problem 5: *(10 points)* 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.

Copy of code files and README file that explains how to compile and run the code are compressed in a ZIP file and have been submitted to TEACH. Program runs with input file named amount.txt.

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.

Data was collected in two different ways. The first way entailed using static coin denominations [1, 5, 6, 10, 15, 20, 25, 30, 35, 50]. So n_2 = n[1,5], n_4 = n[1,5, 15, 20] and so on. The second method I used to collect data was created using randomly generated coin denominations, where the first number in the set was always 1. The two results were very similar to one another as shown in the table below.

Coin Denominations(n)	**Amount(A)**	**n*A**	**Avg. Time(s) (Static n[i])**	**Avg. Time(s) (Random n[i])**
2	10000000	20000000	0.688374	0.701102
2	20000000	40000000	1.32246	1.31001
2	30000000	60000000	2.006	1.97326
2	40000000	80000000	2.71578	2.83373
4	10000000	40000000	0.862355	0.818395
4	20000000	80000000	1.68899	1.71955
4	30000000	120000000	2.58554	2.50609
4	40000000	160000000	3.50653	3.20781
6	10000000	60000000	0.98345	0.966986
6	20000000	120000000	1.87283	2.21451
6	30000000	180000000	2.9119	2.96573
6	40000000	240000000	3.9181	3.80999
8	10000000	80000000	1.17116	1.10952
8	20000000	160000000	2.29825	2.4436
8	30000000	240000000	3.39266	3.74226
8	40000000	320000000	4.50221	4.37357
10	10000000	100000000	1.37689	1.30852
10	20000000	200000000	2.69385	2.81962
10	30000000	300000000	3.9863	4.26106
10	40000000	400000000	5.52955	5.18698

(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)

Graphs for gathered data are shown on the next pages. The graphs show that the data has a strong linear pattern with R^2 values of at least .97 and above. Although the theoretical running time of the algorithm is $\Theta(mn)$, which is not necessarily linear, but since each element of our input is only touched once, the resulting data is more linear. Since an input of size $\Theta(mn)$ will have a running time of $\Theta(mn)$ we can assume that it will be more linear.

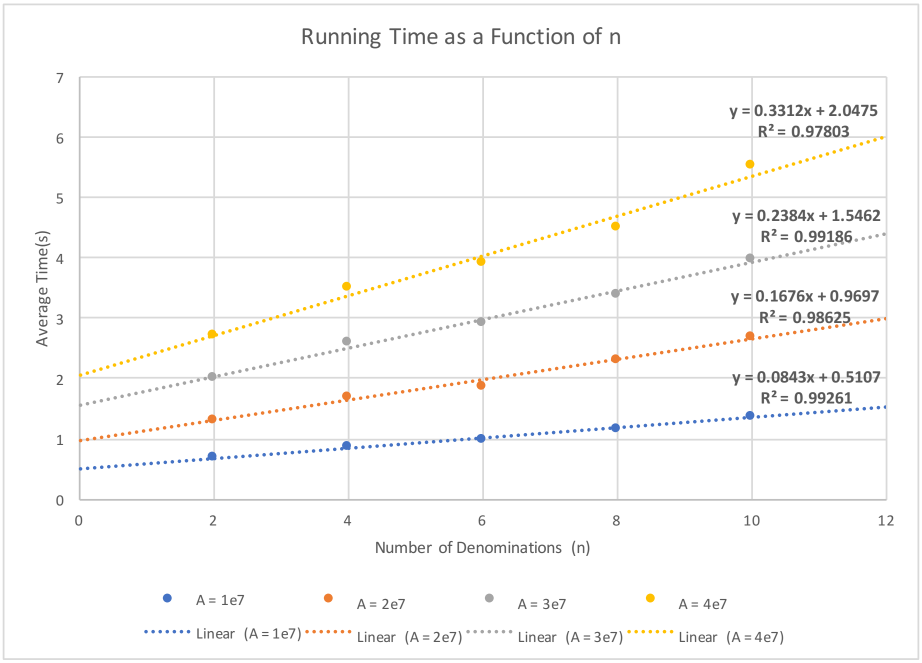

Figure 1: Average Time vs. Number of Coin Denominations(n)

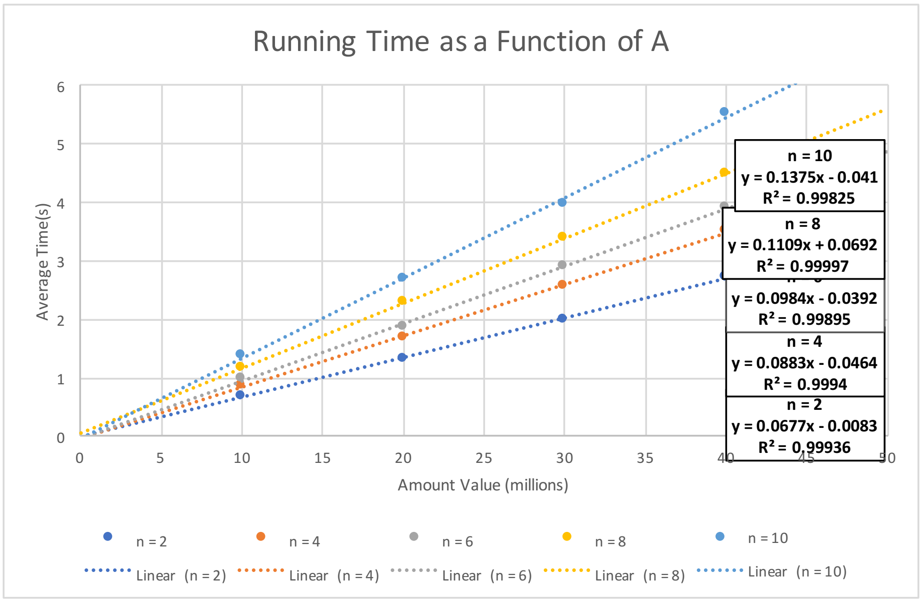

Figure 2: Average Time vs. Amount Values(A)

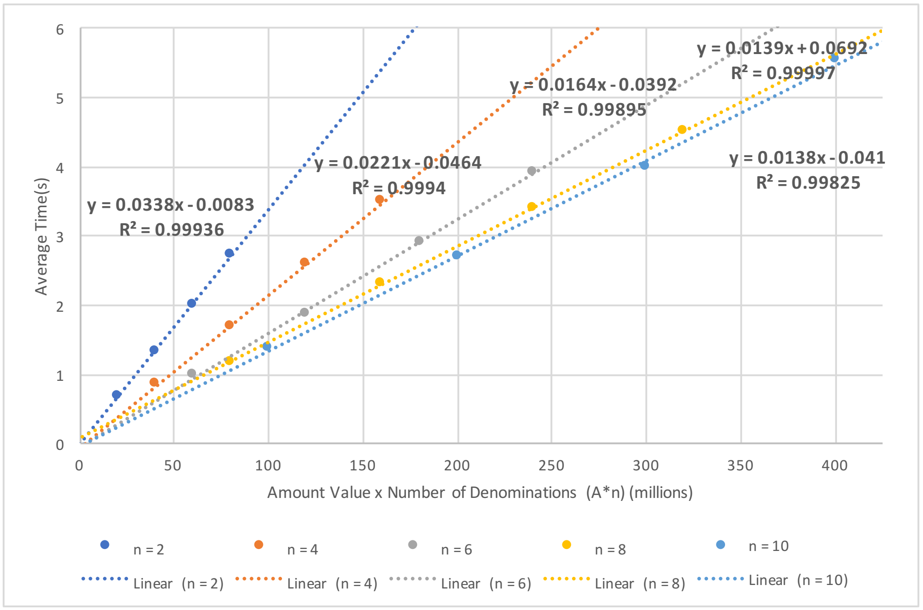

Figure 3: Average Time vs. Amount Values x Coin Denominations (A*n)

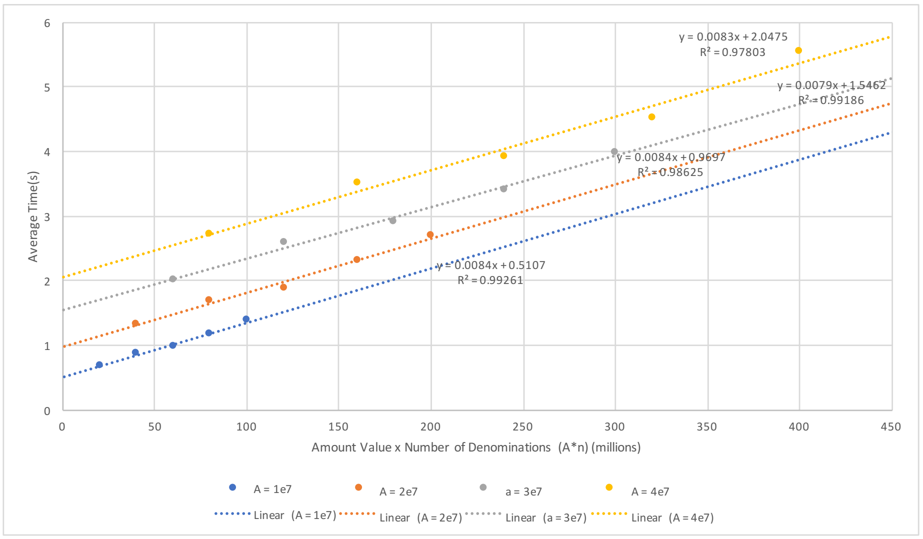

Figure 4: Average Time vs. Amount Values x Coin Denominations (A*n)