Josh Huff, huffj@oregonstate.edu
Professor Schutfort
CS 325 – Winter 2018
January 28, 2018

Homework 3

---

1. 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 <= i <= n, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length n - i.

[1, 24, 39, 48] with respective densities of 1, 12, 13, and 12.

An algorithm using a greedy approach that emphasizes density would cut a piece of length 3 and have a piece of length 1 remaining for a total value of 40. However, the optimal choice is to leave the bar in one piece, because the value of such is 48.

2. 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.

Below is a Python implementation of the CLRS pseudocode for a memoized rod-cutting algorithm's helper function, with the differences for a cut cost included in bold. If the value is the full length of the rod, the cut cost is set to zero and nothing is subtracted. Otherwise, each subproblem will deduct the cut_cost in its call so that the number of cuts will be equal to one less than the number of pieces necessary to reach the optimal price (e.g. if the optimal price involves 3 pieces of rod, the number of cuts will be 2).

```
def memoized_cut_rod_aux(price, n, r):

    if r[n] >= 0: return r[n]
    if n == 0: return q
    else:
        q = -(math.inf)
        for i in range(n):
            if i > n - 2:
                cut_cost = 0
            else:
                cut_cost = 2 # cut_cost can be any integral value
            q = max(q, price[i] - cut_cost + memoized_cut_rod_aux(price, n - i - 1, r))
            r[n] = q
    return q
```

3. 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 × 3) + (2 × 8), and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is 19 = (2 × 2) + 1 + (3 × 2) + 1 + (2 × 2) + 1 + 2.

a. Compute the product-sum of 2, 1, 3, 5, 1, 4, 2.

The product-sum is 27, and can be expressed as 2 + 1 + (3 * 5) + 1 + (4 * 2)

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

The formula is OPT[J] = max(OPT[J – 1] + V[J], OPT[J – 2] + V[J – 1] * V[J])

Using this formula and OPT[1] = V[0] and OPT[2] = max(V[0] + V[1], V[0] * V[1]) as base cases, the results of calling the product-sum function with a J value of 7 as in problem 3a are:

OPT[1] = 2 (base case)
OPT[2] = 3 (base case)
OPT[3] = max(OPT[2] + 3, OPT[1] + 3) = 6
OPT[4] = max(OPT[3] + 5, OPT[2] + 15) = 18
OPT[5] = max(OPT[4] + 1, OPT[3] + 5) = 19
OPT[6] = max(OPT[5] + 4, OPT[4] + 4) = 23
OPT[7] = max(OPT[6] + 2, OPT[5] + 8) = 27

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

The asymptotic run time of an algorithm using either memoization or the bottom-up approach and the formula in 6b would be $\Theta(n)$.

4. 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.

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

Finding the minimum number of coins to make change for an amount (A) involves creating two arrays and iterating over all possible values up to A using each denomination in V.

The first array (min_coins_required) holds the minimum number of coins required of a certain denomination to satisfy an amount. If a value is less than the coin denomination currently being considered, that value is skipped entirely. If the current minimum at a given index of min_coins_required (i) is greater than the sum of one and the value indexed at the current value minus the face value of the current coin being considered (i – coin_value), then the previously optimal solution has been proven not optimal by contradiction, and a new minimum is set.

The second array (next_coin) holds the index of the next coin denomination to be subtracted from the total amount. After the coin's value is subtracted from the total amount, the value of next_coin at the index of the new amount will point to the next coin. This process is repeated until all necessary coins for summing to the original amount are chosen.

Pseudocode – using A (integer: amount of change sought) and V (array: available denominations)

```
min_coins_required = an array of length A + 1, first element is guaranteed to be zero
next_coin = an array of length A + 1

for j, coin_value in V:
    for i in min_coins_required:
        if i >= coin_value:
            if min_coins_required[i] > 1 + min_coins_required[i - coin_value]:
                first_coin[i] = j
            min_coins_required[i] = min(min_coins_required[i], 1 + min_coins_required[i - coin_value])

solution = an empty array
while A > 0:
    curr_coin = next_coin[A]
    add V[curr_coin]) to solution
    A =  A - V[curr_coin]

result = count instances of each denomination appearing in solution
return result
```

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

Instead of being exponential like a naive solution, this algorithm runs in pseudo-polynomial time. It is O(V * A) = O(n * m).

6. Making Change Experimental Running Time

6a. Collect experimental running time data for your algorithm in Problem 4. Explain in detail how you collected the running times.

I populated a list called “possible_denominations” with values 1, 2, 5, 10, 15, 20, 22, 25, 50, and 100 and passed slices of variable length with each amount. As suggested by a peer tutor, I used n from 3 to 10 and amounts from 1000 to 5000 with a stride of 500, where n is the length of the array containing coin denominations and the amount equals the target number of change.

These arguments were passed with the make_changes function to the timeit Python library and the results, including the value of n, A, n * A, and the average run-time (ten runs for each pair of arguments) of each iteration were written to a text file.

In plotting the results, I chose the median values of n and A (7 and 3000, respectively) as constants and took care to eliminate duplicate products of nA.

(n)	(A)	(n * A)	Avg Run	(n)	(A)	(n * A)	Avg Run
3	1000	3000	0.0148	7	1000	7000	0.0273
3	1500	4500	0.0220	7	1500	10500	0.0419
3	2000	6000	0.0290	7	2000	14000	0.0559
3	2500	7500	0.0367	7	2500	17500	0.0708
3	3000	9000	0.0437	7	3000	21000	0.0859
3	3500	10500	0.0513	7	3500	24500	0.0996
3	4000	12000	0.0588	7	4000	28000	0.1132
3	4500	13500	0.0662	7	4500	31500	0.1279
3	5000	15000	0.0739	7	5000	35000	0.1432
4	1000	4000	0.0175	8	1000	8000	0.0306
4	1500	6000	0.0264	8	1500	12000	0.0470
4	2000	8000	0.0358	8	2000	16000	0.0631
4	2500	10000	0.0448	8	2500	20000	0.0789
4	3000	12000	0.0538	8	3000	24000	0.0954
4	3500	14000	0.0637	8	3500	28000	0.1119
4	4000	16000	0.0727	8	4000	32000	0.1285
4	4500	18000	0.0817	8	4500	36000	0.1469
4	5000	20000	0.0910	8	5000	40000	0.1648
5	1000	5000	0.0208	9	1000	9000	0.0344
5	1500	7500	0.0314	9	1500	13500	0.0525
5	2000	10000	0.0428	9	2000	18000	0.0703
5	2500	12500	0.0532	9	2500	22500	0.0891
5	3000	15000	0.0643	9	3000	27000	0.1065
5	3500	17500	0.0758	9	3500	31500	0.1228
5	4000	20000	0.0869	9	4000	36000	0.1415
5	4500	22500	0.0970	9	4500	40500	0.1646
5	5000	25000	0.1093	9	5000	45000	0.1807
6	1000	6000	0.0247	10	1000	10000	0.0377
6	1500	9000	0.0367	10	1500	15000	0.0579
6	2000	12000	0.0490	10	2000	20000	0.0776
6	2500	15000	0.0617	10	2500	25000	0.0978
6	3000	18000	0.0747	10	3000	30000	0.1175
6	3500	21000	0.0896	10	3500	35000	0.1373
6	4000	24000	0.1010	10	4000	40000	0.1577
6	4500	27000	0.1131	10	4500	45000	0.1778
6	5000	30000	0.1254	10	5000	50000	0.1958

6b. 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)

The actual results appear linear, but would likely show a steeper growth with larger values of n and A. The confidence for each graph was over 99.

See next page.

Figure 3 – Run-time as a function of A (n = 7)

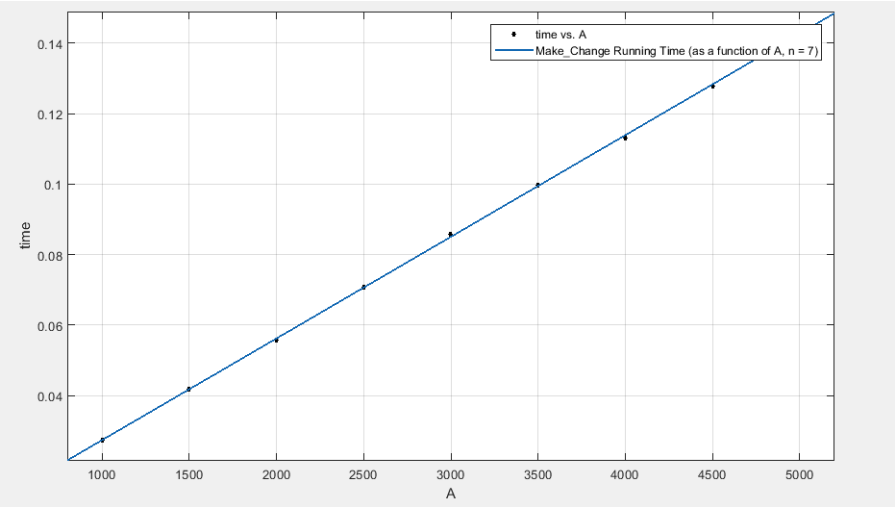

Figure 2 – Run-time as a function of n (A = 3000)

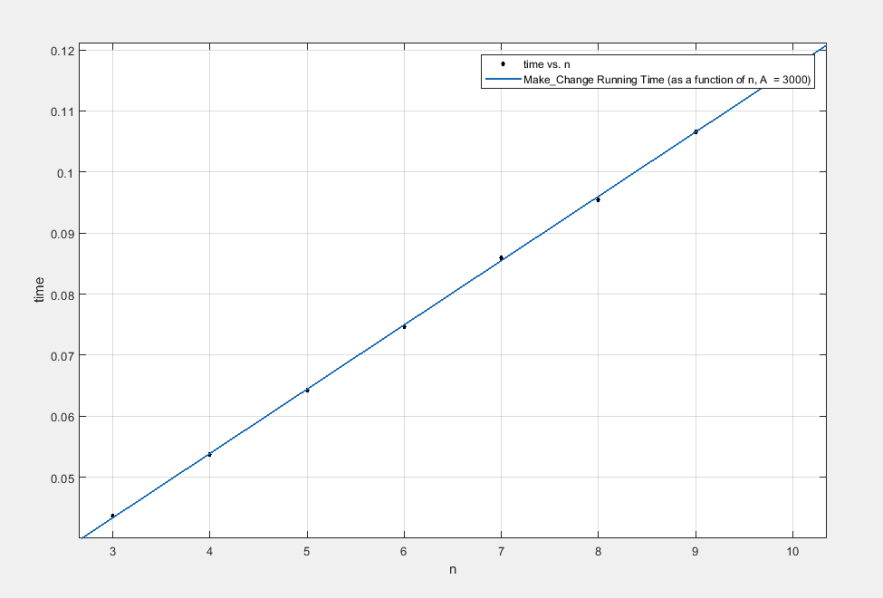

Figure 3 – Run-time as a function of nA