James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Homework 3

Due Date: Sunday, April 22, 2018

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

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 $\frac{p_i}{i}$, 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 $n - i$.

ANSWER:
Below is a counterexample to demonstrate that the described greedy strategy does not always determine an optimal way to cut rods.

Assume we're utilizing the following Lengths and Prices to find densities…

Length (i)	Price (p_i)	Density (p_i/i)
1	**1**	$\frac{1}{1} = 1$
2	**18**	$\frac{18}{2} = 9$
3	**30**	$\frac{30}{3} = 10$
4	**32**	$\frac{32}{4} = 8$

As demonstrated above, through applying the greedy algorithm we can determine that the densest rod (most expensive) has a length of 3 inches. With this information, the greedy algorithm would then attempt to initially cut the next rod (4 inches) into a 3-inch segment, turning the rod into a 3-inch piece and a 1-inch piece. When combining the value of those two lengths we would receive a total price of \$31 ($\$1 + \$30$). This is less money than the \$36 maximum (optimal) profit that could have been obtained but cutting the rod into two 2-inch segments. It's also less than the \$32 that could have been obtained by simply leaving the rod uncut in its original 4-inch state.

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

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

15.1-3
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.

ANSWER:
Below is pseudocode for the dynamic-programming algorithm to solve the modified rod-cutting problem.

```
cutRod_Modified(p, l, c)
        res := [0, 1, …, n]
        res [0] = 0
        for j in range (1, l)
                q := p [j]
                for i in range (1, (j - 1))
                        q := max {q, (p [i] + res [j - 1] - c}
                end of inner for loop
                res [j] := q
        end of outer loop
        return res[n]
```

The first change to the original rod-cutting algorithm was to alter the number of values the function expected to include price (*p*), length (*l*) and the new addition of fixed cost (*c*). The function also required an additional nested inner for loop that calculated the sum of the prices of the pieces minus the costs of making the cuts to determine revenue (*q*).

Problem 3: *(6 points)* **Test Time:** OSU student, Benny, is taking his CS 325 algorithms exam which consists of n questions. He notices that the professor has assigned points $\{p_1, p_2, \dots, p_n\}$ to each problem according to the professor's opinion of the difficulty of the problem. Benny wants to maximize the total number of points he earns on the exam, but he is worried about running out of time since there is only T minutes for the exam. He estimates that the amount of time it will take him to solve each of the n questions is $\{t_1, t_2, \dots, t_n\}$. You can assume that Benny gets full credit for every question he answers completely. Develop an algorithm to help Benny select which questions to answer to maximize his total points earned. **Note:** NO partial credit is assigned to problems that are only partially completed.

a) Verbally describe a DP algorithm to solve this problem.

ANSWER:

Since the problems exhibits optimal substructure ("an optimal solution to the problem contains within it optimal solutions to subproblems") a dynamic programming algorithm could be utilized to find its solution. Namely, the knapsack problem could be utilized to solve the problem above and determine the optimal solution. The knapsack problem operates under the idea that you want to maximize the value of the items that you can contain in the limited space of our "knapsack". In this circumstance it would be Benny attempting to obtain the maximum amount of points (p) on the exam based on the number of questions (n) within the set amount of time (T) for the test. Rather than just assuming the questions worth the most points should be what Benny should do first (as a greedy version of the knapsack algorithm would do), the dynamic programming version would break each question down into subproblems to help determine a value for each question and therefore find an optimal solution. The dynamic programming implementation of the algorithm is faster than a recursive version but consumes more memory storing each iteration (subproblem) in a table for later reference.

b) Give pseudo code with enough detail to obtain the running time, include the formula used to fill the table or array.

ANSWER:

```
Time_test (v, w, n, W)
       For j in range (0, W):
              m[0, j] := 0
       end of for loop
       For i in range (1, n)
              For j in range (0, W):
                     if w[i] <= j
                            m[i, j] := max{m[i - 1, j], v[i] + m[i - 1, j - w [i]]}
                     else
                     m[i, j] := m[i - 1, j]
              end inner for loop
       end of for outer loop
```

c) What is the running time of your algorithm? Explain.

ANSWER:

The running time complexity is $\theta(np)$ where n is the number of problems and p is the points available for each individual problem. This running time is obtained because for each problem (p) you must make a comparison for each value from 1 to the total amount (1, 2, 3, …, p) against each number of problems (1, 2, 3, …, n). It's ultimately two linear comparisons multiplied together to determine the overall $\theta(np)$ or $\theta(nW)$ time complexity.

d) Would Benny use this algorithm if the professor gave partial credit for partially completed questions on the exam? Discuss.

ANSWER:

The Knapsack Dynamic programming algorithm does not allow for values of items to be segmented and would therefore not work if partial credit was awarded for attempted answers. That being said, there is a variant Fractional Knapsack method that could be applied in this scenario to help determine the optimal solution. The fractional method allows us to select fractional items rather than performing exclusively whole selections. The fractional knapsack algorithm, unlike the standard binary version, can be solved through means of a greedy strategy.

Problem 4: *(5 points)* **Making Change:** Given coins of denominations (value) $1 = v_n < v_2 < \cdots < v_n$, 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.
- 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 to make change for A.

ANSWER:

The make change dynamic program receives an array (list) of denominations and a total amount. The algorithm uses the denominations to find the minimum amount of coins combinations necessary to equal the amount provided. To accomplish this, it must analyze/evaluate each element in the denominations array with each incremental number from 1 to the total amount. Like the knapsack problem described above, the problem exhibits optimal substructure and can be solved dynamically. This means that for every value between 1 and the total is used to determine how many coins it would take to make that value and that combination is stored for later reference. These values are stored in an array (or table) and used for later evaluations to save processing time. Unfortunately, what we save in time, we use in memory utilization.

```
make_change(V[], A)

        T[0] = 0
        for i in range (1, len(V)):
                T[i] = inf
        End for loop
        for i in range (1, A+1):
                for j in range (1, len(V)):
                        if V[j] <= i
                                if 1 + T[i - d[j]] < T[i]
                                        T[i] := 1 + [i - d[j]]
                End inner for loop

        end of outer for loop
```

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

ANSWER:

The theoretical running time of the making change dynamic programming algorithm is the number of Denominations provided multiplied by the total Amount. The program has two loops, an outer loop that needs to process through each element of the denominations array, and a second inner loop that needs to process from 1 through to the value of the total amount.

$\boldsymbol{n = denominations\ (V)}$
$\boldsymbol{m = amount\ (A)}$

$\boldsymbol{\theta(nm)}$

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.

ANSWER:

My implementation for the Make Change Dynamic Programming algorithm was submitted to TEACH in a zipped folder under the filename make_change.py. The program takes arrays from a file called amount.txt, Finds the minimum amount of coins necessary to create the amount, and writes the results to a file called change.txt. As with Homework 1 & 2, I again used python as my development language. The algorithm was tested on my desktop and the university FLIP servers and confirmed to work correctly. Below is a screen shot from my code of the function that performs my implementation of Make Change Dynamic Programming. Please see my zip file TEACH submission for the full version of the program.

Make Change Dynamic Programming (Python):

```
"""
/*****************************************************************
* Description: make_change function (Dynamic Programming)
* Function receives an amount (total), a list of denominations (coins),
* and a length of the list (number of coins) from main.  Returns the
* minimum number of coins required to reach the given amount, as well as,
* the combination of the coins used.  Uses bottom up dynamic programming.
* Requires two 2D arrays to capture the minimum number of coins and the
* the amount of the coins.
*****************************************************************/
"""

def make_change(amount, denominations):
    # Below is an efficient way of Establishing an array and populating with values
    # Initialize the first element of each internal array to zero and everything else to infinity (used for comparisons)
    total = [[0 if count == 0 else float("inf") for count in range(amount + 1)] # Create an array that is length of the amount that contains
        for count in range(len(denominations))]                                 # arrays that are length of the array of denominations.
    # Example: [[0, inf, inf, inf, inf, inf], [0, inf, inf, inf, inf, inf], [0, inf, inf, inf, inf, inf]]
    #

    # Need an array of arrays in order to capture each coin change and comparison made when evaluating the total array.
    coins = [[0 for count in range(len(denominations))] for count in range(amount + 1)] # Create a 2D array that will hold the number used for each coin
    # Example: [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

    for dCount in range(len(denominations)):                                    # Increment through each denominations value
        for aCount in range(1, (amount + 1)):                                   # Increment from 1 to the amount value
            if aCount < denominations[dCount]:                                  # If the current amount value is less than the current denominations value
                total[dCount][aCount] = total[dCount - 1][aCount]               # Assign the value from the previous denomination in the 2D array
            else:                                                               # Otherwise...
                total[dCount][aCount] = min(total[dCount - 1][aCount],          # If we took the current denomination value
                1 + total[dCount][aCount - denominations[dCount]])              # Assign it to the current total array location
                coins[aCount] = coins[aCount - denominations[dCount]][:]        # Captures the coins as they are processed and adds them to our coins array
                coins[aCount][dCount] += 1                                      # Increment current coin value in the array based on where we are in the loop
    return (total[len(denominations) - 1][amount], coins[amount])               # Return the minimum number of coins and the coin combination
```

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

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.

ANSWER:

To collect the running time data as a Function of the Amount (A), I created a loop that incremented the value of the amount from 10,000 to 100,000 in increments of 10,000. I thought about altering the denominations, but since they have to process through each number (1, 2, 3…n) to the total, I determined that this would have no impact on the run time when evaluating the algorithm on large amount sizes. Therefore, I used the standard USD coin values of 1, 5, 10, and 25 in my array of denominations.

Running Time as a Function of A (Amount)

```
Jamess-MacBook-Pro:Homework_3 hippler$ python make_change_timed.py
Amount: 10000
Time to Complete: 0.0713448524475 seconds
Amount: 20000
Time to Complete: 0.132941961288 seconds
Amount: 30000
Time to Complete: 0.207766056061 seconds
Amount: 40000
Time to Complete: 0.292195081711 seconds
Amount: 50000
Time to Complete: 0.402953147888 seconds
Amount: 60000
Time to Complete: 0.457489013672 seconds
Amount: 70000
Time to Complete: 0.543876886368 seconds
Amount: 80000
Time to Complete: 0.606305122375 seconds
Amount: 90000
Time to Complete: 0.643466949463 seconds
Amount: 100000
Time to Complete: 0.759752035141 seconds
Jamess-MacBook-Pro:Homework_3 hippler$
```

To collect the running time data as a Function of the denominations (n), I created a loop that incremented the amount of denominations from 10,000 to 100,000 in increments of 10,000 (similar to the strategy for gaging the Amount run time). This means that each array would be increased by 10,000 elements each iteration with that array values starting at 1 and ending at n (1, 2, 3, …, n). I set the amount to a static number of 100 since this was a measure of how a growing number of denominations effected the run time rather than an increasingly larger amount.

Running Time as a Function of n (Number of Denominations)

```
Jamess-MacBook-Pro:Homework_3 hippler$ python make_change_DenominationTimed.py
Denominations: 10000
Time to Complete: 0.0632400512695 seconds
Denominations: 20000
Time to Complete: 0.12925696373 seconds
Denominations: 30000
Time to Complete: 0.184139966965 seconds
Denominations: 40000
Time to Complete: 0.265343904495 seconds
Denominations: 50000
Time to Complete: 0.315627098083 seconds
Denominations: 60000
Time to Complete: 0.388108968735 seconds
Denominations: 70000
Time to Complete: 0.468410015106 seconds
Denominations: 80000
Time to Complete: 0.489822149277 seconds
Denominations: 90000
Time to Complete: 0.560252189636 seconds
Denominations: 100000
Time to Complete: 0.624435901642 seconds
Jamess-MacBook-Pro:Homework_3 hippler$
```

To collect the running time data as a Function of the denominations multiplied by the Amount (nA), I created a loop that utilized both procedures individually implemented to calculate the running times of Amount and Denominations in coordination. Both the value of the amount and the length of the denominations start at 100 and increment by 100 until they reach 1,000 (loops 10 times before completion). I initially tried to perform the same scale of 10,000 as before but this was taking longer to run and wouldn't actually affect the results.

Below is the result from executing the program. In the next section (Part B) I present the corresponding graphs and discuss my interpretation of the results.

Running Time as a function of nA (Denominations • Amounts)

```
Jamess-MacBook-Pro:Homework_3 hippler$ python make_change_NATimed.py
Amounts: 100
Denominations: 100
Time to Complete: 0.0113320350647 seconds
Amounts: 200
Denominations: 200
Time to Complete: 0.0509119033813 seconds
Amounts: 300
Denominations: 300
Time to Complete: 0.117516040802 seconds
Amounts: 400
Denominations: 400
Time to Complete: 0.239617109299 seconds
Amounts: 500
Denominations: 500
Time to Complete: 0.393487930298 seconds
Amounts: 600
Denominations: 600
Time to Complete: 0.628585100174 seconds
Amounts: 700
Denominations: 700
Time to Complete: 0.964836120605 seconds
Amounts: 800
Denominations: 800
Time to Complete: 1.41175317764 seconds
Amounts: 900
Denominations: 900
Time to Complete: 1.89316916466 seconds
Amounts: 1000
Denominations: 1000
Time to Complete: 2.64952993393 seconds
Jamess-MacBook-Pro:Homework_3 hippler$
```

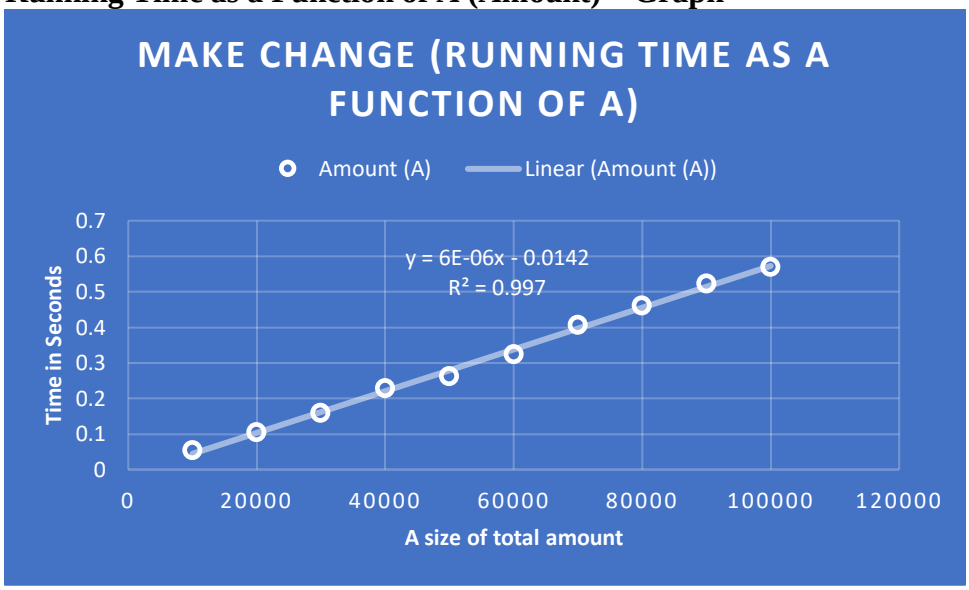
b) On three separate graphs plot the running time as a function of A, running time as a function of n and running time as a function of nA. Fit trend lines to the data. How do these results compare to your theoretical running time? (Note: n is the number of denominations in the denomination set and A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

ANSWER:

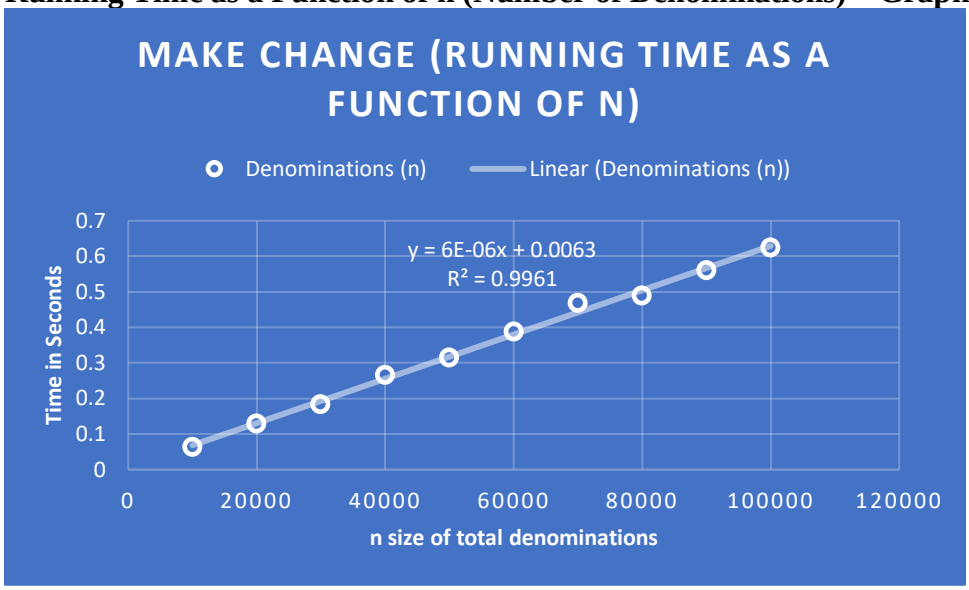
As we can see from the results depicted in the graphs below, when analyzing either the denomination or the total amount growing rapidly individually, the trend lines appear almost linear. In fact, I fitted both graphs to a linear trend line and was given a regression of over 0.99 on both accounts meaning the data was almost a perfect fit. While the theoretical running time of the make change algorithm is $\theta(VA)$, I believe we are seeing more linear results, because the growth of either the length of denominations (V) or the total amount (A) are growing so much faster individually that the impact of the other static element is trivial and is not apparent in the run time results.

On the other hand, when having both the length of the denominations and the total run time rapidly grow in tandem, we can see that the trend line becomes more polynomial and is more in line with the theoretical running time of $\theta(VA)$ generally associated with the making change dynamic programming algorithm. This result is different than those of the first two graphs because the growth of both denominations and amounts have noticeable impact on the running time of the program.

Running Time as a Function of A (Amount) – Graph



Running Time as a Function of n (Number of Denominations) – Graph



Running Time as a function of nA (Denominations • Amounts) – Graph