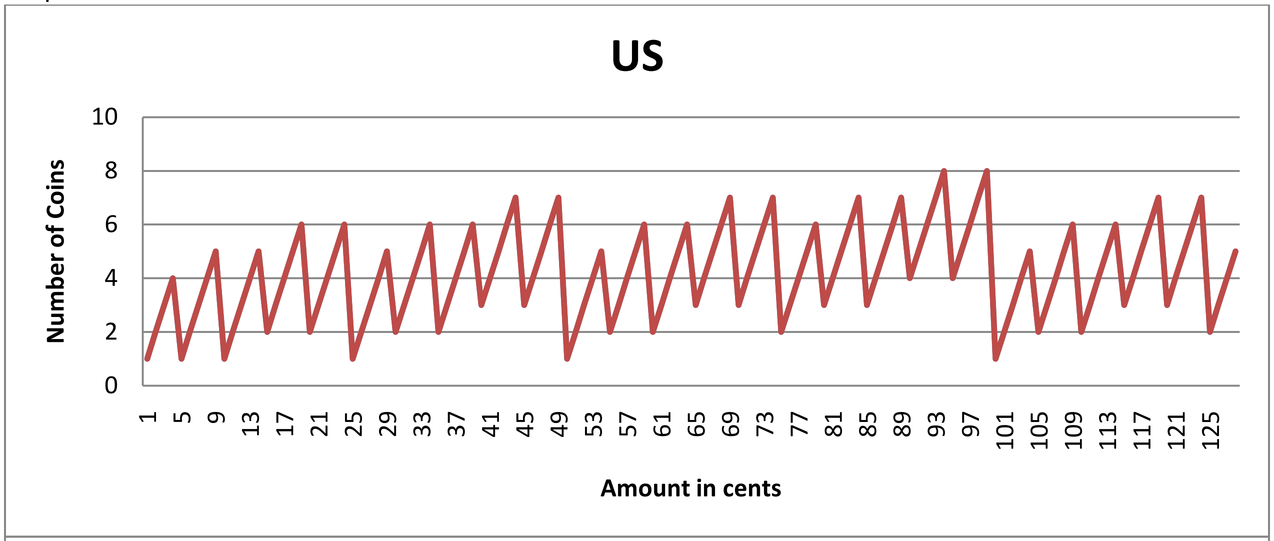
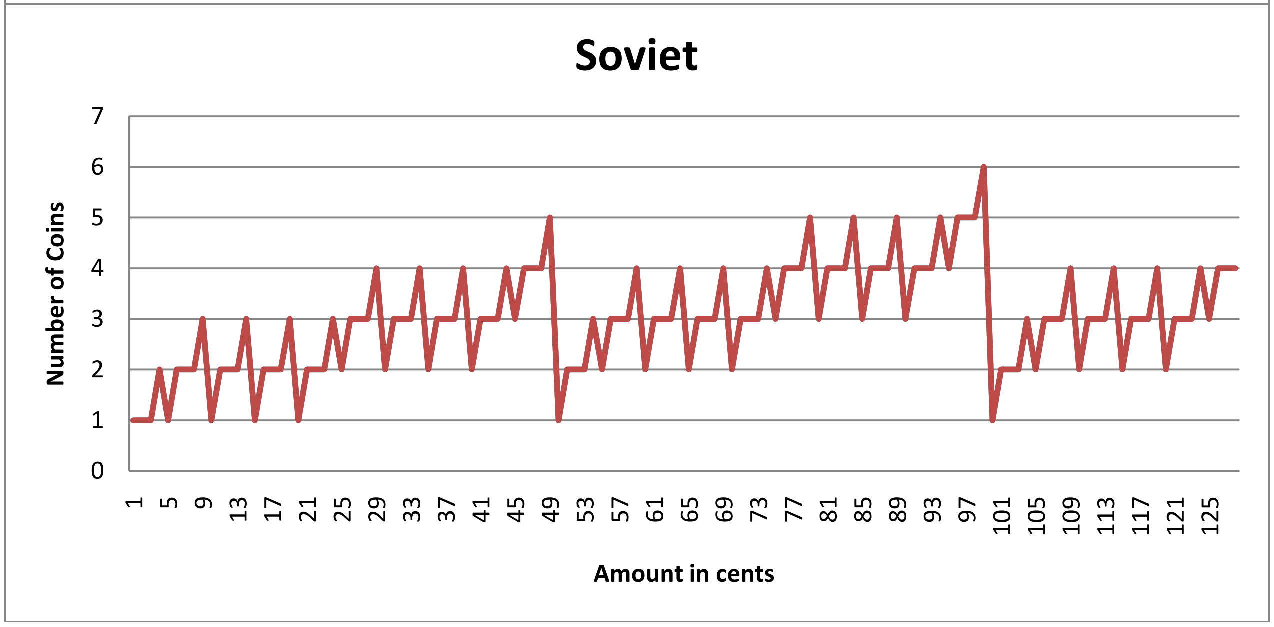
Lab 4: Tyler Holland (tyhollan)

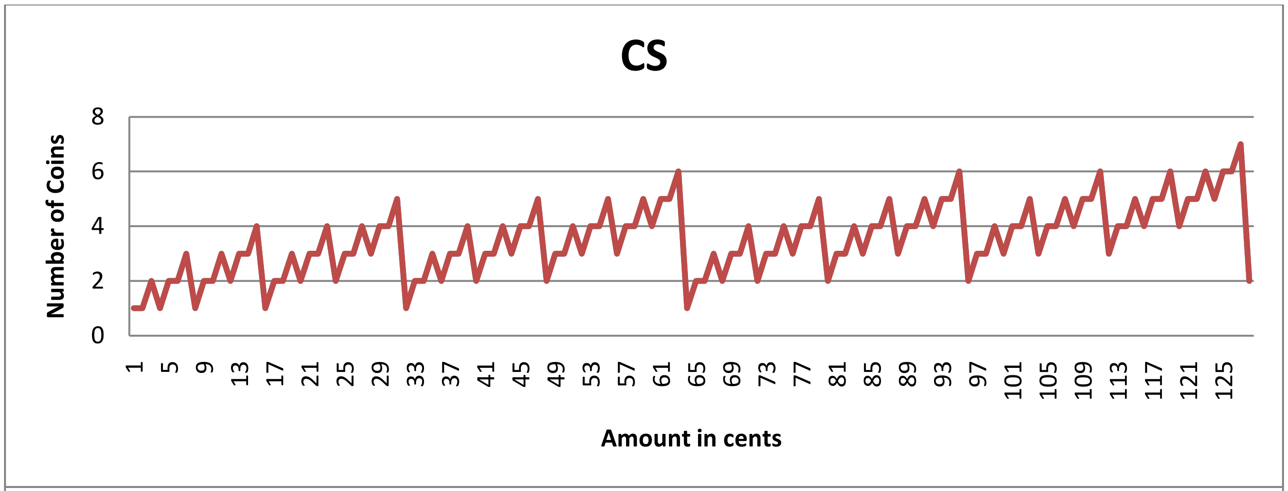
Question 1 and 2: In Making Change, is a greedy algorithm optimal?
Answer: Yes and no. It depends on the type of currency involved. For US, Soviet, and CS currencies, greedy algorithms always returned an optimal result, same with the dynamic programming method. However, for US No Nickel, and Crazy currencies, the greedy algorithm would only sometimes return an optimal result, but the dynamic programming algorithm always gave an optimal solution.

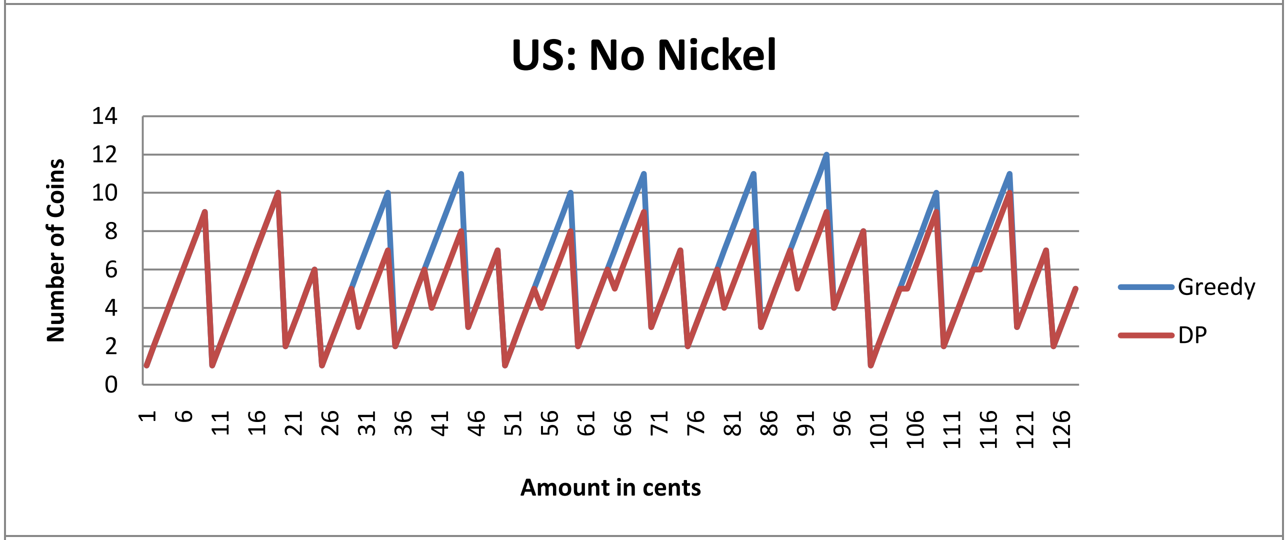
For clarification, an optimal result is only based on the number of coins needed to make a certain amount of change. When a greedy algorithm and a dynamic programming algorithm return with the same number of coins, the denominations can still differ.

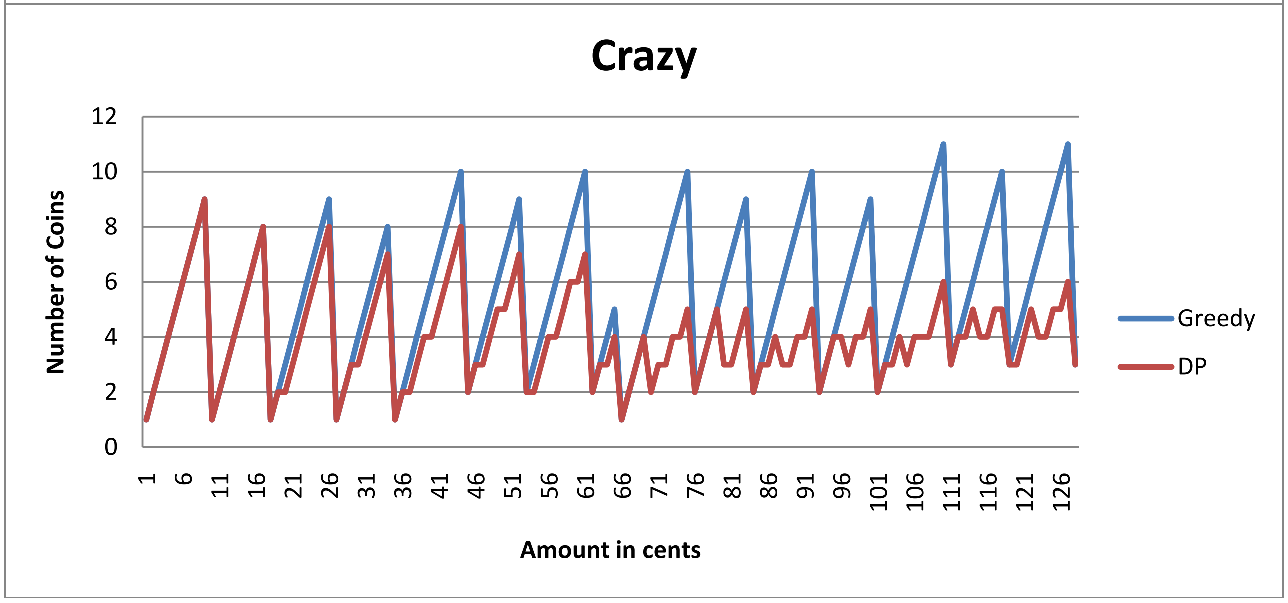
Graphs:






CS
Number of Coins
0
2
4
6
8
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117 121 125
Amount in cents


US: No Nickel
Number of Coins
0
2
4
6
8
10
12
14
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116 121 126
Amount in cents
Greedy
DP

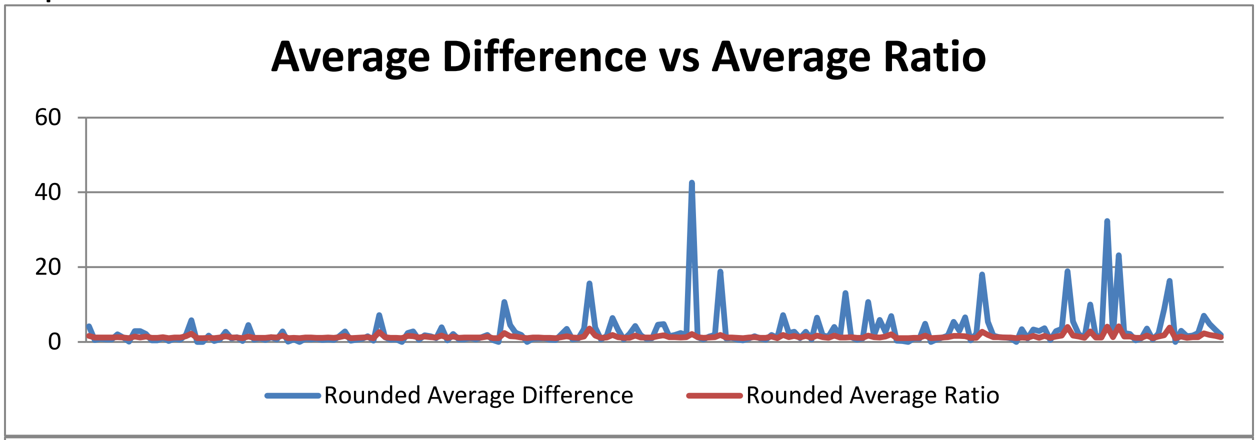

Crazy
Number of Coins
0
2
4
6
8
10
12
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116 121 126
Amount in cents
Greedy
DP

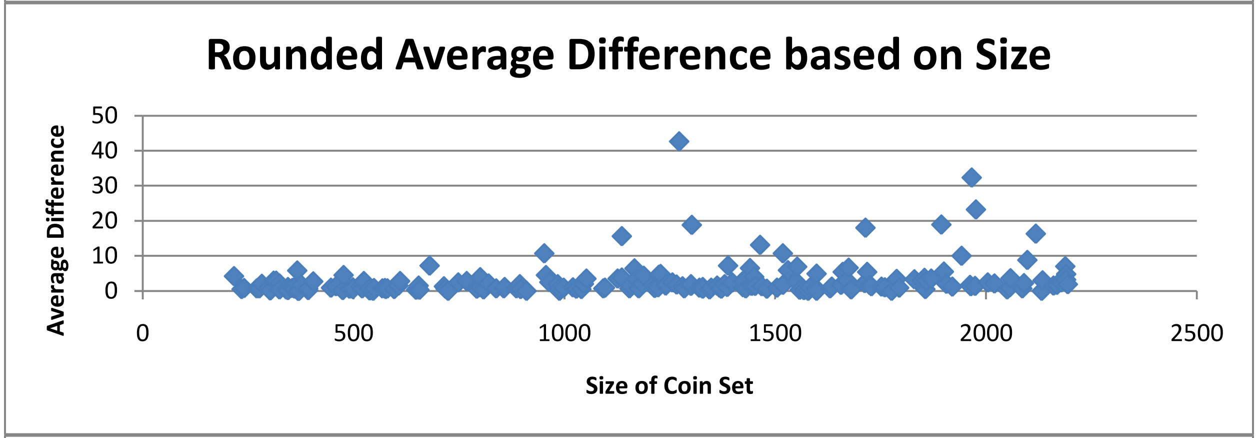
Question 3: How likely is a greedy algorithm to provide an optimal solution for an instance of the Making Change problem?
Answer: In my study of 200 randomly generated coin sets, 10 returned completely optimal results. That gives the greedy algorithm a 5% chance to return an optimal solution given a random coin set.

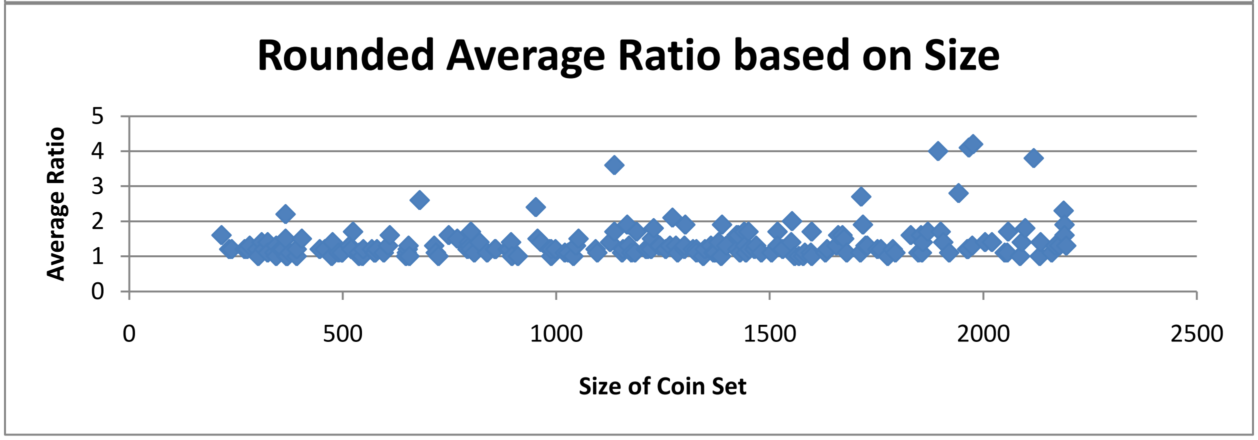
There was no trend in which coin sets gave optimal results based on the number of coins. There are both small (under 300) and large (over 2000) sets that gave optimal results, while sets of nearly equal sizes also gave not optimal results in different runs. This shows that the greedy algorithm only returns optimal results when the currency has somewhat specific denominations, such as US or Soviet coinage.

Question 4: What is the difference between the optimal solutions (computer by the dynamic programming algorithm) and the greedy solutions to the making change instances?
Answer: In terms of the average difference between the greedy and dynamic programming solutions for Make Change, the greedy algorithms tended to give answers that had on average 3 or more additional coins when compared to the DP algorithm. The highest difference was an average of 43 additional coins, and the lowest were less than one additional coin.
In terms of the ratio between greedy and DP results, the average is that the greedy algorithm returns with 1.25 times the number of coins that the DP algorithm does. The highest ratio is a little above 4 times, and the lowest ratio is very close to 1 times.

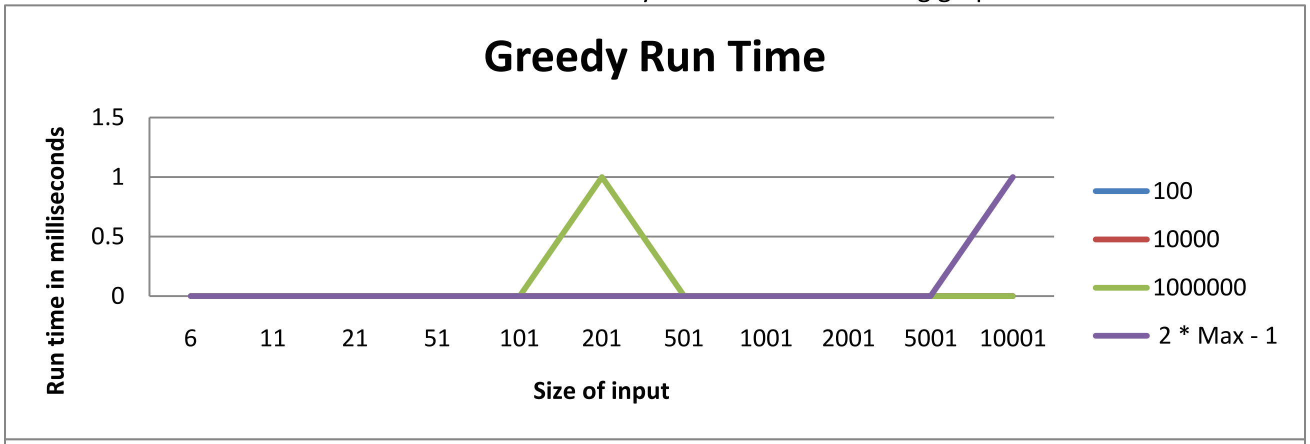
Graphs:

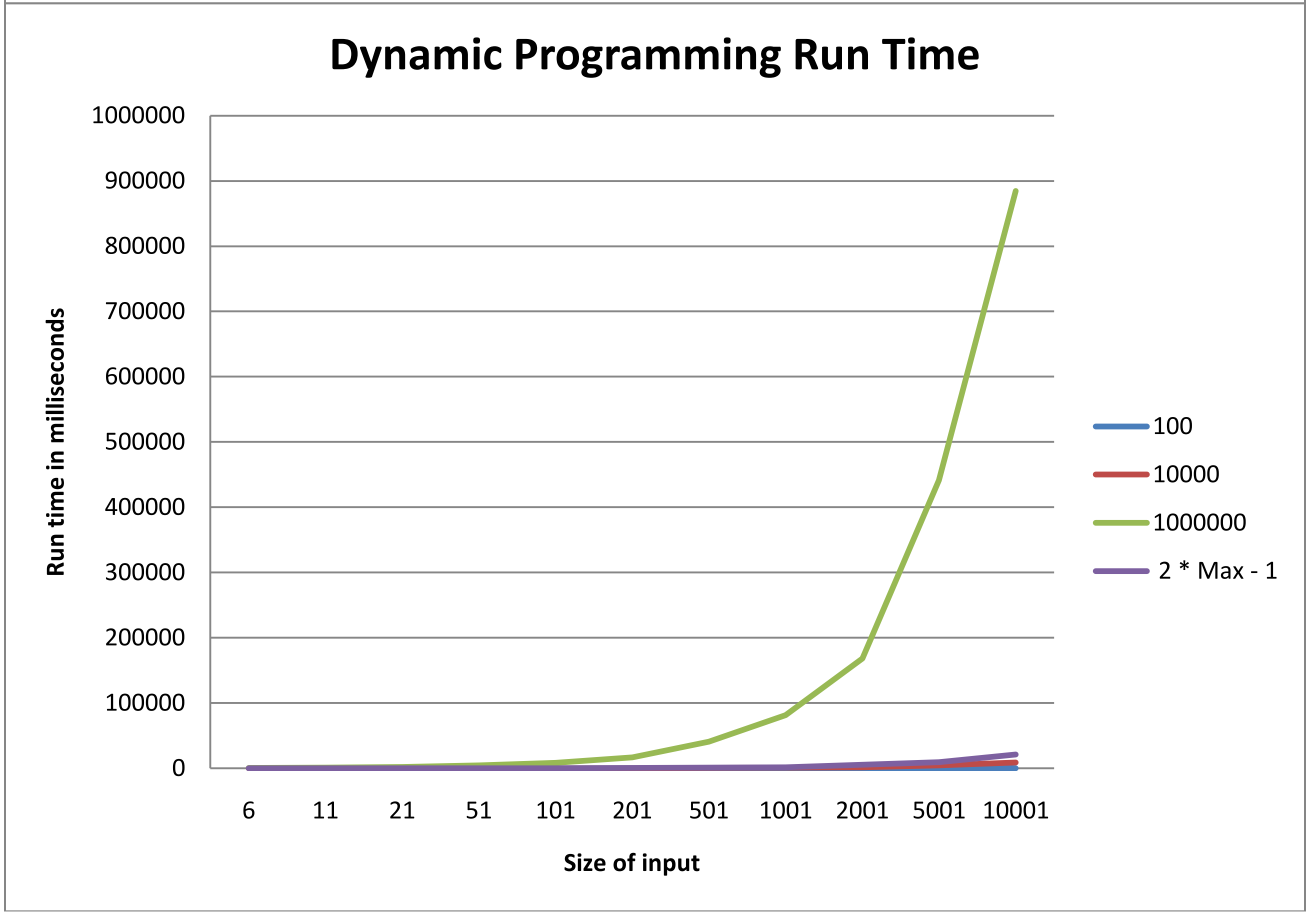

Average Difference vs Average Ratio
60
40
20
0
Rounded Average Difference
Rounded Average Ratio


Rounded Average Difference based on Size
Average Difference
50
40
30
20
10
0
0
500
1000
1500
2000
2500
Size of Coin Set


Rounded Average Ratio based on Size
Average Ratio
5
4
3
2
1
0
0
500
1000
1500
2000
2500
Size of Coin Set

Question 5: What is the difference between the running time behavior of the dynamic programming algorithm and the greedy algorithm?
Answer: The dynamic programming algorithm takes **much** longer to complete than the greedy algorithm does when large coin values are used. After a coin set of size 1000, the greedy algorithm stays at a very fast speed of less than a full millisecond, while the dynamic programming one takes above 3 milliseconds to run. The results can be most accurately shown in the following graphs:



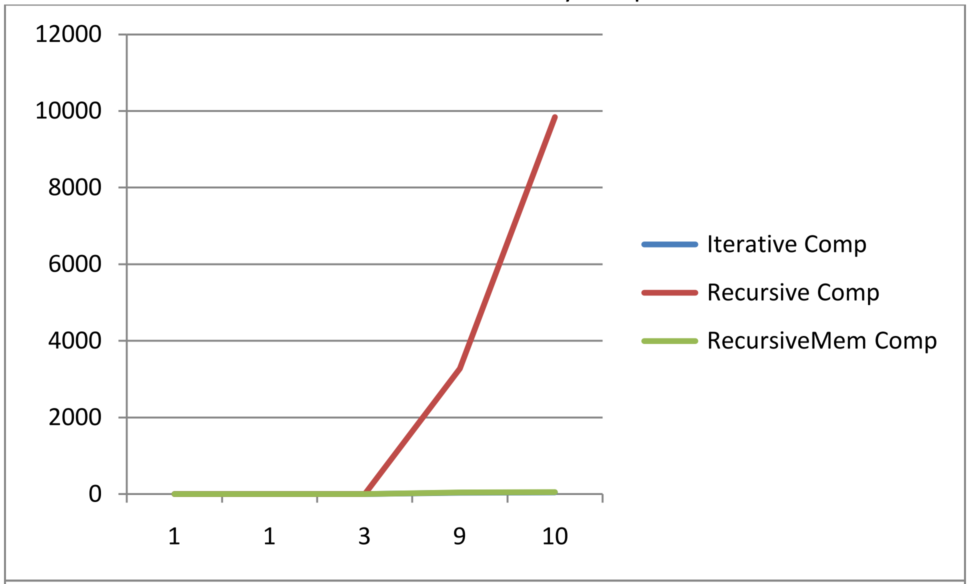


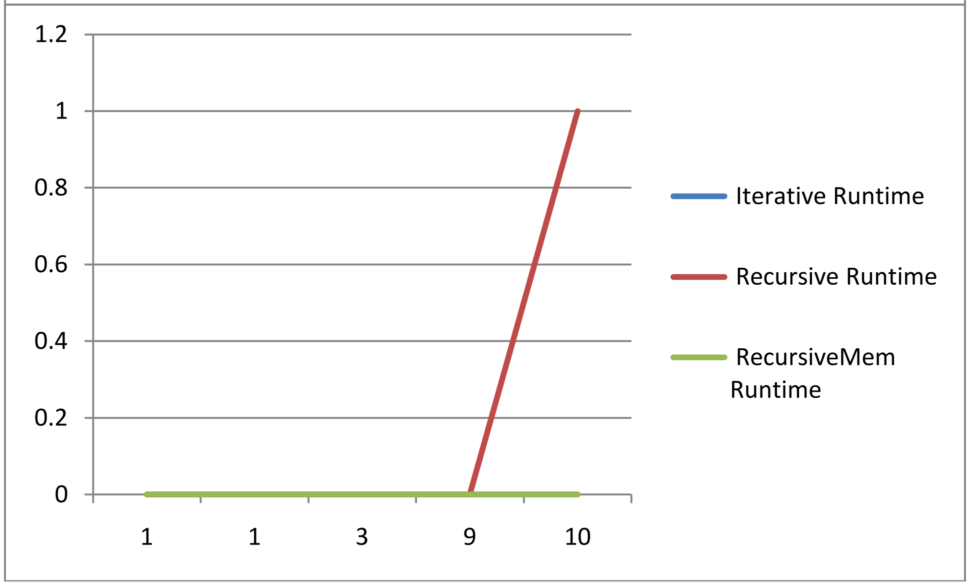
<u>Rod Cutting</u>

Analysis of results:

After running the test cases, I was able to see that the Iterative function needed to use the least amount of comparisons, where CutRecursive had to use the most. In terms of running time, CutRecursiveMem was the fastest, and CutRecursive was the slowest.

This happened because with large data sets, memorizing the past answers greatly increases the speed of the algorithm, and also lowers the number of comparisons. The iterative approach needed to go back multiple times to find each solution, and the recursive approach was in a middle ground between the two. Recursion has the benefit of returning values back to previous functions, so it does not need as many comparisons.


Iterative Comp
Recursive Comp
RecursiveMem Comp


Iterative Runtime
Recursive Runtime
RecursiveMem Runtime