Group Assignment Two Project Report

Faaiq Waqar

Q: Describe, in words, how you fill in the dynamic programming table in *change_dp*:

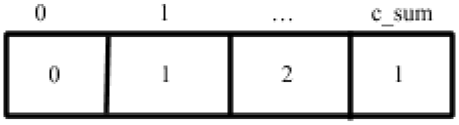
First, we can begin by defining and understanding the dynamic programming table itself. We are Using a dynamic programing table of size n, where n is equal to indices of 0 through the objective sum that we want to attain. For each of these, we will use the table index to represent the ideal coin use for that specific index, so that we may use these in a recursive definition for our final dynamic programing table. Here is an example visual of what the table might look like:

0	1	…	c_sum

Now, for the purposes of reference and continuity in the filling out of the table, we will begin by adjusting initial values of the table, that is, the value in 0 will be 0 as an empty set will always yield the proper amount of coins needed to obtain that result. For the other values in the table, we will fill with a max value (very large) so we can use for reference in comparison later. In the diagram we can just imagine these as infinity.

0	1	…	c_sum
0	inf	inf	inf

From here we can begin applying the following fill out format for each block. We will step over one, and for each coin consider the following: with each coin in the index, is the current index minus the coin value plus one, indicating one more cent, smaller than the value already in our dp spot? If so, we have an optimal solution! We should do this every time for every block in order to ensure we get the best possible answer, considering all coin values, we will iterate through all coins per table index, and in the end, will have the final value in the table be the optimal solution to the problem. Here is an example of what this might look like:



0	1	…	c_sum
0	1	2	1

Q: Give pseudocode for each algorithm, *change_dp* and *change_greedy*:

Change Dynamic Programming Approach:

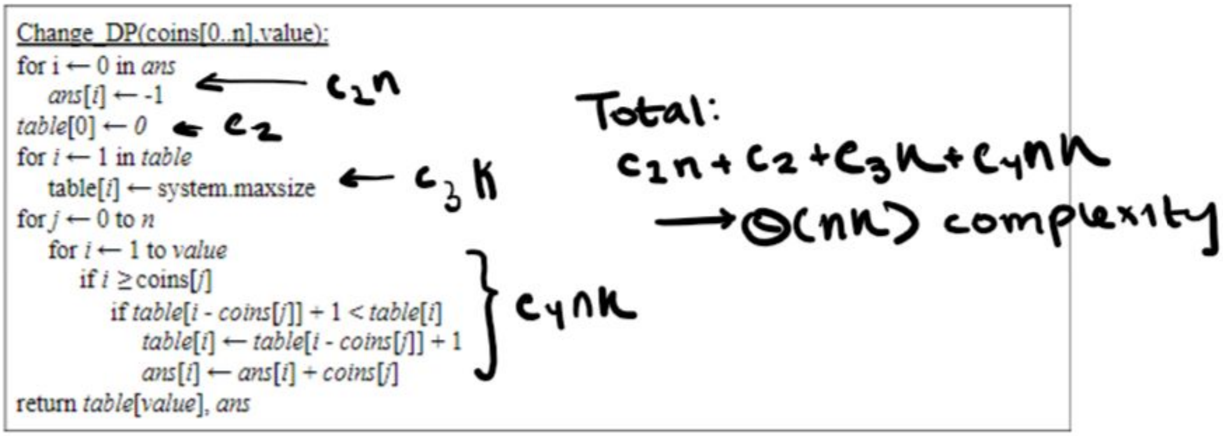
```
Change_DP(coins[0..n],value):
for i ← 0 in ans
    ans[i] ← -1
table[0] ← 0
for i ← 1 in table
    table[i] ← system.maxsize
for j ← 0 to n
    for i ← 1 to value
        if i ≥coins[j]
            if table[i - coins[j]] + 1 < table[i]
                table[i] ← table[i - coins[j]] + 1
                ans[i] ← ans[i] + coins[j]
return table[value], ans
```

Change Greedy Algorithm Approach:

```
Change_Greedy(coins[0..n],value):
for i ← 0 to i
    coin_counts[i]  = 0
value_remaining ← value
index ← n - 1
while index ≥ 0
    while value_remaining ≥ coins[index]
        value_remaining ← value_remaining - coins[index]
        coin_counts[index] ← coin_counts[index] + 1
    index ← index - 1
return coin_counts
```

Q: Give the theoretical run-time analysis for both the approaches:

Dynamic Programing Approach:



```
Change_DP(coins[0..n].value):
for i ← 0 in ans
    ans[i] ← -1
table[0] ← 0
for i ← 1 in table
    table[i] ← system.maxsize
for j ← 0 to n
    for i ← 1 to value
        if i ≥ coins[j]
            if table[i - coins[j]] + 1 < table[i]
                table[i] ← table[i - coins[j]] + 1
                ans[i] ← ans[i] + coins[j]
return table[value], ans
```

$\leftarrow c_2 n$

$\leftarrow c_2$

$\leftarrow c_3 k$

$\} c_4 n k$

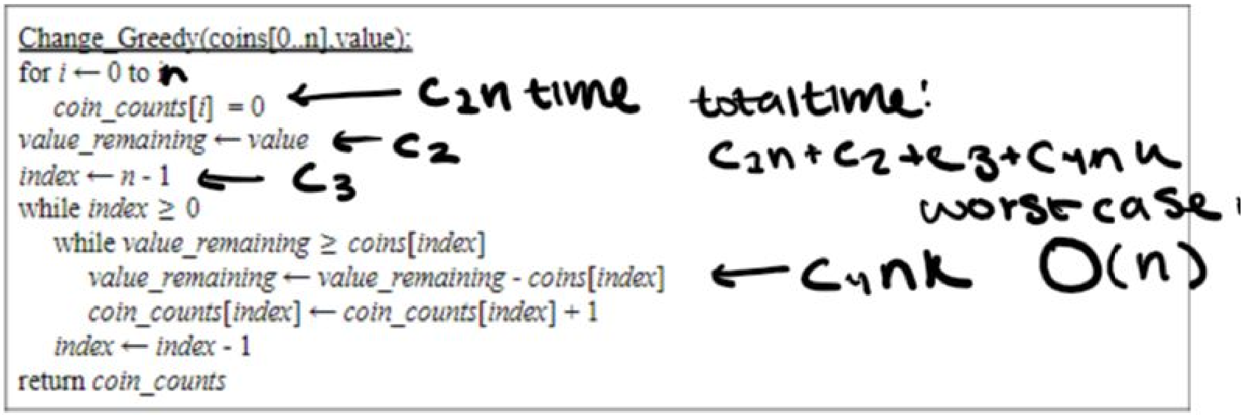
Total:

$c_1 n + c_2 + c_3 k + c_4 n k$

$\longrightarrow \Theta(nk)$ complexity

Here I have labeled the computational time of each of our pseudo code segments, giving us an understanding of the constants and the computational time of the loop through the elements of the graph, we can see that we end up with computations tied to $\Theta(nk)$ time, where n represents the number of coins, and k represents the integer sum we hope to get

Greedy Algorithm Approach:



```
Change_Greedy(coins[0..n].value):
for i ← 0 to n
    coin_counts[i] = 0
value_remaining ← value
index ← n - 1
while index ≥ 0
    while value_remaining ≥ coins[index]
        value_remaining ← value_remaining - coins[index]
        coin_counts[index] ← coin_counts[index] + 1
    index ← index - 1
return coin_counts
```

$\longleftarrow c_1 n$ time

$\leftarrow c_2$

$\leftarrow c_3$

$\longleftarrow c_4 n k$

totaltime:

$c_1 n + c_2 + c_3 + c_4 n k$

worst case

$O(n)$

Once again, I have labeled the complexity of the algorithm through understanding the single computations, and understanding the worst case analysis, to find that the solution runs in $\Theta(n)$ time, where n represents the number of coins. Notice that the value to be obtained does not have the same effect on this algorithm

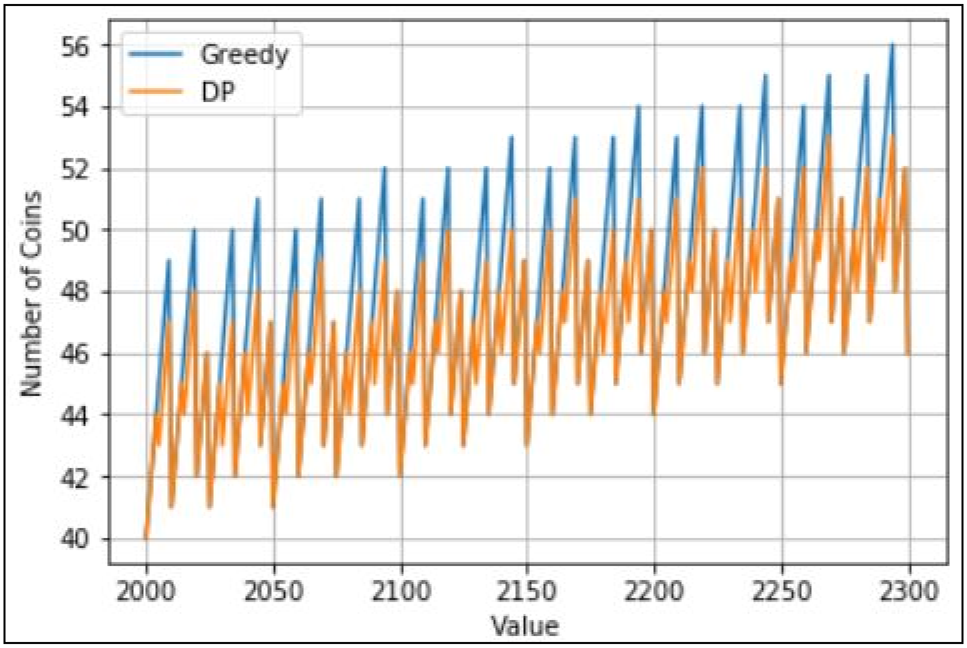
Q: Prove the dynamic programming approach by the use of induction. Use the following to prove that … $T[i] = \min_{j:coins[j] \leq i} \{T[i - coins[j]] + 1\}, T[0] = 0$

Theorem: The dynamic programing table at an index will return the minimum amount of coins

Base Case: At index 0 of the dynamic programming table, the table value will be 0

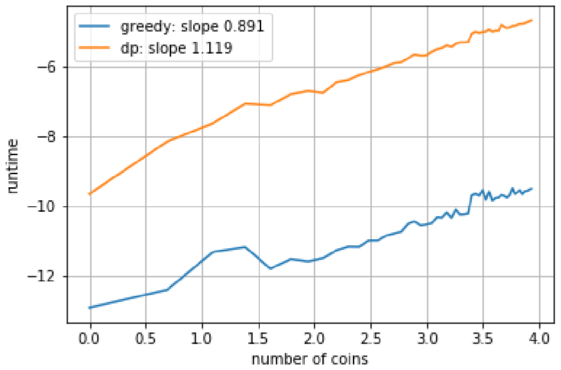
Proof: If i = 0, there is only one set of coins that we can use to obtain that, and that would be an empty set of coins, meaning 0 coins. If i >= 1, let max be the least amount of coins used for the value at T[i] for the coin set [0..n]. Since max is relative to the best solution, and an index at previous index [j] in coins gives us the previous chance, we know that this will give us the minimum solution.

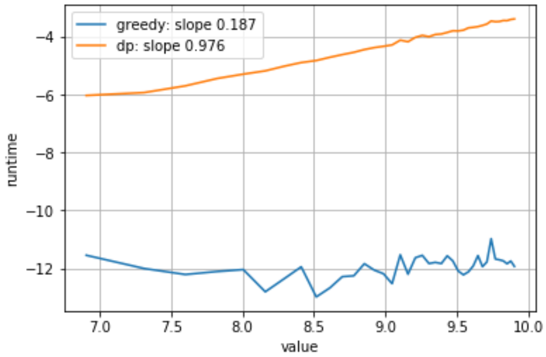
5.



The biggest noticeable thing about the plot that I was not expecting was the repeated spikes, not just in the use of coins, but more so in the jump of alignment where the dynamic programming algorithm and the greedy algorithm meet for intervals, and then spike away in size. This does make me wonder if there is an even better mathematical approach to this algorithm that has us using less computational time than the one we arrived that for the dynamic programming approach (O(nk)). Perhaps understanding the overall sum as a modulo of something else could help us get a better algorithm.

Another thing I noticed that I mentioned before was the spikes from high to low coin counts. This makes sense because of correlation with coin sizes, but upon seeing it immediately I was a little surprised. Now, in comparison, I have to say the greedy algorithm, while not always exactly right, gets pretty close, and for someone with less time to run the computation (like in their head as a cashier), it will certainly reach a close minimization that someone could get if not given an automatic coin dispenser or program to tell them the minimal amount of coins.





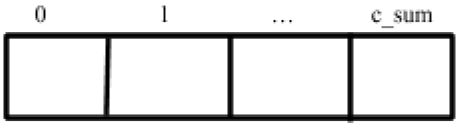
6.

First, let's do an analysis of our runtime in comparison to value graph. We can see the slope analysis of the graph, where the dp slope has a constant increase in runtime as the value is incremented. we have a slope estimate of 0.976, which would correlate to an increase in a value of 1 in comparison to slope runtime, and according to our runtime analysis done in problem 4, this should make a lot of sense. Both the number of coin input and the size of the value are equally important pieces to the algorithm. Our dynamic programming table also carries the space complexity of the size of the value, and thus it makes sense for such a strong correlation to exist. Now, if we look at the greedy algorithm graph in comparison to the graph, there is no real strong correlation, and although we do have a slope given of 0.187, but given the actual unusual plotting points of the data on the graph, we can say with a good amount of certainty that this is due to a very weak correlation coefficient, and this makes sense as the runtime of the greedy algorithm according to our analysis is not very dependent on this, and so its likely the runtime spikness is due to the runtime of coin computations, and not as much the value itself..

Now, when it comes to the number of coins, we can see both of the algorithms have stronger correlation to the runtime as this increases, and this should come at no surprise. Both of the algorithms rely on iterating one way or another through the possible coin selections to see if the amount of coins is a good fit, and to no surprise this shows up, both in our runtime analysis and in our graphical visualization, and this should come as no surprise. Now, the changes in increase may be because the dynamic programming algorithm will always run through a computation for each coin, where the greedy algorithm may skip some computations for some coins. Now, finally, the biggest thing we can get out of our 2 analysis is the fact that the dynamic programming algorithm has longer runtime than that of the greedy algorithm, which we kind of assumed.

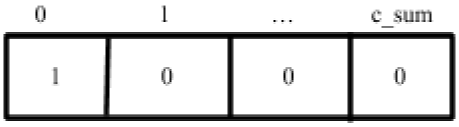
7.

If we were to compute the number of possible ways we can make change, I might decide to make a dynamic programing table of the same size, and in the same light as this algorithm, rather than store amount of coins, use each iteration to calculate if there is a solution, pull the number from the previous index plus one. Let's begin by thinking of this in the context of the dynamic programming table, here is what our table should look like:



0	1	…	c_sum

Now, let's fill the initial index to 1, and this is because we know that an empty set is a solution on its own, but we will have no other way to have a zero solution set, we will use this as a reference for the solutions further on. here is our table before further manipulation



0	1	…	c_sum
1	0	0	0

Now let's fill out the table, at every index we will consider the following problem: For each coin we have, does the solution at the current index minus the coin value have something greater than 0, if so take that value, add one, and add it to the current place in the table, do this for all of them and get the solution set at the final index