Project 2B Report

Psuedocode and theoretical runtimes for the Greedy Algorithm, Recursive Solution, and the Dynamic Programming Soltuion of the Coin Counting Problem:

int GreedyCountCoins(vector<int> coins, int val) {

1. int count = 0

2. int i = coins.size() - 1

3. while (i >= 0 && coins[i] >= val) {

4. --i

}

5. if (i >= 0) {

6. ++answers[i]

7. count = 1 + GreedyCountCoins(coins, val - coins[i])

}

8. return count

}

Line Cost Times

|  |  |  |
| --- | --- | --- |
| 1. | 1 | 1 |
| 2. | 1 | 1 |
| 3. | 1 | Coins.size() |
| 4. | 1 | Coins.size() |
| 5. | 1 | 1 |
| 6. | 1 | 1 |
| 7. | T(val-coins[i]) | 1 |
| 8. | 1 | 1 |

The theoretical performance of the greedy algorithm solution is Θ(n). While similar algorithms such as the CUT-ROD problem for the greedy algorithm is Θ(nlog(n)), there is no sorting needed in this specific scenario unlike the CUT-ROD problem, cutting the runtime to Θ(n).

int RecursiveCountCoins(vector<int> coins, int val) {

1. if (val == 0) {

2. return 0

}

3. int q = 9999999

4. for (int i = 0; i < coins.size(); ++i) {

5. if (val >= coins[i]) {

6. q = min(q, 1 + RecursiveCountCoins(coins, val - coins[i]))

}

}

7. return q

}

Line Cost Times

|  |  |  |
| --- | --- | --- |
| 1. | 1 | 1 |
| 2. | 1 | 1 |
| 3. | 1 | 1 |
| 4. | 1 | Coins.size() |
| 5. | 1 | Coins.size() |
| 6. | T(val-coins[i]) | Coins.size() |
| 7. | 1 | 1 |

The theoretical performance of the recursive solution is Θ(2n).

vector<vector<int>> DynamicCountCoins(vector<int> coins, int val) {

1.    vector<int> results(val + 1, 999999999);

2.    results[0] = 0;

3.    vector<vector<int>> answers(val + 1, vector<int>(coins.size() + 1, 0));

4.    for (int i = 1; i <= coins.size(); ++i) {

5.        for (int j = coins[i - 1]; j <= val; ++j) {

6.            if (1 + results[j - coins[i - 1]] < results[j]) {

7.                results[j] = 1 + results[j - coins[i - 1]];

8.                answers[j] = answers[j - coins[i - 1]];

9.                ++answers[j][i - 1];

            }

        }

    }

10.    return answers;

}

Line Cost Times

|  |  |  |
| --- | --- | --- |
| 1. | C1 | 1 |
| 2. | C2 | 1 |
| 3. | C3 | 1 |
| 4. | C4 | Coins.size() |
| 5. | C5 | Coins.size() \* (val / 2) |
| 6. | C6 | Coins.size() \* (val / 2) |
| 7. | C7 | Coins.size() \* (val / 2) |
| 8. | C8 | Coins.size() \* (val / 2) |
| 9. | C9 | Coins.size() \* (val / 2) |
| 10. | C10 | 1 |

The theoretical solution of the dynamic solution is Θ(n\*k), where n is the number of coins and k is the value of the target change.

Recursive Solution Runtimes:

A screenshot of a computer

Description automatically generated

After this, it would not make sense to keep running the program due to how long it took do the computation for 73 cents, it would take about 1000 times longer to compute 83 cents in comparison to the 2 and a half hours it took to compute the 73-cent example.

Greedy Algorithm Solution Runtimes:

A screenshot of a computer

Description automatically generated

This one was able to run to the end very quickly. Instead of the runtime exploding, it has a more linear approach runtime due to the simplicity of the program. The runtime has much more to do about the complexity of the solution for the input, so even though values like 50 are greater than 49, 49 will take much longer due to the complexity of the solution.

Dynamic Solution Runtimes:

A screenshot of a computer

Description automatically generated

For the dynamic programming solution, it was slower than the greedy solution, but still much faster than the recursive solution. Since the theoretical time complexity is slower than the greedy algorithm but faster than the recursive solution, this lines up with the theoretical results.

Graphs to compare recursive, greedy, and dynamic execution time solutions:

Comparison of Greedy vs Dynamic:

A blue screen with white text

Description automatically generated

For the weird coin system, we saw similar results. The greedy algorithm went much faster than the recursive algorithm, however, the greedy algorithm provides a not correct solution for the weird coin system, while the recursive algorithm finds the correct solution. This is because the greedy algorithm finds that the best way to make 69 in change is to subtract 25 first, which then eliminates the option to use 23 three times to create 69. This scenario can happen whenever you have a system where there is a denomination of value that is less than 2x the value of the preceding value. For example, since the weird system has 23 and 25 in their denominations, there will be a point, such as the 69 example, where the greedy algorithm will fail. Even with a system such as 1, 5, 10, 21, 40, trying to build 42 will cause an inaccurate result, as the greedy algorithm will say it takes 3 coins, while the recursive will correctly realize it only takes 2. So, although the greedy algorithm was much faster, it did not give the best result.

Dynamic on the other hand, finds the correct answer even on the weird coin system, is faster than the recursive solution for large samples, but is slower than the greedy solution. For a general case that needs to cover every coin system, this may be the best option for a compromise for faster solution time but also correctness. For example, since It can store the result of 46, in the weird change problem, it is able try 23 with 46 to get the correct answer while not having to do all of the calculation of the recursive calls, since the calculation is already done.

For my execution times, I believe the program has shown that it is consistent with the theoretical runtimes I assigned earlier. With the purely recursive one not being feasible to execute past a coin count of 73, it is incredibly slow when the numbers get big. Alongside that, the greedy algorithm did a great job of being incredibly quick. The dynamic program was as expected, right in the middle in terms of performance. There were no surprises in the runtime of this part of the project.

So, in conclusion, the greedy algorithm did a great job with the U.S. system, as it was able to provide the best denomination for all values and was incredibly quick. The recursive results show, however, that it is important to understand that there may be issues with such an algorithm if there are certain denominations that are present. This is where the dynamic solution is important, as it blends the two algorithm’s strengths, giving a slight compromise for the speed of the greedy algorithm but getting much closer to it than the recursive algorithm’s runtime, while still keeping the accuracy of the recursive algorithm.