S5155126
Caleb Howard
Trimester 1 2020

Coin Change Problem Using Primes Solver C++

Overview

The coin change problem is a well known problem that addresses the question of finding the total number of combinations in a set of coins that will amount to a given target where each coin can be used multiple times.
The variation of the problem that this program solves will use all primes numbers from 1 to the target (including target) for the coins and will introduce an optional restraint of length on all combinations that can be included in the final result amount of combinations.
This document will include an explanation of the solver, description of how it works, results and a performance analysis.

Solution

The solver uses a bottom up dynamic programming implementation to calculate a final result.
A 3D solution matrix is used to store the subproblems that eventually build up to the final solution. The size of the matrix is shown by table[coins+1][target+1][subproblem_target+1].
The matrix structure is as follows:

- Each column represents each subproblem's target amount which is represented by the column index [0..target].
- Each row represents a set of coins that can be used to calculate the subproblem for each sub amount, the first row represents an empty set with no coins, each successor row introduces a new coin from the coin set. (Note that the rows do not actually store an array of coins at each index, they simply store the newly added coin).
- Each subproblem is represented in the 3rd axis of the matrix. A subproblem is a vector where the indexes represent the length of a combination, and the value represents how many combinations there are for that index (length). For example, [0,2,1] translates to 0 combinations of length 0, 2 combinations of length 1 and 1 combination of length 2.

Figure 1 - Solution Matrix for coins = [1,2,3,4,5] and target amount = 5

	0	1	2	3	4	5
[ ]	[1]	[0,0]	[0,0,0]	[0,0,0,0]	[0,0,0,0,0]	[0,0,0,0,0,0]
[**1**]	[1]	[0,1]	[0,0,1]	[0,0,0,1]	[0,0,0,0,1]	[0,0,0,0,0,1]
[1,**2**]	[1]	[0,1]	[0,1,1]	[0,0,1,1]	[0,0,1,1,1]	[0,0,0,1,1,1]
[1,2,**3**]	[1]	[0,1]	[0,1,1]	[0,1,1,1]	[0,0,2,1,1]	[0,0,1,2,1,1]
[1,2,3,**4**]	[1]	[0,1]	[0,1,1]	[0,1,1,1]	[0,1,2,1,1]	[0,0,2,2,1,1]
[1,2,3,4,**5**]	[1]	[0,1]	[0,1,1]	[0,1,1,1]	[0,1,2,1,1]	[0,1,2,2,1,1]

Figure 1 illustrates the base cases as:

- If target amount = 0 then there is 1 combination of length 0 to make change i.e. [1]
- If no coins are given, there are 0 ways to make change i.e. [0,0], [0,0,0], etc.

All other cases are represented as:
For every new coin added we have the option to include it in the subproblem solution or exclude it.

- If the current coin value is less than or equal to the current amount needed then find combinations that sum to current amount by including and excluding the coin
- If the coin value is greater than the current amount then we cannot use it and therefore only find the combinations that exclude it.

The operations to include or exclude a coins are as follows:

- To include the coin we minus the coin from the amount and use that sub problem solution (table[row][sub amount - current coin]).
- To exclude the coin we take a solution for the same amount but not considering that coin (table[row-1][col]).

To actually calculate a sub problem solution we take the sub problem solution vector for the current amount that excludes the current coin which we will call vector1 and, if included, the sub problem solution vector that includes the coin which we will call vector2 and add the values in the vectors together to produce our new solution vector. However, because vector2 introduces a new coin we must offset the additions by +1 to account for this addition that vector1 does not have.

For example, referring to *figure 1*, if table[1][3] was the current subproblem than,
vector1 = table[0][3] = [0,0,0,0]
vector2 = table[1][amount - coin] = table[1][3 - 1] = table[1][2] = [0,0,1]
Solution = [0,0,0,0] +
[0,0,1] (offset of +1)
table[1][3] = [0,0,0,1] is our solution for that subproblem

The get the final number of combinations we can iterate over the final calculated solution vector (table[coins.size][target]) from the index of the minimum number of combinations to the index of maximum number of combinations (as the indexes are the lengths) and sum each amount of combinations in that range to get our final result.

S5155126
Caleb Howard
Trimester 1 2020

Pseudocode

```
BEGIN
    coins = seive_of_eratosthenes(target)
    // First sub problem has 1 combination of length 0
    table[0][0] = 1

    // Base case for empty set is 0 combinations for all amounts (excluding 0)
    For i ⇦ 1 to i < target + 1 do:
        for  j ⇦ 0 to j < j + 1 do:
            table[ 0 ][ i ][ j ] = 0

    For row ⇦ 1 to row < number of coins + 1 do:
        for col ⇦ 1 to col < target + 1 do:

            // Get the sub problem from the above row for the subproblem amount
            y = table[row-1][col]

            // Check if the current coin can be used
            // i.e. if the current coin is not greater than the current amount
            If (col - coins[row-1]) >= 0:

                // Get the sub problem on that row (amount - coin)
                x = table[row][col - coins[row-1]]

                // Add the x and y vectors together with the offset of +1
                // and that is our solution
                table[row][col] = sum_vectors_with_offset(y, x)
            Else:
                // Only use the above vector that does not consider current coin
                table[row][col] = y

    // The final solution is the last vector in table, iterator from minimum number of allowed
    // combinations to the maximum number and sum the total number of combinations for
    // that range
    solution_vector = table[coins.size][target]
    num_combinations = 0
    For i ⇦ min_combinations to i < max_combinations + 1 do:
        num_combitions = num_combinations + solution_vector[ i ]

    Return num_combinations
END
```

```
BEGIN sum_vectors_with_offset(vector1, vector2)
        // Add vector one to sum
        summed_vector = vector1

        // Start at index 1 (+1 offset) of summed vector, add each value from vector2
        For i ⇦ 1 to i < vector2.size + 1 do:
                summed_vector[ i ] = summed_vector[ i ] + vector2[ i ]

        Return summed_vector
END
```

Results

Input (amount min max)	Output	CPU Time (seconds)
5	6	3e-06
6 2 5	7	4e-06
6 1 6	9	2e-06
8 3	12	3e-06
8 2 5	10	3e-06
20 10 15	57	2.1e-05
100 5 10	14839	0.000466
100 8 25	278083	0.000281
300 12	16250414519	0.00691
300 10 15	32100326	0.00406

It can be inferred from the results table above that the algorithm was correct and efficient.

Performance Analysis

The performance analysis will be done on each function which will combine to give the final Time and Space complexity of the program

Let n = number of coins.
Let m = the target amount

sieve_of_eratosthenes()
Time Complexity: O(m (log m) (log log m))
Space Complexity: O(m)

sum_vectors()
Time complexity :
The basic operation here is addition with a constant time of O(1), let's assume the larger vector has a size of k, k <= m. For the worst case (k=m) we are performing 1 constant time for m runs, we therefore get the linear time complexity of O(m).

Space Complexity:
Each iteration over the vectors requires a constant amount of space, this space is freed after each iteration, therefore we get O(1) space complexity for the sum_vectors function.

solve_coin_change()
Time Complexity:
Initially the first row of the solution matrix is set to the base case. This means we push (1), (0,0), (0,0,0).... for m times, this can be written as the number of operations (pushes) as 1, 2, 3….m.
Add the operations together and we get 1 + 2 + 3 + … + m = m(m+1)/2, which in Big O notation is written as $O(m^2)$.

Secondly we run the dynamic solver which runs for each column, for each row minus 1 i.e. for each row [1..n] we run each column [0..m], this gives us a time complexity of
(n-1) * m * (operations in nested loop), to simplify we will use n*m*(operations in nested loop). The main operation in the nested loop is either summing the vectors which performs in O(m) time (solved in the above section) or copying a vector to a new vector which also has O(m) time. Therefore we now have $n*m^2$ time complexity.

The final operation consists of adding the combinations to get our final total number of combinations. If we consider min combinations to be 0 and max combinations to be the target amount as our worst case we get m operations or O(m) time complexity.

Putting the complexities of this function together we get $m^2 + n * m^2 + m$, simplifying this to take the most dominant term we get $O(n*m^2)$ for the solve_coin_change function.

Space Complexity:
For each each column in the solution matrix we are storing a subproblem solution vector that is the length of the sub target amount (index of the current column), the total memory for all columns in a single row can be written as 1 + 2 + 3 +...+ m+1, which equals (m+1)*((m+2))/2. To simplify the complexity we can use m(m+1)/2. We store n rows of each column, therefore we get n(m(m+1)/2), in Big O notation this translates to $O(n*m^2)$. The space complexity for the solve_coin_change function is $O(n*m^2)$.

Performance conclusion:
Now that we have calculated the complexity for each method we take the most dominant part that is proportional to the input.

Time Complexities:
For $O(n*m^2)$ + O(m (log m) (log log m)),
$O(n*m^2)$ is the dominant term, therefore,
Final Time Complexity: $O(n*m^2)$

Space Complexities:
For $O(n*m^2)$ + O(m), $O(n*m^2)$ is the dominant term, therefore,
Final Space Complexity: $O(n*m^2)$