2801ICT Computing Algorithms – Assignment 1 (Change making problems)

S5084207

Yoon Jin Park

Bachelor of Computer Science

Griffith University (Gold Coast Campus)

Introduction

The purpose of this report is to outline algorithms used for solving change making problems with constraints of the number of coins used and to analyse the results and performance of the algorithms. The change making problems provided in the assignment instruction requires counting the number of solutions to make up a target value with or without restricted number of coins. Two potential constraints include minimum number of coins (hereafter, Minimum coins) and maximum number of coins (hereafter, Maximum coins).

For instance, if an input has only one number 5 (a target value), then there is no restriction with the number of coins to be used. If there are two numbers, the first number is a target value while the second number is the number of coins to be used to obtain solutions. Where there are three numbers, the first number is still a target value and the second and third number represents Minimum and Maximum coins, respectively. In this case, the possible solutions need to be found for a range between Minimum coins and Maximum coins, inclusive. It is noted that the three numbers with the second number 1 and the third number equivalent to the first number, then this problem is same with one number problem without any coin restriction.

The next section provides two main algorithms used to solve the problems (the dynamic programming algorithm and the backtracking and pruning algorithm), followed by the last section describing the results and analysing performance to obtain the number of solutions.

Algorithm Design

Overview

For the purpose of solving the problems, the list of change making problems is categorised into two: the cases requiring the number of all possible solutions and the cases requiring the number of possible solutions with certain number of coins used. The first cases for all possible solutions are solved through the dynamic programming algorithm while the second cases for possible solutions with specific number of coins used are solved through the backtracking and pruning algorithm.

Algorithm Description

Checking prime numbers

To check whether a given number is a prime number, 1) if number is 2, then it is a prime number; 2) if number is divided by 2 and the remainder is 0, then it is not a prime number; and 3) loop from 3 to the square root of Value with step 2, check whether the number is divisible by

these. If it is divisible (the remainder is 0), then it is not a prime number. If the number passes all the three conditions, it is a prime number. The pseudo code is as follows:

```
Algorithm is_prime(integer number)
        //number is two, prime
        if number==2:
                return True
        //if number is divided by 2 and 0 remainder, it is not a prime
        if number%2==0:
                return False
        root = floor(squareroot(number))
         //as the multiplication of 2s are removed, we can check odd numbers only
        For n <- 3 to root with step 2
                if num%n==0:
                        return False
        //anything else, it is a prime number
        return True
```

Dynamic programming algorithm for all possible solutions

The initial algorithm to find all sets of the solutions uses dynamic programming. By constructing a table with all possible values and coin denominations, the number of solutions is counted by using the previous number of solutions saved in the table. The example below displays with Value (target value) 5, Minimum coin: 1, and Maximum coin: 6. The result is on the last column and row intersection and 1 needs to be added to this number to account for the gold coin.

Table 1. Dynamic programming table with no modification to the memory use

	0	1	2	3	4	5
1	1	1	1	1	1	1
2	1	2	2	3	3	3
3	1	1	2	3	4	5

The total number of solutions for Value 5, Minimum coin: 1, and Maximum coin: 6

This algorithm has Time complexity and Space complexity O(nm) where n is Value (more accurately Value +1) and m is the size of coin denominations (i.e. prime numbers smaller than the value and 1). In this case, the number of coin denominations, m, cannot be larger than the value, hence, the complexity is linear.

However, there is an opportunity to improve Space complexity with this algorithm (Time complexity does not change). Since the algorithm is constructed based on the data from the previous and current rows, only two rows are required technically. The table is then built by alternating the new results between two rows.

Table 2. Dynamic programming table with the reduced memory use (only two rows)

a. Constructing first two rows

	0	1	2	3	4	5
1	1	1	1	1	1	1
2	1	1	2	2	3	3

b. Constructing third row in the first row based on the result on the second row

	0	1	2	3	4	5
3	1	1	2	3	4	5
2	1	1	2	2	3	3

Despite its efficiency, the dynamic programming algorithm is not used for most cases because many problems are specified as identifying the number of solutions with certain number of coins. Since the dynamic programming algorithm tends to be used for an optimization problem, it is difficult to adjust the algorithm to find the number of solutions meeting the constraints (i.e. the

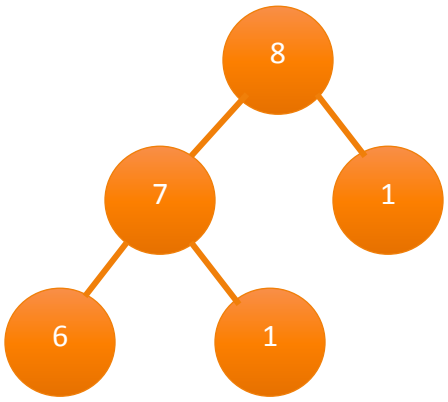
number of coins used). Therefore, the backtracking and pruning algorithm is developed to deal with these cases.

Backtracking and pruning algorithm for the number of solutions with certain number of coins

Between Backtracking and Branch and Bound algorithms, it is decided to adopt Backtracking and Pruning as Backtracking is an algorithm for identifying all possible solutions based on depth-first recursive search[1] and it tends to be more efficient than a Branch and Bound algorithm[2]. Also, Brach and Bound algorithms are generally adopted for optimization problems because it is useful to track best current solutions[3].

Backtracking and pruning algorithm, however, has a worse Time complexity than the dynamic programming algorithm due to its recursive nature.

For example, Value 8 with Minimum/Maximum coin 3 is firstly expanded with the backtracking and pruning algorithm as follow:



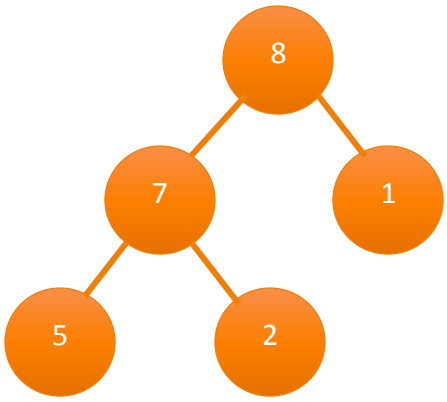
However, leaf nodes [1,1,6] already has the three coins (i.e. checking the number of coins used is first pruning method) and 6 is not a prime number, hence, the algorithm backtracks to the value 7 and splits 7 with the next prime number, 2. The difference between the next prime number 2 and the value 7 is 5, which is another prime number and three coins are used only, thus, this combination [1, 2, 5] is considered as a solution.

---

[1] Lecture Note Module 3 p.16

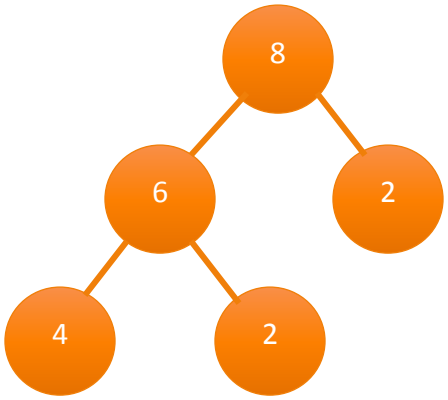
[2] https://pediaa.com/what-is-the-difference-between-backtracking-and-branch-and-bound/#Backtracking%20vs%20Branch%20and%20Bound%20-%20Comparison%20of%20Key%20Differences

[3] Lecture Note Module 3 p.26

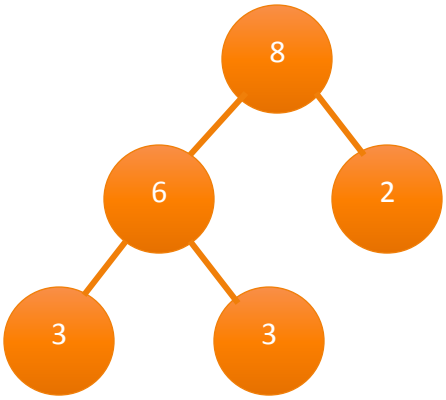


Similarly, [1, 3, 4] is tried but 4 is not a prime number. Next, [1, 5, 2] is not expanded because this list is the equivalent solution of [1, 2, 5]. With this example, the backtracking algorithm incorporates the second pruning condition. That is, if the difference should be smaller than the current coin denomination, the algorithm does not expand further and moves to the next prime number.

After exhausting the total combinations for the value 7, the algorithm backtracks and finds the solutions for the value 8 with the next prime number, 2. The difference between the value 6 and the coin denomination 2 is 4, which is not a prime number.



The value 6 is then tested with the coin denomination 3 and as the difference 3 is a prime number, this is considered as another solution. Because of the incorporated second pruning method, the algorithm does not expand after this for the value 6.



In this way, with the two pruning methods, the algorithm reduces its expansions at least half by removing the invalid larger number of coins and the same combinations as the order of coins does not matter. In addition, the solutions for invalid number of coins used are stored during this algorithm because the number of solutions for the smaller number of coins used is stored while running the algorithm. Thanks to this nature, the algorithm for only the maximum number of coins is run to obtain the number of solutions. These results are stored in a dictionary data structure as a form of {number of coins: number of solutions}. Finally, the number of solutions from this dictionary is added to get the number of solutions between Minimum and Maximum coins.

The details of Time and Space complexity is examined in Performance Analysis. The complete implementation was done with Python (though C++ was also implemented partly for the main algorithms only).

Algorithm Pseudo-Code

Main algorithm consists of two parts: cal_combination_all and cal_combination. First, the method cal_combination_all identifies the total number of solutions for the specified value by using the dynamic programming algorithm.

Figure 1. Pseudocode for cal_combination_all

```
Algorithm cal_combination_all()
        Initialise a 2-D matrix with 1s for two rows and 0 to Value columns
        row = 1, column = 1, row_index = 1
        For row <- 1 to the size of coin_denomination list
                For column <- 1 to Value
                        If column minus coin_denomination[row] >= 0 then x = the
                        matrix[row_index][column - coin_denomination[row]]
                        Else x = 0
                        If row_index=0 then
                                If column >= 1 then y = the
                        matrix[row_index+1][column] Else 0
                        Else
                                If column >= 1 then y = the matrix[row_index-
                        1][column] Else 0
                row_index += 1, row += 1
                If row_index=2 then row_index -=2
        result_row = row_index+1 if row_index = 0 else row_index-1
        return matrix[result_row][value]
```

The second backtracking and pruning method, cal_combination, solves the number of solutions by the number of coins used to make the solutions.

Figure 2. Pseudocode for cal_combination

```
Algorithm cal_combination(integer remaining_value, integer current_coin_position,
integer restricted_number_of_coins, integer current_number_of_coins)

    boolean number_of_coins_check = current_number_of_coins == restricted
    _number_of_coins - 1

    //two base cases and one utility condition

    If (number_of_coins_check is true) and (remaining_value is not in coin_
    denomination list) then return 0 //not a prime number or 1 - not a solution

    If (number_of_coins_check is true) and (remaining_value is in coin_denomination
    list) then (add 1 to the dictionary containing the number of solution by the number
    of coins used) and (return 1) //one of the solutions

    If (number_of_coins_check is false) and (remaining_value is not in
    coin_denomination list) then (add 1 to the dictionary containing the number of
    solution by the number of coins used) //utility condition to keep a track of the
    solutions for the different number of coins

    integer number_of_combinations

    For index <- current_coin_position to the size of coin_denomination list

        difference = remaining_value – coin_denomination list[index]

        If difference >= coin_denomination list[index] then

            current_number_of_coins++

            cal_combination(difference, index, restricted_number_of_coins,
            current_number_of_coins) and add the result to
            number_of_combinations  //recursive call

        Else return 0

        current_number_of_coins--

    return number_of_combinations
```

Results and Algorithm Analysis

Results of the problem

Where the number of total solutions is required, the modified dynamic programming algorithm is used while the backtracking and pruning algorithm is used for all other cases.

It was noted that Python programs takes some time even for a basic program on my computer (i.e. time taken to print out "Hello World"). The time taken for the results below extracted the time taken for a basic program. All time taken was calculated based on the average of the three experiments.

Table 3. Results for Python implementation

Input	Output	Ave. CPU Time (Secs)	Input	Output	Ave. CPU Time (Secs)
5	6	0.000000	**20 10 15**	57	0.000000
6 2 5	7	0.000000	**100 5 10**	14839	0.088542
6 1 6	9	0.000000	**100 8 25**	278083	1.869792
8 3	2	0.000000	**300 12**	4307252	72.968750
8 2 5	10	0.000000	**300 10 15**	32100326	443.703125

The results of Value 5 without constraints and Value 6 with Minimum coin 1 and Maximum coin 6 use the dynamic programming algorithm. The result from the matrix then adds 1 to account for the gold coin solution, hence, 5+1=6 and 8+1=9, respectively.

Table 4. Value 5 without constraints

	0	1	2	3	4	5
3	1	1	2	3	4	5
2	1	1	2	2	3	3

Table 5. Value 6 with Minimum coin 1 and Maximum coin 6

	0	1	2	3	4	5	6
3	1	1	2	3	4	5	7
5	1	1	2	3	4	6	8

All other problems use the backtracking and pruning algorithm. For example, the results of Value 6, Minimum coin 2, and Maximum coin 5 are composed of:

Table 6. Example results from the backtracking and pruning algorithm

The number of coins used	The number of solutions	The solutions meeting constraints
2	2	[5,1], [3,3]
3	2	[2,2,2], [3,2,1]
4	2	[2,2,1,1], [3,1,1,1]
5	1	[2,1,1,1,1]
Total	**7**	

These results are obtained by running the algorithm with the maximum number of coins and storing all possible results in the dictionary form: {(2,2), (3,2), (4,2), (5,1)}. Then the values with the relevant keys were summed up (2+2+2+1) to provide the number of solutions.

Performance analysis

The backtracking and pruning algorithm has the completeness because it searches for all solutions. The results with the backtracking and pruning algorithm implemented in Python are better than the benchmark results for Python implementation on the assignment instruction. For example, CPU Time for Value 300, Minimum coin 10, and Maximum coin 15 is measured as 443.703125 seconds (benchmark: 3992.311160 seconds).

As seen in Pseudo code, searching, comparisons, assignments, recursion calls, subtractions and additions occur each call, thus, it is assumed that these operations are constant, $O(1)\ or\ O(c)$. For simplicity, I will use a constant 1 to represent these repetitive operations. Time complexity of the backtracking and pruning algorithm for the worst case is as follow:

$$T(n) = 2 * \big(T(n-c)\big) + 1\ (or\ O(c))$$

$$where\ T(0) = 0,$$

$$m\ is\ the\ size\ of\ coin\ denomination\ list, \qquad n\ is\ the\ target\ value\ to\ make\ up,$$

$$and\ c\ is\ the\ difference\ between\ n\ and\ the\ specific\ coin\ denomination$$

$$(i.e.\ prime\ number\ smaller\ than\ n).$$

To make this more general and address the worst-case problem,

$$T(n) = 2 * \big(T(n-1)\big) + 1$$

$$T(n) = 2 * (2T(n-2) + 1) + 1 = 2^2T(n-2) + 2 + 1$$

$$T(n) = 2^2 * (2T(n-3) + 1) + 2 + 1 = \ 2^3T(n-3) + \ 2^2 + 2^1 + \ 2^0$$

$$T(n) = \ 2^kT(n-k) + 2^{k-1} + \cdots + \ 2^2 + 2^1 + \ 2^0$$

$$where\ n - k = 0\ (n = k), \qquad T(n) = \ 2^n * 0 + (2^n - 1) = 2^n - 1 \ \leq c * \ 2^n$$

Therefore, the worst case for this algorithm is exponential, $O(2^n)$ although with pruning, it is more likely Time complexity is better than the worst case. The best-case is $\Omega(1)$ where the gold coin case is allowed.

Space complexity of this algorithm is $O(n)$ as the worst case as the algorithm uses depth-first recursion (i.e. it stores information worth the maximum depth) and it does not store information other than a dictionary collection of the number of coins used and the number of solutions, which is always lesser than Value n.

Time complexity of the dynamic programming algorithm is as follow:

$$T(n) = m * n\ (comparisons\ and\ assignments\ equivalent\ to\ 1) \ \leq 2 * mn$$

$$where\ m\ is\ the\ size\ of\ coin\ denomination\ list, and\ n\ is\ the\ target\ value\ to\ make\ up$$

Thus, Time complexity of the Dynamic programming algorithm is $O(mn)$.

Space complexity of the Dynamic programming algorithm for this version (with two rows) is $O(2n) \approx O(n)$.

The dynamic programming algorithm is much faster to find the number of solutions where the total number is required. To illustrate, Value 300, Minimum coin 1 and Maximum coin 300 takes only **0.000000** seconds (extracting the time taken to print out "Hello World") whilst the backtracking and pruning algorithm is not able to finish finding total number of solutions within 21 hours.

Conclusion

Overall, the backtracking algorithm can deal with cases for the specific constraints, but less efficient even with the pruning methods. It seems that there is an opportunity to use the dynamic programming approach combined with the backtracking and pruning approach to improve Time and Space complexity. I tried to construct this algorithm but failed to do so within the time frame as there seems to be many rules needed to be built in to construct the matrix generated by the dynamic programming.