Coin Change Problem

2801ICT Assignment One

s5131071 - Rusty Blewitt - rusty.blewitt@griffithuni.edu.au

2801ICT - Computing Algorithms - Dr. Saiful Islam

Problem

Find the number of different ways you can perfectly pay (cannot overpay) a certain balance using a specific set of coins.

Coin values that can be used to reduce balance to zero:

- 1
- All prime numbers that are less than the balance
- A ***gold coin*** which is the initial balance itself

Subproblems:

- No restrictions on amount of coins used
- Use only a specific number of coins *E.g, use only 6 coins*
- Use an amount of coins in a specific range *E.g, use between 3 and 8 coins*

Inputs and outputs

For each problem that is solved, the algorithm can receive either ***one***, ***two*** or ***three*** integers as initial inputs.

Outputs will always be a single integer representing how many different ways the balance can be paid given the following restrictions imposed by amount of inputs.

1 input given	Input 1	Balance to be paid
2 inputs given	Input 1	Balance to be paid
	Input 2	Amount of coins balance must be paid with
3 inputs given	Input 1	Balance to be paid
	Input 2	*Minimum* number of coins balance can be paid with
	Input 3	*Maximum* number of coins balance can be paid with

Results obtained

These results were achieved using the C++ implementation attached ***"5131071_Pay_in_Coins.cpp"*** on a stock standard *2015 MacBook Air* with a *1.6ghz Intel Core i5* processor and *4GB 1600 MHz DDR3 RAM.*

Input	Output	Time (secs)	Input	Output	Time (secs)
5	6	0.000126	20 10 15	57	0.000051
6 2 5	7	0.000027	100 5 10	14839	0.005313
6 1 6	9	0.000018	100 8 25	278083	0.060194
8 3	2	0.000018	300 12	4307252	3.895810
8 2 5	10	0.000017	300 10 15	32100326	17.199585

Comparing results for these specific inputs to the corresponding given in the project outline it can be determined that this algorithm produces correct results for the given inputs in a reasonable time.

Functions created for algorithm

```
// Checks if a value, n, is in a vector, v.
bool in_vector ( int n, vector<int> v )

// Uses a Sieve of Eratosthenes approach to compute all primes < n and returns a vector
// of 1, aforementioned primes and n itself
vector<int> get_coins ( int n )

// This is the algorithm itself, a recursive function that takes the balance that needs to
// be paid, a pointer to the coins vector, additional arguments and an int to track how many
// coins have been used so far as inputs and returns an int (the corresponding solution)
int recurse( int balance, int* coins_ptr, int arg1 = 0, int arg2 = 0, int coins_used = 1 )
```

Algorithm pseudocode

---

function get_coins (n) : // Gets primes using a Sieve of Eratosthenes approach

coins [ ] <- [1, *all primes below n*, n];

return coins;

---

function recurse (balance, coins, arg1, arg2, used) :

solutions <- 0;

while (coins not empty) :

newbal <- balance - (first coin in coins);

if (*newbal == 0* **) :** // If balance paid perfectly

if (additional args given and satisfied):

solutions += 1;

break while loop;

else if (*newbal < 0* **):** // If now overpaying

break while loop;

else: **//** If balance still exists

if (*additional args given and are now violating them* **):**

break while loop;

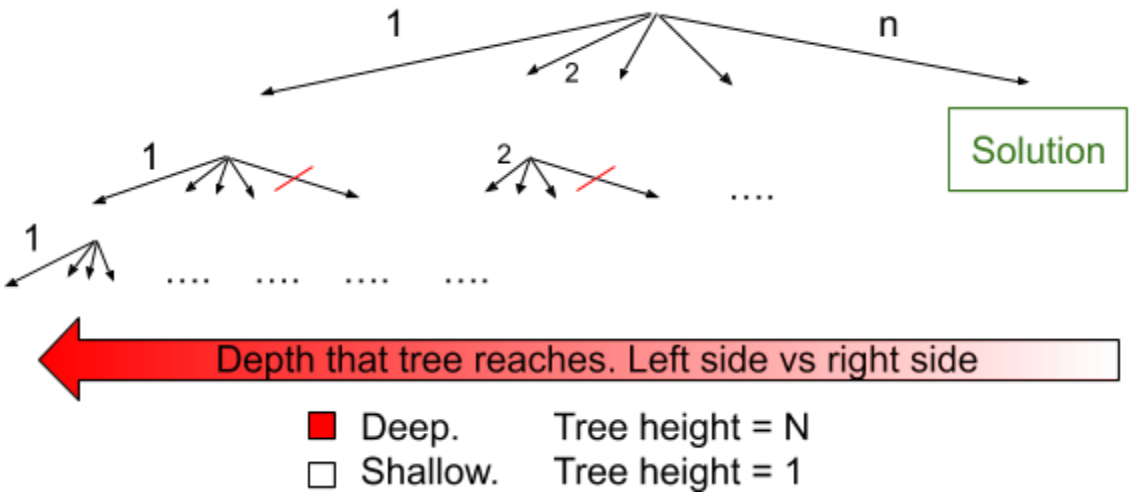
else:

solutions += recurse (newbal, coins[1:], args..., used +1)

return solutions; // All coins have been tried for this level of recursion, return.

Algorithm Analysis

The algorithm used performs well when comparing times with the given example results as can be seen on page 2. The algorithm itself is essentially a depth-first search which is achieved using recursion.



The diagram above represents the search process for valid combinations of coins (solutions). At each level of the tree, the while loop will check if using the current iteration of coin will result in the balance being **above**, **equal** to or **below** zero.

If **below**, the while loop will break, essentially *pruning* the tree by not expanding any further for that iteration or the following in the current loop. This pruning is represented by a red slash in the above diagram.

If **equal** to, the while loop will note that it has found a solution, then prune as using an additional coin (recursing) or using a greater value of coin (next iteration of while loop) will result in overpaying.

If **above**, this means the balance is not yet paid and therefore before continuing to try iterations of higher valued coins, it will also call itself (**if** doing so will not violate arguments given). This recursion is taking the current coin as one coin that will be used to pay the balance and now moving forward to see what the next coin it will use will be.

The red arrow in this diagram indicates the depth that the tree will reach when comparing the left and right sides. Due to the pruning, the right side will be noticeably more shallow as higher values of coins will result in sooner goal reaching / pruning.

Algorithm Analysis (Time Complexity)

Function by function breakdown.

- Function: in_vector( ) = $O(n)$

  Explanation: In the worst case scenario, the value that is being searched for is either nonexistent in the vector or it is the last item in the vector. In these cases the operations undertaken will be a constant ***C1*** (for loop overhead), plus a constant **C2** (equality comparison) times **n**.

  Removing constants, it therefore can be said that the function in_vector has an asymptotic upper bound of **O(n)**.

- Function: get_coins( ) = $O(n^2)$

  Explanation: The outer for loop of this function will always be executed just under **0.5n** times => **Θ(n)** as this could be tightly bound for example by a function 0.4g(n) and 0.6g(n).

  The inner for loop will be reached only a certain amount of times, dependant on the size of **n**. This coefficient of **n** will diminish as the value of **n** increases but for the sake of brevity and only at the cost of accuracy in the analysis of this auxiliary function, it is valid to say that this inner for loop is O(n).

  The inner for loop will execute once again, an amount of times less than n and again this is best represented as O(n).

  Using the above calculations it can be determined that this function has a time complexity of $O(n^2)$.

- Function: recurse( ) = $O(n^n)$

  Explanation: The while loop will always iterate less than n times as it is only iterating over the prime numbers and for each iteration at most n recursions will follow (for the case of using all 1 coins). There is plenty of pruning involved

throughout this process and the amount of pruning will also vary dependent on the constraints so most often the amount of operations will be well beneath the asymptotic upper bound of $O(n^n)$ however this best represents the rate of growth.

The aggregation of these functions leave us with a time complexity of $O(n + n^2 + n^n)$, which can be simplified to $O(n^n)$.

Algorithm Analysis (Space Complexity)

Function by function breakdown.

- Function: in_vector( ) = $O(n)$

  Explanation: This function completes it's operations in place, however, when we pass a vector to a function in C++ it creates a copy of the vector, therefore for a vector of size n we will use some constants plus a storage of size n to execute the function.

- Function: get_coins( ) = $O(n)$

  Explanation: Bar auxiliary values such as the vectors themselves and the for loop overhead, this function will create two vectors whose sizes will each become increasingly less than n as n grows to large values. These can be viewed as taking $C + x * n$ each, $C + 2xn$ which has an asymptotic upper bound of $O(n)$

- Function: recurse( ) = $O(n)$

  Explanation: The *recurse* function only uses pointers and auxiliary variables. The accumulation of space comes from the amount of functions which will be *open* during the algorithm's execution. As this is a depth-first search, the algorithm will only ever have *n* functions open as the tree height will only ever be n at a maximum. As the function takes 5 arguments and an additional variable

new_balance is created within each function. We can say that recurse takes up 6 space. As we will have *n* open *recurse* functions at any given time, we can say that the space used at any given time will be 6n, which is therefore **O(n).**

The aggregation of these functions leave us with a space complexity of $O(n+n+n)$, which can be simplified to $O(n)$.

Summary

This algorithm is 'correct' based off the results which are produced as expected and in the expected amount of time while using negligible amounts of space.

The get_coins() function can be reused with slight modifications (removal of the addition of 1 and n to the vector that is returned) for any problems that require prime number generation.

Solving this problem allowed for a deeper understanding and exploration of how to measure time and space complexity and how these measures are important to consider in algorithm choice when input values grow high.

As technology progresses it will be of great interest to computer scientists and enthusiasts alike to revisit problems such as these periodically to evoke an appreciation for the advances in computing speeds, especially at the time of this report being written as we approach the age of quantum computing.

End of document.