2801ICT – Assignment 1

Algorithmic Design Report

Contents

Overview

This report details an implementation of an algorithm to solve the problem of finding the number of ways a dollar amount can be paid in coins. These coins have values of prime numbers. A limit can also be given on the amount of coins (including 1) that can be used to calculate the integer, either with an upper limit or both an upper and lower limit. For example, the amount of prime coins 5 has is six:

5	2 + 3	1 + 2 + 2
1 + 1 + 3	1 + 1 + 1 + 2	1 + 1 + 1 + 1 + 1

Note that 2 + 3 and 3 + 2 are the same sum. If given a lower limit of 2 and an upper limit of 5, 8 has ten partitions:

1 + 7	5 + 3
3 + 3 + 2	5 + 2 + 1
2 + 2 + 2 + 2	5 + 1 + 1 + 1
3 + 2 + 2 + 1	3 + 3 + 1 + 1
1 + 1 + 2 + 2 + 2	1 + 1 + 1 + 2 + 3

C++ was chosen as the language to solve this problem and three functions are used, one of which is the driver function.

Algorithm Description

Two algorithms are used; the first calculates all primes up to (and possibly including) a given integer. The second is the focus of this report and it calculates all solutions to the given problem. A third function is the driving part of the entire program.

Function: Primality Checker

A rudimentary execution of an iterative method for calculating prime numbers, a brute force approach was employed. Given an integer *n*, this function will check for any number *x* less than *n* and greater than 1 if the modulus of *n* and *x* is 0. If the result is 0 then the function returns false but if no statements evaluate to 0 it will return true. Hence, if there is no such number less than *n* (that is not 1) that *n* can be evenly divided by, then *n* is a prime.

Pseudocode

```
PrimalityCheck(n):
// Input: n
// Output: True if the integer is prime, false if it isn't.
for i ← n – 1 down to i > 1 do:
        if n % i = 0:
                return false
        end if
end for
return true
```

Algorithm: Prime Partitions

This algorithm performs the search for solutions to the problem using a branch and bound programming paradigm. It uses three variables; one to keep track of the amount of coins that can be used (limited by the input), the current sum of primes, and a list of prime coins for each branch of the search tree. Each node of the tree is a prime, *n*, and its children are all of the primes of *n* greater than or equal to *n (Figure 1)*.

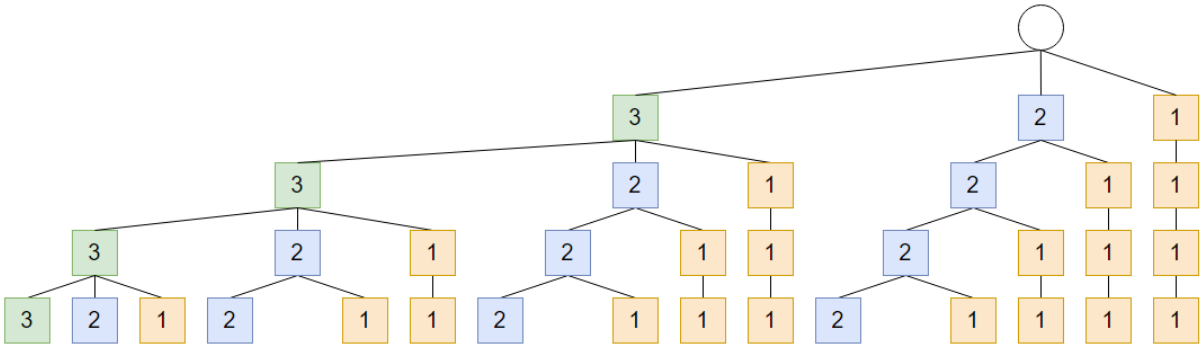

***Figure 1**: A slightly pruned search tree for n = 3.*

Every node at each level has its prime subtracted from the current sum, the number of usable coins decremented, and is then passed into a recursive call.

There are various cases in which the recursion will stop:

- Cases which return a solution:
  - If the algorithm has used up the total amount of coins given to it initially and the sum has reached 0. This means that the algorithm has found a list of integers whose sum is equal to *n* and it has done so within the limit of coins provided.
  - It could go the other way and if both the sum and coins allotted are equal, a solution has been found.
- Cases which return a failure:
  Any number of combinations of the sum and allowed coins that weren't mentioned above will return a failure. This is because the sum and number of coins must align. I.e., if we have more coins left to create a sum, but the sum has reached 0 or *n*, then it has not satisfied the requirement that is must use *m* coins to sum up to *n.* The same can be said for the reverse; if limit of coins has been exhausted but there are more numbers that need to be added to the sum that is another failure.

Now that the base cases have been defined, they can be used to limit the search space.

Pruning

Already the tree that is used to search for solutions has one form of pruning applied to it; each node's children will only contain primes less than or equal to itself. In this way the algorithm will never move backwards but can add numbers to a sum more than once.

Pruning based on digit limit allows the algorithm to ignore primes if the potential sum will exceed or fall under the limit of coins provided by a lower and upper bound. For instance, if the algorithm needs to find a sum for *n* = 3 with only two coins then it can only have the sum:

2 + 1

Hence the search space can be drastically limited (*Figure 2*) as there are no other branches that can both use two coins in the sum *and* sum up to 3.

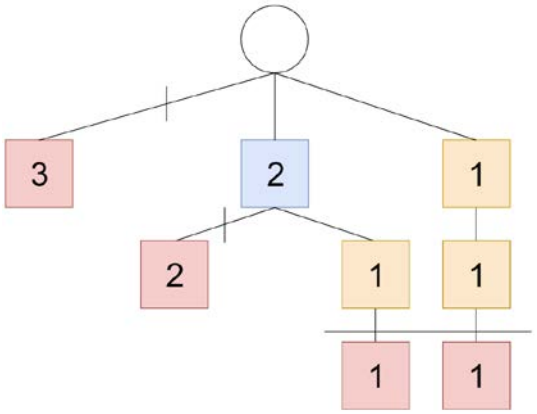

Figure 2: *Pruning based on the number of coins required for a solution.*

Conversely, the same logic can be applied to the sum of primes being searched. When a sum solution has been found, but the number of coins used does not meet the requirements then no more solutions will be found in the current branch.

Once all branches have been searched, the resulting number of solutions is returned.

Pseudocode

```
PrimePartitions(coins, sum, primes)
// Input: coins → an integer representing the limit of coins that can be used in a sum.
//        sum → an integer to keep track of the sum of the primes in the current branch.
//        primes → a list to keep track of the current primes being used to calculate solutions.
// Output: The amount of solutions that were found in the search space.
if coins = 0 and sum = 0 or coins = sum:
        return 1
end if
if sum ≤ 0 and coins ≠ 0 or coins = 0 and sum = 0:
        return 0
end if
solutions ← 0
for i ← 0 to primes.length do:
        newPrimes ← primes[i…primes.length]
        coins ←coins – 1
        sum ←sum – primes[i]
        solutions ←solutions + PrimePartitions(coins, sum, newPrimes)
end for
return solution
```

Results & Algorithm Analysis

Predictions, analysis, and results for this solution will be detailed in this section of the report. Space complexity of the main algorithm and prime calculation function are not considered, however their time complexity in both the worst and best cases will be analysed. Once calculated these results will be used to make assumptions on the way the algorithm will behave based on different input sizes. Then those assumptions will be compared with the actual results.

Algorithmic Time Complexity

Primality Checker

To properly assess the time efficiency of this function, input will be measure as the magnitude of the given integer *n*. The basic operation of the function is:

$$if\ n\ \%\ i = 0$$

Representing this statement with *x* gives a summation of:

$$\sum_{i=n-1}^{2} (x)$$

Therefore, the best-case scenario is if *n* = 1, in which the function will simply return true. Hence:

$$T(n)_{best} = \Omega(1)$$

However, for magnitudes of *n* that are larger than 1 the basic operation will be executed *n* amount of times. In this case, the functions time complexity can be given by:

$$T(n)_{worst} = O(n)$$

Prime Partitions

The time complexity of this algorithm relies on the number of coins that are allowed for a solution and the size of the list of primes being passed to it with each call. But the recursive call is dependant of the size, *n,* of the prime list, so that is used to assess the overall time efficiency.

For each call of the function, several comparisons are made to check the base cases. These take 1 unit of time to complete. When the function recurs, it does so *n* times and each time with one less list element each time. Therefore, the following recurrence relation can be used:

$$T(n) = \begin{cases} 1 & n = 0 \\ nT(n-1) + 1 & n > 0 \end{cases}$$

Solving this relation gives:

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

$$n - k = 0$$

$$\therefore n = k$$

$$\therefore T(n) = n^n + n^{n-1} + n^{n-2} + \cdots + 1$$

This gives a summation of:

$$T(n) = \sum_{k=0}^{n} n^k = O(n^n)$$

Results

Input	Output	CPU Time (seconds)
5	6	0
6 2 5	7	0
6 1 6	9	0
8 3	2	0
8 2 5	10	0
20 10 15	57	0
100 5 10	14839	0.109
100 8 25	278083	5.125
300 12	4307252	43.36
300 10 15	32100326	451.449