2801ICT Computing Algorithms Assignment 1 Report (Coins)

Ovi (Ovidu Opris) - s5133659

Griffith University Gold Coast

# 

# 

# 2801ICT Computing Algorithms Assignment 1 Report (Coins)

# Project Overview

## Task

The task is to create and come up with an algorithm that will, given a set of inputs read from a file, print out the number of ways a set amount can be paid for in coins, where the value of those coins may be any prime number, and including the value of the total amount itself.

## Solver Requirements

The first stage in solving the the main part of the program is generating the primes for the values of the coins, and then finding out all the possible arrangement of coins that could be added together to come up to the value desired.

The solver is then required to output these results to a file at ./Output.txt relative to the working space the solver is running in.

## Source

The project source may be found on GitHub at <https://github.com/SirHall/coins>.

## Building

The build system used is *scons*, which is a python-based build system for C, C++, Java, Kotlin, C#, etc. This ensures that the project can easily be built on Windows, Linux, and MacOS(untested). To install *scons* enter pip3 install scons into a terminal, move to the project root, and enter scons.

There are multiple settings that can be used while building the source, namely the bt (build type) and mt (multi-threading) settings. To use a setting simply follow it with an equals sign and the setting desired as follows: bt=standard

#### bt has three settings: standard, debug and release.

standard is the standard builder with no compiler optimisations enabled, if bt is not set, it will default to this setting.

debug is the same as standard, but enables debugging for the built binary to be used with GDB or other debugging software, it however does not enable any compiler optimisations.

release is used when publishing the build, it enables -O3 and -ffast-math optimisations which drastically increase the performance of the solver, from this point onwards all times will be recorded using release. Also note, that during Windows testing this setting did not work. If testing on windows make sure to only use the standard or debug build settings.

#### mt itself has two settings, true and false.

Setting mt to true, will enable multi-threading. This implementation of multi-threading does not split the workload of a single problem across the cores, but rather if there are 12 problems that need to be solved, each core will pull from a central queue whenever it finishes a problem, and push the results to a result list in order. This implementation uses the pthread (posix thread) library, which is not officially supported on Windows. If building on Windows either set mt to false, or don’t set it to anything(if not set, mt will default to false).

# Implementation

## Prime Number Generator

The prime number generator simply starts with a base list of {1, 2, 3}, using these base primes it generates primes up to any value required. It increments by 2, (to filter out all even numbers), and tests it against all previous primes generated to test if their modulus results in a 0. If it does, then it is not a prime and the value is incremented by 2 and checked again against all previously generated primes. If the new value cannot be equally divided against any of the previous primes it is added to the primes list, the value incremented by 2, and the checking continues until the value reaches the max desired.

A certain optimisation is to ensure that the primes being checked against the value never go above the square root of n, as anything above this is a redundant check. It was decided that this optimisation was not required as in testing it was found that generating primes of even up to 1,000 can be generated once in 7.2122 ⨯ 10-05 seconds, since even problems of this scale largely fall outside the scope of this project it was decided that a brute force prime number generator was more than fast enough.

## Coins Solver

The coin solver firstly scores, the maximum, and minimum number of coins that can be used to fulfill the desired amount. It then goes to the first coin index, and selects the smallest possible coin size(a value of 1). It then moves to the second slot, and places the smallest value in it until the end slot is reached. If this is a solution, it is added to a solution counter, otherwise the algorithm recurses back to the original coin index. The first coin index’s value is then set to the next prime index, a prime index of 1 returns a prime value of 2, the second prime. All following coin indices can then only be filled with coin values that are less than, or equal to the preceding coin indices values, this ensures that no duplicate coin combinations are generated.

##### For any coin index ‘i’, it’s prime value can only be less than or equal to the prime value immediately preceding it(i - 1), this ensures that all coin values are sorted and that no duplicate coin combinations are tested.

For any index, if the sum of all coins reach the amount desired, the program will check to see if the coin slots used up falls within the inclusive range set by the input. Therefore if the desired amount amount = 100, minCoins = 5 and maxCoins = 10, it will make sure that the coins used up falls within the range 5..10 inclusively. Doing it this way instead of having a set number of coins and iterating through all possible coin counts separately also helps to drastically increase the performance of the program.

Since the prime number list is pre-generated, if at any time the next prime number goes above the total amount left for any coin slot, the function immediately returns.

Any time the the program moves to the next slot, it will calculate the amount left, which is the (totalAmount - sumOfAllCoinsUsed). For each slot, if (primeUsed < amount), recursively move to the next coin slot and repeat. If (primeUsed == amount && slotsUsed >=minCoins && slotsUsed <= maxCoins), then increment the solutions, and return, otherwise simply return.

A simplified version of the recursive function can be see as follows:

**int** SolveRecursively(amount, primeIndex, spacesLeftMin, spacesLeftMax):

**if**(spacesLeft **==** 0 **OR** primes.GetPrime(primeIndex) **>** amount):

**return** 0

solutions = 0

**for**(i = primeIndex; i **<** primes.TotalPrimes(); i**++**):

**if**(primes.GetPrime(i) **>** amount)

**break** **//**Our chosen prime **is** larger than the amount

#We've reached the end and found a solution

**if**(spacesLeftMin **==** 1 **&&** amount **==** primes.GetPrime(i)):

solutions**++**

**if**(spacesLeftMax **>** 0):

solutions **+**= SolveSingleRec(

amount **-** primes.GetPrime(i),

i, #Our current prime index

spacesLeftMax **-** 1, #Decrease avaliable max slots

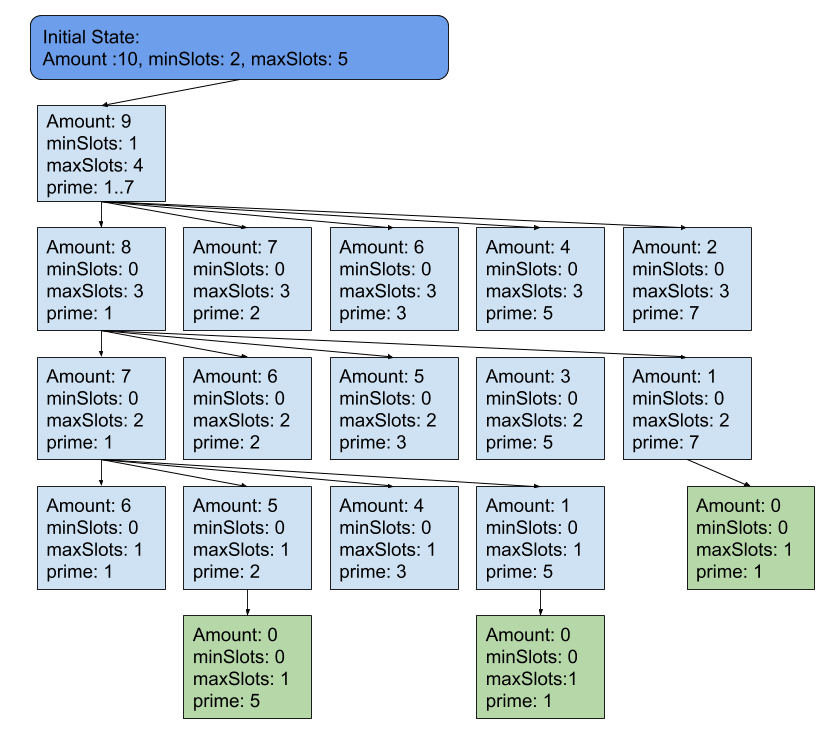
#Decrease avaliable min, preventing overflow

spacesLeftMin **>** 1 ? spacesLeftMin **-** 1 : 1

)

**return** solutions

A visualisation of how the program works: (Only part of the search tree is displayed)



## Multi-Threading

This solver can be compiled to use a thread pool which pulls problems and solves them locally(not supported on Windows, Linux only). After solving a problem, the threads will then place the solution in a solutions list, in the original order.

If multi-threading is enabled at compile time, the program will ask when it runs the number of worker threads the user wishes the program to create, each thread will then pull a problem from a central list, solve it, push the answer into a central answers list, and repeat until all work is done.

# Performance

## Close Analysis

For the standard input list:

5

6 2 5

6 1 6

8 3

8 2 5

20 10 15

100 5 10

100 8 25

300 12

300 10 15

The program takes 5.78979s to complete the entire list on an Intel i7-6500U running at 3.1GHz. However, a different metric is required to find the generalized time complexity of the solver, primarily the total cycles performed by the solver. For an input of 300 20, the solver goes through:

7,622,207,568 cycles  
 61,447,518 solutions

26.6723 seconds

That’s an average success rate of 0.8%, meaning that 0.8% of all cycles will find an answer, where the average time per cycle is at 3.499288 ⨯ 10-9 seconds, and the average time to find a solution is 4.340664 ⨯ 10-7 seconds.

However, to find a more full image of the number of cycles required to fulfill some problem, an input sheet was setup where all total amounts were 300, where the number of coins(min and max) ranged from 1..15.

A second sheet was written up, where the total amount increased from 10 to 300 in varying increments and where the coin counts remained at a constant 20.

While this is a greatly simplified way of analyzing the program, it would not be realistically feasible to find a generalized equation that finds the total cycles made given the total-amount, minimum coins and maximum coins, as each one drastically changes the decision making process of the algorithm. This is also why master theorem cannot be applied to this program. It would also be incorrect to assume that if one wanted to find the cycles taken to solve ‘300 5 12’, that they could do ‘cycles(300 12) - cycles(300 5)’, because the coin counts themselves are not treated as separate iterations, but is taken into account and allows the algorithm to cut off early if it finds a solution before reaching the maximum coin slots. Therefore, ‘cycles(300 5 12) ≠ cycles(300 12) - cycles(300 5)’. This in turn means that there is no simple way of producing an algorithm that will print out the exact number of cycles required to find all solutions to a problem, and since the subproblem size is variable and varies wildly from each step to the next, the master theorem cannot be applied.

The solution gone for on the other hand, is if one wanted to find the time required to find the solution to 300 30, the user would generate a list of problems where the coin count increments towards 30. The resulting cycles for all the inputs can than be analyzed using regression, and an equation formed that should be able to reliably predict the cycles required to search 300 30.

Input data:

300 1

300 2

300 3

300 4

300 5

300 6

300 7

300 8

300 9

300 10

300 11

300 12

300 13

300 14

300 15

Coins vs Cycles

1 63

2 1,321

3 13,972

4 96,901

5 482,413

6 1,898,595

7 6,092,128

8 16,847,353

9 40,661,227

10 88,723,486

11 176,052,986

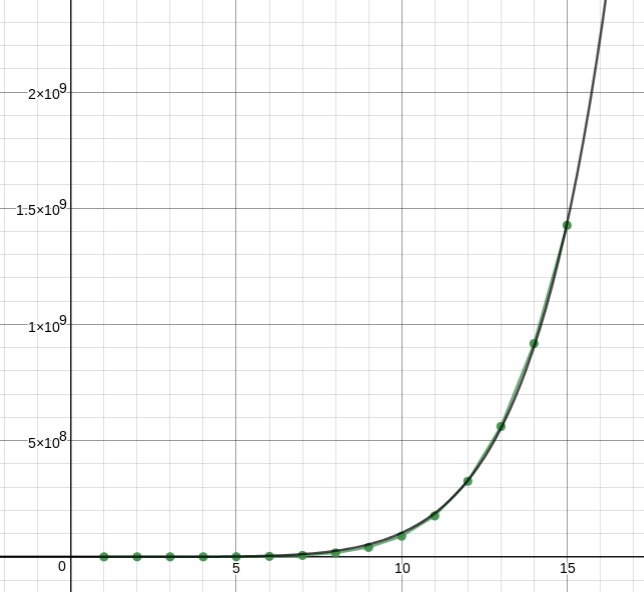
12 325,177,194

13 560,793,766

14 917,459,768

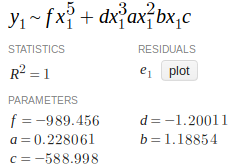
15 1,426,416,733

Graphed in green, (trendline in green)



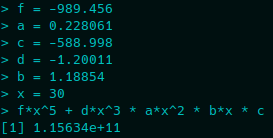
This graph shows that the time gain for larger coin counts is in-fact exponential.

Regressive function:



This function having an R2 of near 1 (this would be down to floating point rounding errors, and not a large enough floating point representation being used) means that assuming given a total amount of 300, the regression function should be able to very confidently give the approximate cycles for 300 30.

Using this data, these values, the approximate cycles for 300 30, was calculated using R:



Finding the solution to 300 30 would take approximately 1.15634 ⨯ 1011 cycles. Given that the mean time of a cycle is approximately 3.499288 ⨯ 10-9, therefore the expected time to solve 300 30 should be approximately 404.6367 seconds.

When run it was found that the solver completed the search in 161.11 seconds which is 39.82% of the predicted time. Whilst this may be seen as largely inaccurate at first glance, it is worth remembering that all computations shown here were done in R, which primarily uses double precision floating point arithmetic. For regression analysis the Desmos Graphing Calculator was used, which primarily works using single precision floating point arithmetic. Considering the sizes of the exponents of the values used, a 40% inaccuracy is incredibly close given range of values used, and how far ahead of the real data points the test was extrapolated, especially since the expected accuracy was expected to be off by orders of magnitude(or off by several digit places).

The real data points only go up to a coinCount of 15, whilst a coinCount of 30 was tested, this combined with the aforementioned arguments would lead to the conclusion that this in-fact a very good way of predicting the compute time of a specific problem, though for real applications it would be wise to use double precision or higher(like long double in C/C++) in all floating point calculations.

This performance analysis detailed a method which may be used in the future to predict the estimated time it would take to solve a specific problem using common regression techniques.

# 

# Conclusion

Lorem ipsumus