Prime Number Generation and Testing Document

Kenan Karavoussanos 1348582
Marc Karp 1356562
Preshen Goobiah 1355880
Shaylin Pillay 1060478

School of Electrical and Information Engineering,
University of the Witwatersrand, Johannesburg 2050, South Africa

September 10, 2018

Contents

1 Introduction

1.1 Problem Statement

The main goal of this lab is to implement and understand the testing procedures for software. It will be achieved by developing and implementing a function PrimeNumbers() that takes a positive integer number X, and generates all prime numbers less than or equal to X. Thereafter, to develop a main program that serves as a test-driver to test various scenarios of use of the module that contains the PrimeNumbers() function.

1.2 Overview of Solution

The Sieve of Eratosthenes prime number generating algorithm was implemented in this work. The algorithm was coded along with various helper functions to enhance the modularity of the solution. This was implemented in the Python programming language. Furthermore, unit testing is done for the main algorithm and aforementioned helper functions of the module Lab2.py. This is done using the unittest Python Unit Testing Framework in the $test_Lab2.py$ module. The $test_Lab2.py$ module is separated by different test conditions described in the scenarios section later in this document. The tests have been written to maximize test coverage while preventing redundant testing.

Due to the nature of integration testing and the fact that there is only a single module. Integration testing cannot be performed. However potential strategies for Integration testing are described later in this document.

1.3 How to run the program and its tests

2 Program Description

2.1 Project Requirements

2.1.1 Module Requirements

The following describes the functional requirements of the Prime Number Generator:

- The Module shall take in a positive integer X as input.
- The Module shall return the list of prime numbers, sorted from smallest to largest, less than or equal to X
- The Module shall return a meaningful error code and message if the input is not a positive integer greater than 1.

2.1.2 Testing Requirements

The following describes the requirements for testing the Prime Number Generator:

Unit Testing A single unit in the scope of this system is defined as a single python module, namely Lab2.py. The unit test requirements for this module are as follows:

- The Test-Driver shall test for valid output when given valid input i.e a positive integer.
- The Test-Driver shall test for valid output when given various boundary cases between invalid and valid input.
- The Test-Driver shall test for correct error codes and messages when given invalid input.

Integration Testing Due to the existence of only one module in the system (other than the Test-Driver) there is no need for integration testing at this point. However, a strategy for future integration testing is discussed later in this report.

System Testing Similarly to Integration Testing, the existence of only a single model prevents full system testing beyond the unit testing done on the Prime Number Generator. However a strategy for system testing is described later in this report.

2.2 Core Algorithm

Sieve of Eratosthenes is a simple and ancient algorithm used to find the prime numbers up to any given limit. It is one of the most efficient ways to find small prime numbers[2]. The idea is to find numbers in the table that are multiples of a number and therefore composite, to discard them as prime. The numbers that are left will be prime numbers[3].

2.3 Running the Program

A user opens the "Lab2.py" file using a terminal. The terminal then runs the program, which asks the user to input a number greater than 1 of their choice. Once inputted, the program then runs and computes a list of prime numbers less than or equal to the given inputted number. The output is then shown in an array to the user, using the terminal as an interface.

3 Testing

3.1 Description

Due to the scope of the testing only being one module, it is unnecessary to perform integration testing, as such, unit testing has been performed on this module with checks on:

1. Given valid input, is the correct output produced?
2. Given edge cases as input, is the correct output produced?
3. Given invalid input, are the correct exceptions raised?

3.2 Testing Scenarios

3.2.1 Description of input file

Constraint

3.2.2 Unit tests

Unit testing is used since any errors contained in each individual unit of the source code could lead to incorrect output being produced. The aforementioned test is broken up into three macro subtests, namely test_generate_eratothenes_list, test_remove_composites and test_PrimeNumbers. These test whether the Eratosthenes list has been generated correctly, whether all composite numbers have correctly been removed and whether the output of prime numbers is correct, respectively.

test_PrimeNumbers: This macro test is used to validate whether the list generated by the algorithm is correct.

- Test a small correct input

  This scenario tests for correctness given a small valid input. If a small number, such as "7", is entered, a small list of outputted numbers should be returned in particular [2,3,5,7]. This test will ensure that the algorithm does not crash or produce an incorrect output when there is little computation to be done.

- Test a large correct input

  This scenario tests for correctness given a large valid input. Similarly to the above test case, if a large number is entered as input, the algorithm needs to run correctly and not crash due to the given number being too large.

- Test a Value Error (Input: 1)

  This scenario tests whether a "1" was entered as an input, which is not accepted by the algorithm. A prime number is defined as a natural number greater than or equal to 2 with 2 factors, 1 and itself. Since 1 is less than 2 and only has 1 factor, it is not a prime and hence no primes can be generated that are less than or equal to it.

- Test a Value Error (Input: 0)

This scenario tests whether a “0” was entered as an input, which is not accepted by the algorithm. Similarly to the above, “0” is less than 2 and hence no primes can be generated that are less than or equal to it.

- Test a Value Error (Input: -1)

  This scenario tests whether a “-1” or negative number was entered as an input, which is not accepted by the algorithm.Similarly to the above, “-1” is less than 2 and not a natural number and hence no primes can be generated that are less than or equal to it.

- Test a Type Error (Input: 6.4)

  This scenario tests whether an integer was entered as input. If not, then it is not a natural number and hence the algorithm will not be able to compute a list of primes less than or equal to it.

test_remove_composites This macro tests whether all composite numbers have been removed by the algorithm. This will ensure that the algorithm is performing accurately.

test_eratosthenes_list

3.2.3 Integration Testing

The following describes a strategy for Integration testing should more modules be added to the system.

A Detailed design document that clearly defines all interactions between components is required to perform good integration testing. The correctness of these interactions is what is tested during integration testing so working from a well defined source of truth is paramount.

To conduct Integration testing via the bottom-up approach, the design of the system needs to be broken into distinct sections i.e grouping modules that should integrate to perform a single task. Each module in a group should be unit tested before integration testing. The interface between these components is then tested. This testing is typically done in a pairwise manner on each pair of modules. Once the pairwise testing is complete the full group of modules is then integrated and testing occurs once again.

With bottom-up testing the higher level components that run the modules typically do not exist at the integration testing phase so a Test Driver is necessary to act as the module that facilitates the interaction between the tested modules.

3.2.4 System Testing

The following describes a strategy for system testing should more modules and/or groups of modules be added to the system.

First and Foremost a well defined system requirements document should exist. The meeting of these requirements is what is tested at this phase. This testing is considered Black-Box testing so no prior knowledge of the system is necessary to perform this testing procedure.

This phase of testing is done after integration testing where the whole system is integrated into a complete system.

A list of requirements should be given to the tester (preferably a tester who has not used the system before). The tester should then seek to evaluate whether each relevant requirement in the system requirements document is met.