

programming assignment

by Alin Astefanei

Submission date: 29-Apr-2025 02:28PM (UTC+0100)

Submission ID: 257122851

File name: 58288_Alin_Astefanei_programming_assignment_653609_1668562612.docx (110.21K)

Word count: 3208

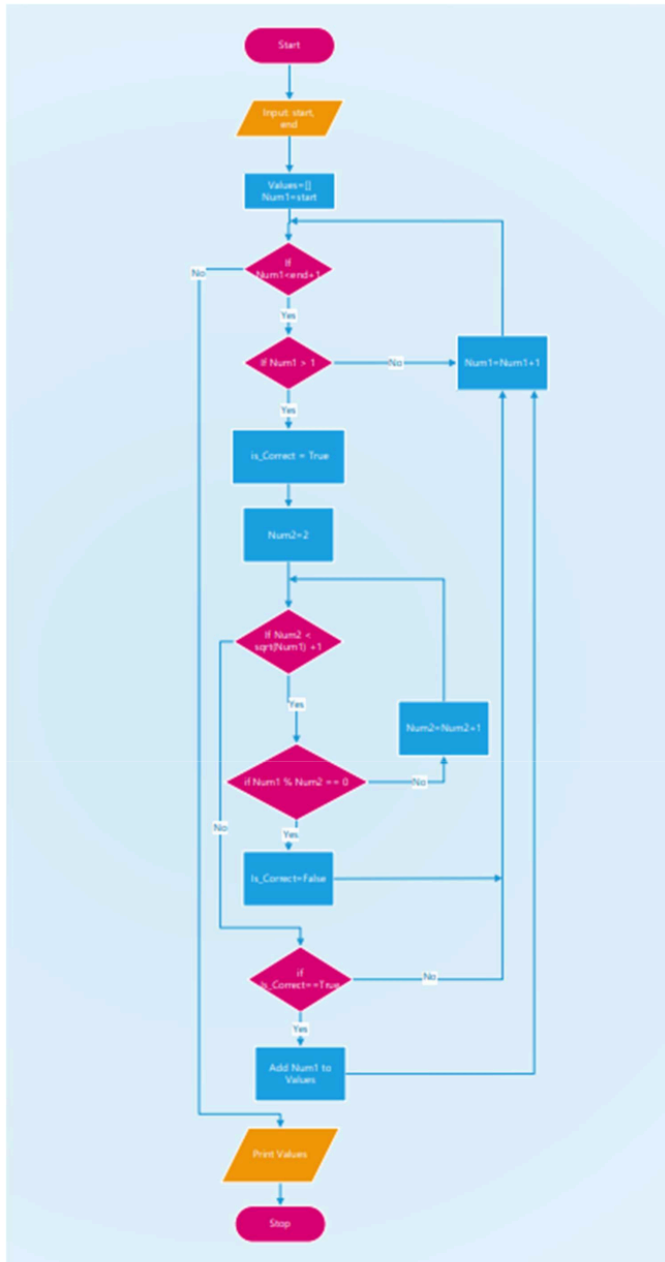
Character count: 15560

Introduction to computer programming

ACCA4020

Table of contents:

- **Q1. Flowchart Purpose and Loops**
Understanding the overall aim and the structure of the loops in the flowchart.
- **Q2. Loop Interaction and Boolean Variable**
Explanation of how loops interact and the role of control variables like Boolean flags.
- **Q3. Conditional Statements in the Flowchart**
The logic behind key conditions used to determine prime numbers.
- **Q4a. Python Program Implementation**
A step-by-step translation of the flowchart into functional Python code.
- **Q4b. Testing the Program**
Test cases demonstrating that the program works with valid inputs.
- **Q5. Validation with Edge Cases**
How the program handles unusual or boundary inputs.
- **Q6. Optimization Techniques**
Efficiency improvements made to reduce computational load.
- **Q7. Reflection on Learning**
Personal insights gained during the assignment and programming process.



1a. Describe the purpose of the flowchart. What problem or task is the flowchart designed to address?

The flowchart is designed to solve a very clear and practical problem: finding all the prime numbers between two numbers provided by the user. When the program starts, it asks the user to input a starting number (start) and an ending number (end). From there, it goes through each number in that range one by one, checking if each number is a prime. If it is, the number gets added to a list, and by the end, the list contains all the prime numbers found between start and end.

To quickly explain, a prime number is any number greater than one that can only be divided evenly by 1 and itself. So, for example, 5 is prime (only divisible by 1 and 5), but 6 is not (divisible by 2 and 3 as well).

The flowchart uses a smart method for checking if a number is prime. Instead of checking every number up to $\text{Num1} - 1$, it only checks divisibility up to the square root of the number. This is a well-known shortcut because if a number has a factor larger than its square root, the other factor would have already been found as a smaller number. This simple trick saves the program a lot of time, especially when dealing with larger numbers.

The process starts by creating an empty list called Values to store the prime numbers it finds. It then sets Num1 to the starting value and loops through each number up to the ending value. For every number, it first checks if it's greater than 1, because primes must be greater than 1. If it passes that check, the program assumes it might be prime and sets a Boolean flag called `is_Correct` to True.

Then, using a second variable Num2, the program checks if Num1 is divisible by any number between 2 and the square root of Num1. If it finds a number that divides evenly into Num1, it marks `is_Correct` as False — meaning the number is not prime. If, after checking all possibilities, `is_Correct` is still True, it adds the number to the Values list.

In short, this flowchart is all about identifying prime numbers in a range using a simple but efficient method. It makes good use of programming concepts like loops, conditional statements, Boolean logic, and optimization (through square roots) to achieve its goal in a clean and structured way.

Overall, it's a good example of how careful thinking and smart programming techniques can make solving even a classic problem like finding prime numbers more efficient.

1b. Identify the number of loops used in the flowchart and explain the role of each loop.

When you look closely at the flowchart, you'll notice there are two loops working together:

- an outer loop, and
 - an inner loop.
- Each loop plays an important role in the overall task of finding prime numbers.
-

The Outer Loop

The outer loop is responsible for going through each number from the start value all the way to the end value. It begins by setting Num1 to start, and every time it finishes checking a number, it increases Num1 by 1. This process continues until Num1 becomes greater than end.

You can think of the outer loop as the program's "manager." Its job is to make sure that every single number in the given range is checked properly. Without this outer loop, the program would only check one number and stop.

To summarize the outer loop's role:

- It controls the main flow of the program.
 - It makes sure no numbers are skipped between start and end.
 - It triggers the prime-checking process for each number.
-

The Inner Loop

Inside the outer loop, there's another loop — the inner loop — which is a bit more specialized. Its job is to check if the current number (Num1) is a prime number.

The inner loop starts by setting Num2 to 2, and then it keeps increasing Num2 by 1 each time, checking whether Num1 is divisible by Num2. It keeps doing this until Num2 becomes greater than the square root of Num1. This design is very smart because, as explained earlier, it's unnecessary to check divisibility beyond the square root.

If during any of these checks the program finds that Num1 is divisible by Num2, it immediately marks Num1 as not prime by setting is_Correct to False. Otherwise, if it survives all the checks, Num1 is declared prime.

So, the inner loop's role is:

- *To determine whether the current number is prime.*
- *To efficiently check divisibility only up to the square root of the number.*
- *To set a Boolean flag (is_Correct) based on whether the number has any divisors.*

How the Two Loops Work Together

Together, the outer and inner loops create a full checking system.

The outer loop focuses on which numbers to check, and the inner loop focuses on how to check if each number is prime.

Without the outer loop, we wouldn't move through the range of numbers. Without the inner loop, we wouldn't be able to check if a number is prime correctly.

They're like two gears in a machine, each turning the other to complete the whole task.

2a. Explain how the termination conditions of the inner loop affect the outer loop's processing. What happens when the inner loop exits early?

In the flowchart, the inner loop plays a critical role in determining whether a number is prime. It checks if the current number (Num1) can be divided evenly by any number (Num2) between 2 and the square root of Num1.

Now, the way the inner loop ends (its termination) directly affects what happens next in the outer loop.

There are two main ways the inner loop can terminate:

- 1. It finishes checking all possible divisors up to the square root without finding any divisor.*
- 2. It finds a divisor early and exits the loop immediately.*

When the Inner Loop Exits Early

If the inner loop finds a divisor early — meaning $\text{Num1} \% \text{Num2} == 0$ — it sets the Boolean variable `is_Correct` to `False`.

At this point, the number is clearly not prime, and there's no need to keep checking. So the program can skip the rest of the checks and move forward.

In practice, this early termination is very important because:

- *It saves time — the program doesn't waste effort checking unnecessary divisors.*
- *It makes the program faster, especially when `Num1` is a composite number with small factors (like 4, 6, 8, 9, etc.).*
- *It allows the outer loop to quickly move on to the next number without delay.*

In other words, early termination of the inner loop speeds up the overall execution of the outer loop by letting the program quickly determine that a number isn't prime and jump back to processing the next candidate.

When the Inner Loop Completes Normally

On the other hand, if the inner loop runs all the way through without finding a divisor, it means `Num1` survived all tests — no divisors were found.

In this case, the Boolean `is_Correct` stays `True`, and the number is considered prime. The program then adds it to the `Values` list.

After that, just like before, the outer loop increases `Num1` and continues checking the next number in the range.

2b. Identify the Boolean variable used in the flowchart and what is the role of this Boolean variable?

In the flowchart, there is one key Boolean variable used throughout the process, and that variable is called `is_Correct`.

A Boolean variable is a special kind of variable in programming that can only have one of two values: True or False. It's perfect for situations where you need to track yes-or-no conditions, such as whether a number is prime or not.

What Does `is_Correct` Do?

In this specific flowchart, the `is_Correct` variable is used to keep track of whether the current number (Num1) is a prime number.

Here's how it works step by step:

1. At the start of checking each number (Num1), `is_Correct` is set to True. This means the program is assuming the number is prime — unless it finds a reason to say otherwise.
2. The program then checks whether Num1 can be divided evenly by any number between 2 and the square root of Num1.
3. If a divisor is found (i.e., $\text{Num1} \% \text{Num2} == 0$), then `is_Correct` is set to False — indicating the number is not prime.
4. After all the checks, the program looks at the value of `is_Correct`:
 - If it's still True, the number is added to the list of primes.
 - If it's False, the number is skipped.

So, in short, `is_Correct` acts like a flag that tracks the result of the prime number check.

Why Is This Boolean Variable Important?

Without `is_Correct`, the program would have no simple way to remember whether a number passed all the tests or not. You could write the logic differently (e.g., using break statements or more complicated conditions), but having this Boolean variable makes the logic cleaner and easier to understand.

It also allows the inner loop and the outer loop to work together more efficiently. The inner loop sets `is_Correct`, and the outer loop checks its final value to decide whether to store the number.

It's also a great example of how Boolean variables are used in real programming:

- To keep track of a condition over time
- To control what happens next based on that condition

3. Consider the following

```
If Num1 > 1  
if Num1 % Num2 == 0  
if Is_Correct==True
```

a.Explain the purpose of a conditional statement in programming.

In programming, a conditional statement is used to make decisions. It allows the program to choose between different paths depending on whether a certain condition is True or False.

You can think of it like a traffic light:

- If the light is green → go
- If it's red → stop
- If it's yellow → slow down

Similarly, in code:

```
if condition:  
    do_this()  
else:  
    do_something_else()
```

These "if" or "if-else" statements are how a program decides what to do based on specific conditions.

The main purpose is to add flexibility and logic to your code. Without conditionals, a program would just run every line from top to bottom — with no ability to respond to user input, calculations, or checks.

Conditional statements let your program:

- *Make choices*
- *Validate data*
- *Handle errors*
- *React to different scenarios*

They're fundamental in any meaningful program.

b. Explain each of the above three conditional statements and discuss how these conditional statements contribute to the results of the program.

1. `if Num1 > 1`

This is the very first check in the program — and a simple but crucial one. The reason it's there is that prime numbers are only defined for numbers greater than 1.

- *0 and 1 are not considered prime numbers.*
- *So, the program skips them right away.*

By including this condition, the program avoids wasting time checking numbers that are automatically invalid. It's an example of input validation, and it ensures that the prime-checking logic only runs on numbers that have a chance of being prime.

If the condition fails (i.e., Num1 is 0 or 1), the rest of the checks are skipped for that number.

2. `if Num1 % Num2 == 0`

This is the heart of the primality test.

It checks whether Num1 is divisible by Num2 without a remainder — which would mean Num1 is not prime.

For example:

- *`10 % 2 == 0` → 10 is divisible by 2 → not prime*
- *`11 % 2 != 0` → keep checking*

If this condition is true, the program knows immediately that Num1 is not a prime number. It sets the Boolean variable `is_Correct` to `False`, which later prevents that number from being added to the list.

This conditional save time and allows the program to make a quick decision instead of continuing unnecessary checks.

3. `if is_Correct == True`

This is the final checkpoint for each number.

After checking all possible divisors (or finding one early), the program uses this condition to decide whether the number should be saved.

If `is_Correct` is still True, it means:

- No divisors were found
- The number is a prime
- It should be added to the list of results

This condition brings everything together — it ties the logic from the inner loop back into the main flow and tells the program what to do next.

4a. Write a Python program named; `generate_special_numbers` that implements the logic described in the flowchart. The program should:
A. Accept two inputs, start and end which define the range of numbers to be processed. B. Implement the logic to filter numbers according to the criteria described in the flow chart C. Print a list of numbers that meet the criteria.

Below is a Python program that mirrors the flowchart's logic to find prime numbers between a given start and end value. We'll follow the same step-by-step flow used in the chart, including use of a Boolean flag, nested loops, and conditions.

```

import math

def generate_special_numbers(start, end):
    values = []

    # Loop through each number in the range
    for num1 in range(start, end + 1):
        if num1 > 1:
            is_Correct = True # Assume the number is prime

            # Only check up to the square root of num1 for efficiency
            for num2 in range(2, int(math.sqrt(num1)) + 1):
                if num1 % num2 == 0:
                    is_Correct = False
                    break # Exit early if a divisor is found

            # After all checks, if number is still marked as prime, save it
            if is_Correct == True:
                values.append(num1)

    return values

```

How the Program Matches the Flowchart

Flowchart Step	Python Code Equivalent
Start and input Start/End	Function inputs start, end
Loop through Num1	for num1 in range(start, end + 1)
Check if Num1 > 1	if num1 > 1:
Set is_Correct to True	is_Correct = True
Inner loop (Num2 check)	for num2 in range(2, int(sqrt(num1)) + 1)
If divisible: set False	if num1 % num2 == 0: is_Correct = False; break
If is_Correct == True	Append to values list

5b. Test the generate_special_numbers program with the following inputs and display the results.

Test Case 1: start = 3, end = 16

`print(generate_special_numbers(3, 16))` -
(output) [3, 5, 7, 11, 13]

Test Case 2: start = 25, end = 42

`print(generate_special_numbers(25, 42))` -
(output) [29, 31, 37, 41]

These test cases confirm that the function behaves as expected and correctly identifies prime numbers according to the logic of the original flowchart.

5. It is important to validate the generate_special_numbers program by using different test cases. Consider the following test cases and describe the results of each test case and explain how the program handles these test cases.

a. Test case 1: start = 12 and end = 22

`python`

`generate_special_numbers(12, 22)`

Expected Output:

[13, 17, 19]

Explanation:

The program checks each number from 12 to 22. Numbers like 12, 14, 15, etc., are not prime, but 13, 17, and 19 pass all checks. This test confirms that the function works correctly in a mid-range span.

b. Test Case 2: start = 30, end = 8

```
python
```

```
generate_special_numbers(30, 8)
```

Expected Output:

```
[1]
```

Explanation:

This is a reverse range (start > end). Python's range (start, end + 1) won't work if start is larger than end, so the function returns an empty list.

In real-world usage, you might want to add a condition that swaps the values or throws an error — but currently, it safely returns nothing, which is acceptable for now.

c. Test Case 3: start = -11, end = 9

```
python
```

```
generate_special_numbers(-11, 9)
```

Expected Output:

```
[2, 3, 5, 7]
```

Explanation:

This test checks negative numbers. The function correctly skips any number ≤ 1 due to if num1 > 1, so only valid primes from 2 to 9 are returned.

This shows the program gracefully ignores invalid inputs and focuses on valid ranges.

d. Test Case 4: start = 24, end = 27

```
python
```

```
generate_special_numbers(24, 27)
```

Expected Output:

```
[1]
```

Explanation:

None of the numbers in this small range are prime. The function correctly returns an empty list.

This is a useful test to confirm the function doesn't generate false positives — it's strict about the prime condition.

e. Test Case 5: start = 13, end = 13

```
python
generate_special_numbers(13, 13)
```

Expected Output:

```
[13]
```

Explanation:

This test checks how the function handles a single-number range.

13 is prime, and since it's both the start and end, the function correctly returns it in a list. This confirms that single-value inputs are handled properly.

6. Identify any optimization techniques used in the given program, `generate_special_numbers`. Explain how these optimizations affect the program's performance, particularly in terms of time complexity. Include an example to illustrate the impact of the optimization.

The main optimization in `generate_special_numbers` is how it checks for prime numbers using a reduced range of divisors.

Instead of checking every number from 2 to `Num1 - 1`, the program only checks up to the square root of `Num1` (`math.sqrt(num1)`). This drastically cuts down the number of checks required — especially for large numbers.

Why This Optimization Matters

Let's say you're checking if 97 is prime:

- *Without optimization: you'd check every number from 2 to 96*
- *With square root optimization: you only check up to ≈ 9.8 (i.e., 2 to 9)*

That's a huge saving in time, especially when scanning large ranges like from 1 to 10,000. This reduces the time complexity of the inner loop from $O(n)$ to $O(\sqrt{n})$.

Example of Impact:

Checking 10,000 numbers with full-range division checks would take a lot more processing time than with the square root method. By cutting unnecessary comparisons, the program becomes scalable, fast, and efficient — which is crucial in real-world applications like cryptography or number theory.

7. Reflect on your learning journey in this module. What specific programming concepts did you find most challenging, and how did you overcome these challenges? Provide examples of how you applied these concepts in your assignment

Throughout this module, one of the most challenging concepts for me was understanding and correctly using Boolean logic and nested loops. At first, it was confusing how a True or False value could control the outcome of an entire block of code — especially when used inside loops.

The key to overcoming this was breaking problems into smaller steps. For example, while working on the `generate_special_numbers` program, I focused on:

- Checking one number at a time
- Then one divisor at a time
- And seeing how `is_Correct` changed based on the result

Writing `print()` statements helped me trace how values changed as the program ran. Eventually, I understood how `is_Correct` acted like a flag — one that gets flipped off if any rule is broken.

Another tough concept was loop efficiency. Initially, I checked every number up to `num1 - 1`, which made my program slow. But after learning about the square root optimization, I restructured my code to be faster.

By applying these techniques in this assignment, I not only completed the task but also gained confidence in writing logical, efficient Python code. These skills will definitely help me in future programming challenges.

programming assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

83/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13



Comment 1

???

PAGE 14



Comment 2

Yes it is correct, but, it needs more explanation with examples.

PAGE 15

ALIN ASTEFANEI

83

QUESTION 1

Correctly describes the purpose of the flowchart [5] Identifies number and role of loops (outer and inner) [5]



Question 1a. and b Answered well.

10

QUESTION 2

*Explains inner loop termination and its effect on the outer loop [4]
Identifies Boolean variable [2] and explains its role [4]*



Well answered

10

QUESTION 3

General explanation of conditional statements [5] Explain Each conditional statement and its impact [15]



Well answered

20

QUESTION 4

Correctly implements the program with valid logic [20] Tests with sample inputs and shows results [5]

 Code is implemented correctly for the given function. A typed version of the test cases are provided instead of the screenshots.

18

QUESTION 5 [15]

Describes and explains five test case results [15]

 Five test cases are explained well, but the results are not supported with the screenshots.

13

QUESTION 6 [10]

Identifies optimizations and explains performance effectiveness [10]

 a more detailed explanation is expected. Lacks clarity

5

QUESTION 7 [10]

Reflection on learning with clear examples [10]

 Good insight

7