# Prime Numbers

# Contents

1. Prime Numbers ..... 2
2. Code Implementation ..... 4
3. Making code more efficient ..... 13
4. Benchmarking ..... 17
5. Figures ..... 20
6. References ..... 20

# 1. Prime Numbers

## 1.1 Prime and Composite Numbers

A prime number is a unique number whereby it is only divisible by one or itself. For example, two, three and five. It is a natural number and greater than one. Composite numbers, however, can be divided into two smaller numbers for example four is the equivalent of two times two (Weisstein n.d.).

## 1.2 Trial Division Primality

This theorem has first been conceptualized by Fibonacci in his book Liber Acaci(Fibonacci 2002). Fundamentally, this algorithm divides integer n with every integer y, starting from 2 to the root of n via enumeration. If the remainder is zero then y is considered to be the divisor. Furthermore, if the modulo is given a zero, the primality test ends.

In context to this, the algorithm goes up to the root of n which can be explained through an intuitive approach by using an example. In this exemplar, n is equal to 100 and the root of 100 is 10. Suppose a x b = 100, where both a and b represent pairs of 100. If a = b and is the square root of 100, which is 10. Furthermore, If a or b is less than 10 itself, then the other variable must be greater than the other, in this case 20 times 5 = 100. Ultimately, in respect to a x b, if the first variable (a or b) gets smaller the other variable will increase. In a more mathematical term (GB 2012):

$a \cdot b = N$ where $1 < a \leq b < N$, $1 < a \leq b < N$

then $N = ab \geq a2 \Leftrightarrow a2 \leq N \Longrightarrow a \leq N - - \sqrt{N}$

In conclusion, if n is a composite number then either a or b must be less than the sqrt(n) or equal to it. To further expand on this, composite numbers must also have a prime factor(p) where p <= sqrt(n).

## 1.3 Sieve of Eratosthenes

The ancient Greek mathematician Eratosthenes created an algorithm that is capable of finding prime numbers up to a given limit. This is called the Sieve of Eratosthenes. The process of this method starts with the first prime number 2 and starts enumerating by marking out all the composite numbers until the end of a given integer (Sorenson, 1990).

In figure 1, the first number prime number starts at 2 (1 is marked due to it not being a prime number.) From there, each second number (yellow) will be crossed off then the third number (red) will be left, and every third number eliminated, and so forth (Sorenson, 1990).

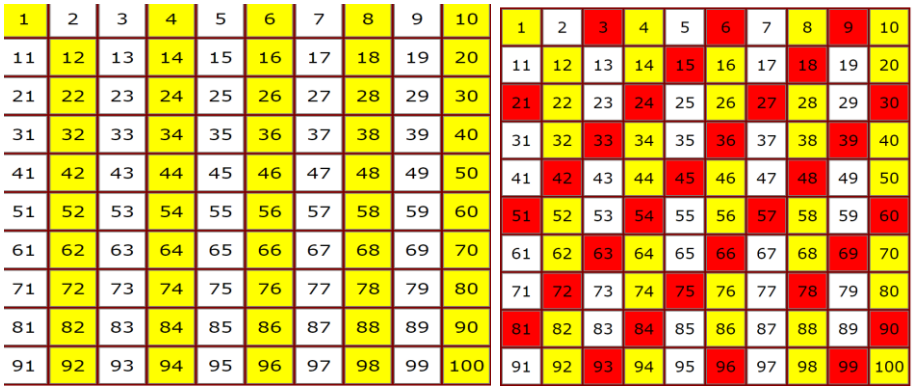


Figure 1: Sieve of Eratosthenes (Visnos n.d)

Figure 2, illustrates that once the sieve has gone through eliminating all the multiples, what is left unmarked on the grid are the remaining primes.

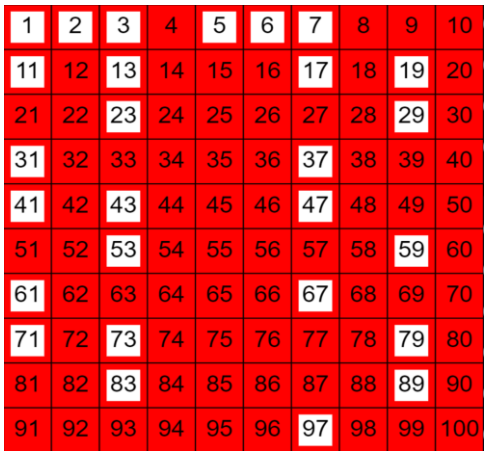


Figure 2: Sieve of Eratosthenes (Visnos n.d)

## 2. Code Implementation

### 2.1 Trial Division Code

```
import math
def trial_division_method(n):
    for n in range(1, n +1):
        if n > 1:
            for i in range (2,n):
                if n % i ==0:
                    break
            else:
                print(n)
```

Figure 3: Trial Division Code (Sublimes Text 3 n.d.)

Figure 3, shows the code above has been broken down into a flowchart will when then be further explained afterwards.

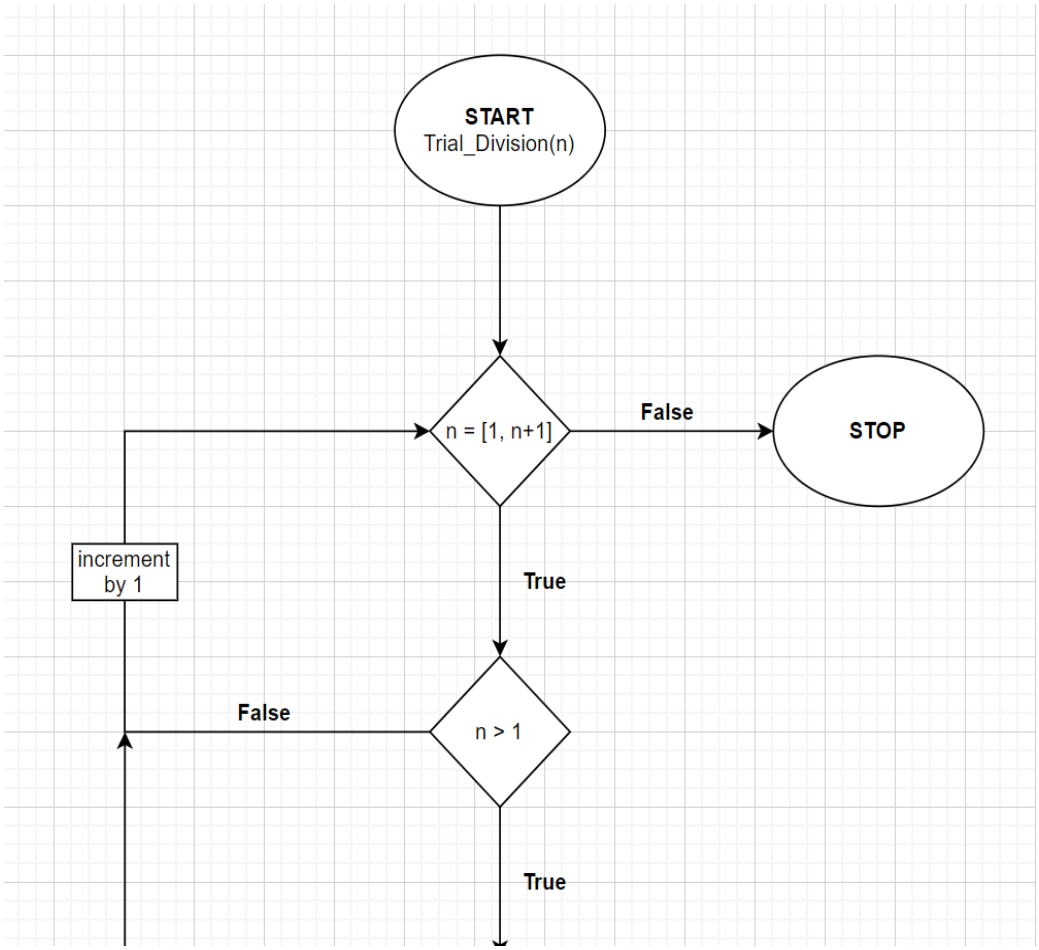


Figure 4: Trial Division Flowchart (Diagrams 2005)

2.1.1 In figure 4, The trial division method checks for the primality of any given integer. To implement this into python, a function is created for the user to provide an integer input that

is ***"deftrial_division_method."*** From there the definitive function must be written out starting with an array of numbers between 1 to the given number n. Th reasoning behind "n+1" is because the function starts at 1 and stops at n-1. In this program, it is vital for a given integer to be inclusive, for example if n = 10 then the iteration range starts from 1 and ends at 9, for 10 to be included, n needs to be 11 (n+1), in this scenario it starts from 1 and stops 10, which is the desired domain.

2.1.2 Now that a range of numbers have been imprinted in the program, the next stage is to provide a precursor before the division. According to the prime theorem, an integer prime cannot be less than 1 thereby ensuring that n is greater than 1. To translate the prime characteristic in python an if condition ie. If n>1 is used. If an integer passes this test, it will the proceed to the next line of code.

On the contrary, if n is less than or equal to 1. Then the condition will be considered to be "false". From there, it will loop back to the for loop and increment (n) by 1. This particular for loop will enumerate all the range of (1, n+1) until the next integer will not satisfy (goes over) that domain. If it does go over that range it will result in a "false" leading the entire program to stop.

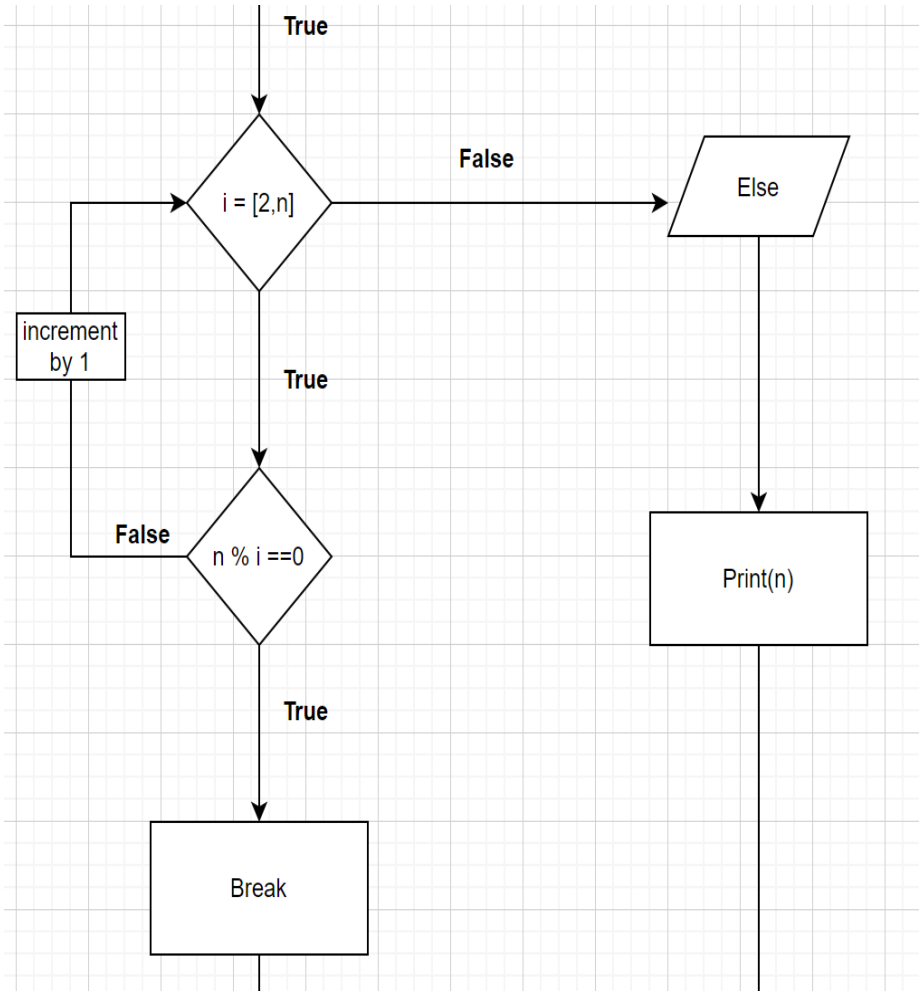


Figure 5: Trial Division Flowchart (Diagrams 2005)

2.1.3 Moving on to figure 5, now that the condition has been encoded, the introduction of the core code begins through a nested loop, iterating through all possible prime values from 2 to n, b. For this to be applied in the code, a for loop will be added to the function thus enumerating through every single number within its range until it get out of range resulting in a false. In this case it will be for i in range (2,n).

2.1.4 By having the list of all possible prime numbers, the division by comparing the diving integer to all the possible primes(i). The additional if n%i==0 condition is to find any numbers that have a remainder of 0. The purpose of checking a remainder of 0 is because the characteristics of a composite number is that of a divisor which can be broken down into smaller integers. Suppose n = 6, for i in range(2,5) , by enumerating through this range where 6%i. Continuing, i decides to increment by 1. If the result has a remainder(False), the inner loop will keep iterating until either is out of range or the conditional n%i==0 is true.. In this scenario it is it 6%2 ==0. Since this is true it will break the inner loop condition, move to the outer loop increment n by 1 for n in range (1,n+1).

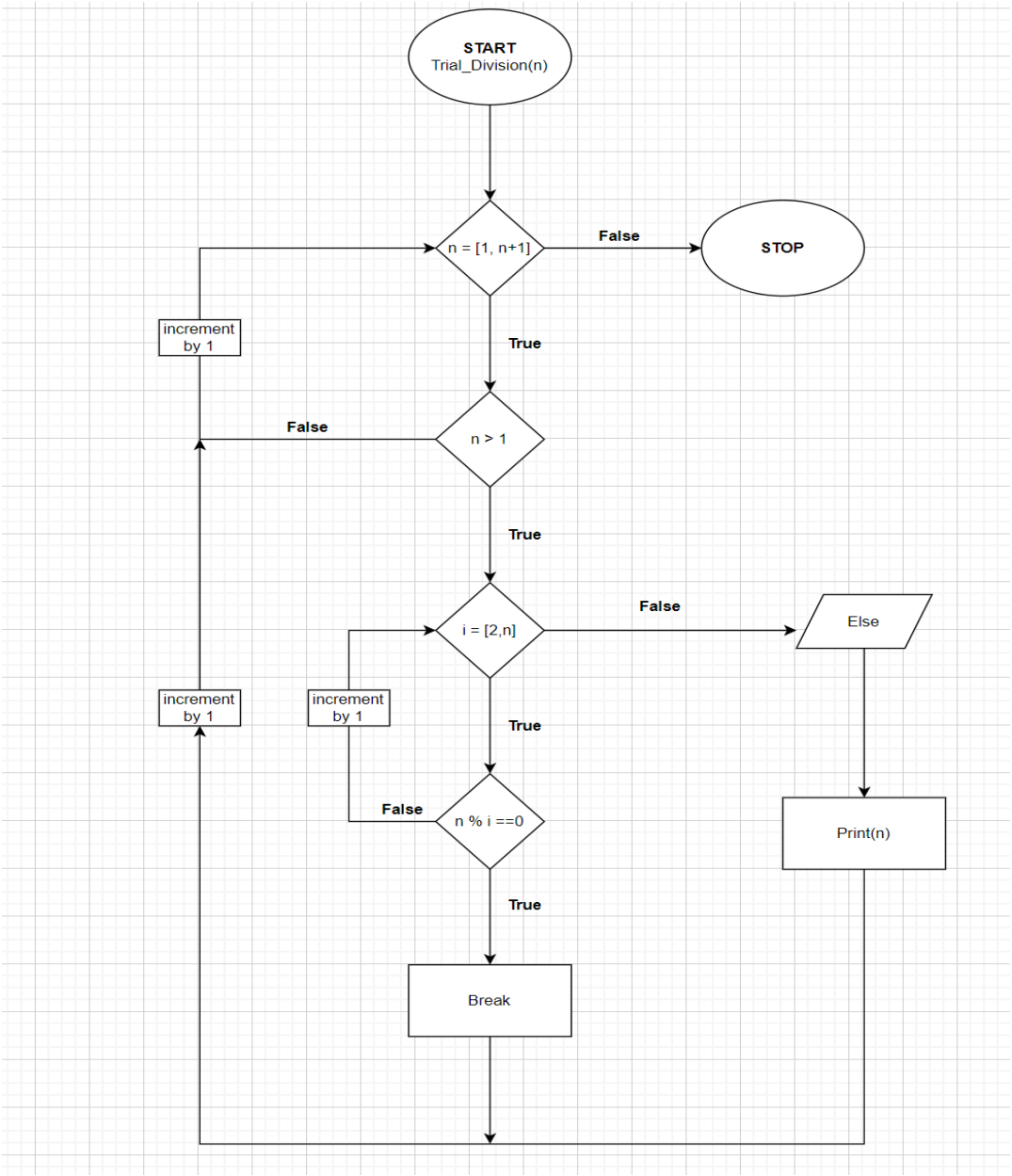

Figure 6: Trial Division Flowchart (Diagrams 2005)

2.1.5 From figure 6, if a given number has enumerated through all the possible primes(i) it will naturally increment by 1 and the condition will thus be out of range (False). Once it is false, it will proceed onto the alternative process which is the else command and print(n) the prime number. From there the cycle continues as as n is incremented out it is out of range for the program to stop.

### 2.2 Sieve of Eratosthenes

```
def sieve(n):
    primes = []
    is_prime = [False,False]+[True]*(n-1)

    for p in range(2,n+1):
        if is_prime[p]:
            primes.append(p)
            for i in range(p**2,n+1,p):
                is_prime[i]=False
```

Figure 7: Sieve of Eratosthenes Code (Sublime Text 3 n.d.)

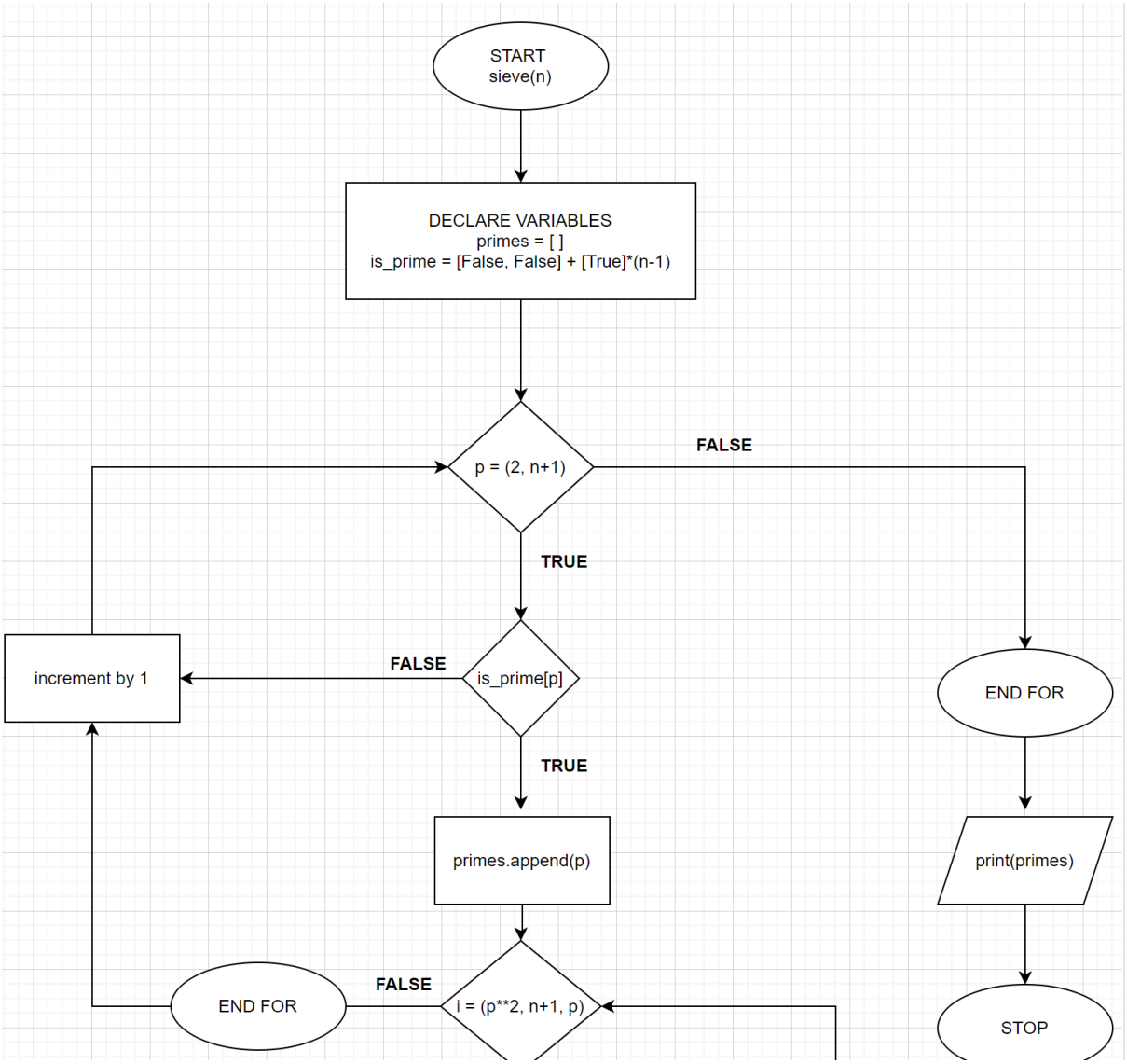


Figure 8: Sieve of Eratosthenes Flowchart (Diagrams 2005)

2.2.1 Referring to both figure 7 and 8, the start-up of the sieve algorithm is identical to the trial division method, def sieve(n). The approach to implement this algorithm is by declaring two variables. The first of which is to be an empty list, primes = [], the purpose of this declaration is to create a list of prime numbers.

The second variable is_prime= [False, False] + [True}*(n-1), which is another list of boolean values that represent integers 0 to the given integer (n-1). By having a list of numbers that are turned on as true besides 0 and 1 (since they are not prime, the numbers will be both false, as is represented by the code highlighted in blue), all the integers up to the nth are assumed as prime (true).

The reasoning behind this boolean method is to enumerate from the first number, start from it's square then falsify or "turn off", every nth numbers as non-prime numbers thus removing composite numbers.

2.2.2 After coding the pre-requisites to the sieve. The code proceeds to execute the first step of the algorithm, that is "to create a list of consecutive integers from 2 through n". Converting this into the python language, a for loop will be initiated for enumeration through a range of 2 to the given integer n +1, that is, for p in range (2, n+1). If the integer passes this condition (true) it will then move onto the next conditional step. If the condition is to be false or not in range between 2 to n+1, it will end the for loop, print the prime list (empty) and stop the algorithm.

2.2.3 The next conditional step, is to check if any boolean values are true or false. The code is_prime[p], checks the pth element within the is_prime list of true and false values. Suppose p=2, if is_prime[2] true or false, meaning the code will look at the second element or value in the is_prime list to clarify if it is true. If it is then it will append the number 2 into the primes list. However, if p is considered to be false, it will then loop back and increment p by 1 and go through the first conditional step to check whether p is in range of 2 to n+1. Once he constraint is true it will then append the integer p into the primes list. Additionally, the completes the second stage of the algorithm "let p =2, the smallest prime number"

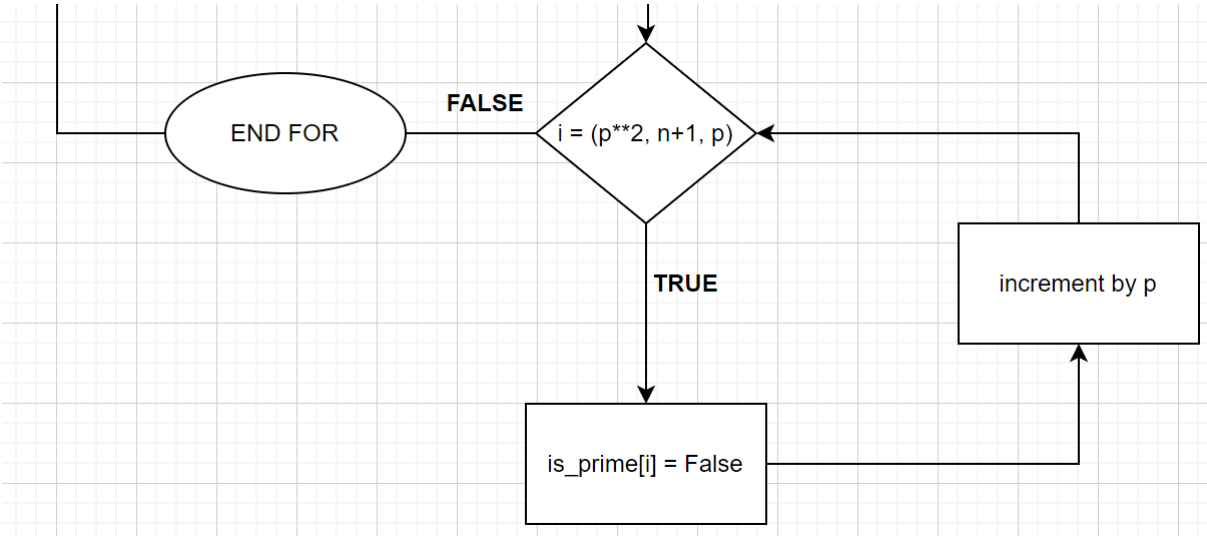


Figure 9: Sieve of Eratosthenes Flowchart (Diagrams 2005)

2.2.4 The third step in the algorithm states that the code must "Enumerate the multiples of *p* by counting in increments of *p* from 2*p* to *n*, and mark them in the list (these will be 2*p*, 3*p*, 4*p*, ...; the *p* itself should not be marked)." From figure 9, to execute this in code, a nested loop will be employed to iterate and "sieve" out all the composite integers. In the image above, enumerate for each integer i in range of p**2 to n+1, for i in range of (p**2, n+1,p), and mark the true boolean values in the is_prime list as false (is_prime[i] = False).

Finally increment by pth value until i is out of range, thus end the inner loop and proceed to the first code by incrementing p for p in range (2, n+1). This section of the code also completes the fourth step in the algorithm that is, "Find the smallest number in the list greater than *p* that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this new number (which is the next prime), and repeat from step 3."

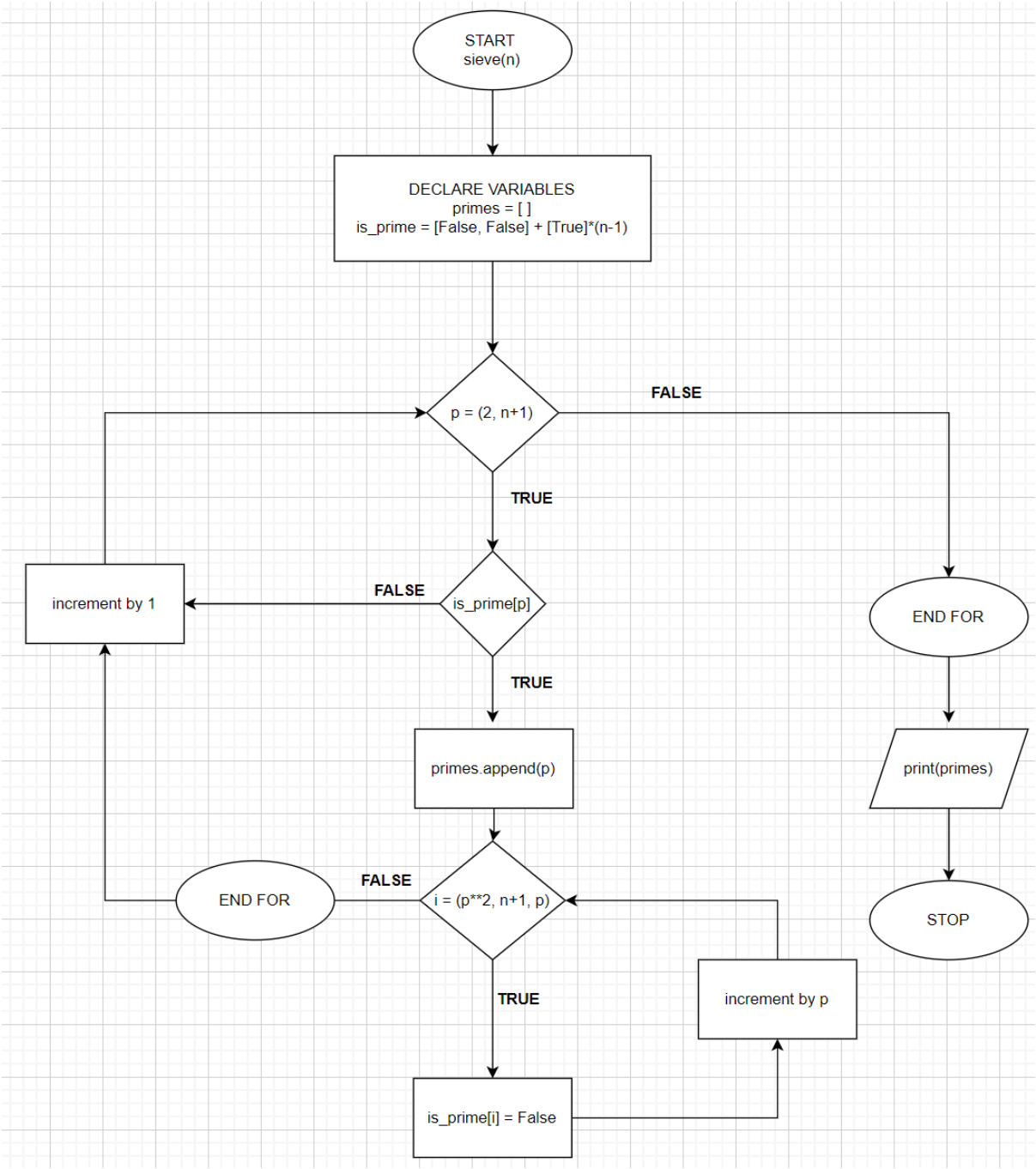


Figure 10: Optimized Sieve of Eratosthenes Flowchart (Diagrams 2005)

2.2.5 As shown in figure 10, once the active stages of the algorithm is processed. It will print out all the remaining True values, these values are considered to not be marked, resulting the integers to be prime numbers. The code print(primes), completes the fifth and final step in the sieve of Eratosthenes algorithm, "When the algorithm terminates, the numbers remaining not marked in the list are all the primes below *n*."

## 2.3 Output

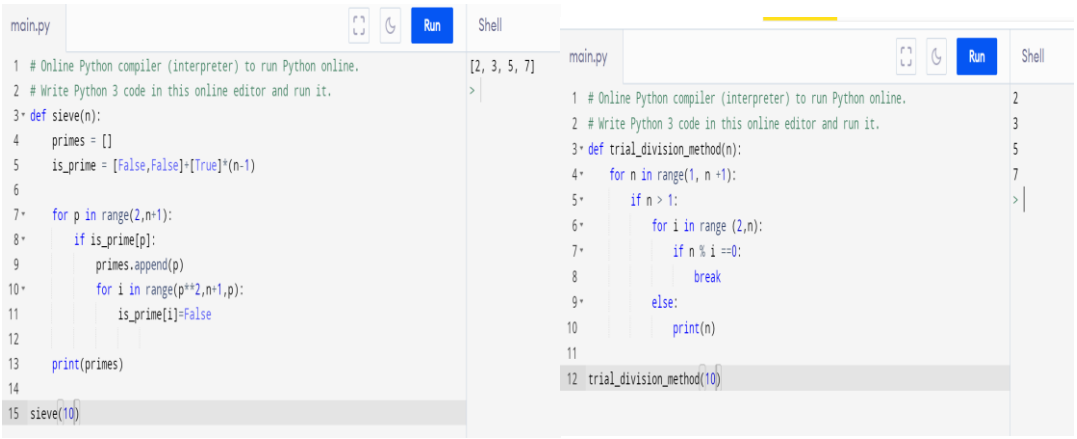


Figure 11: Code (Programiz 2021)

Using the programiz online compiler, figure 11, shows sieve algorithm on the left and trial division on the right. By testing both functions using sieve(10) and trial_division_method(10), the output shown on the right hand for both images displays the same result, which is the desired outcome that is, printing out prime numbers.

For expansion, the algorithm differs from one another. The Sieve of Eratosthenes eliminates multiples whereas the trial division algorithm test for remainders. The algorithms also differ, as the trial division method needs to check every single integer including any multiples whereas the sieve function eliminates them completely and moves onto the next smallest prime, making it more efficient.

# 3. Making code more efficient

### 3.1 Removing Dead Code

Initially in the trial division algorithm,
for n in range(1,10):
If n>1

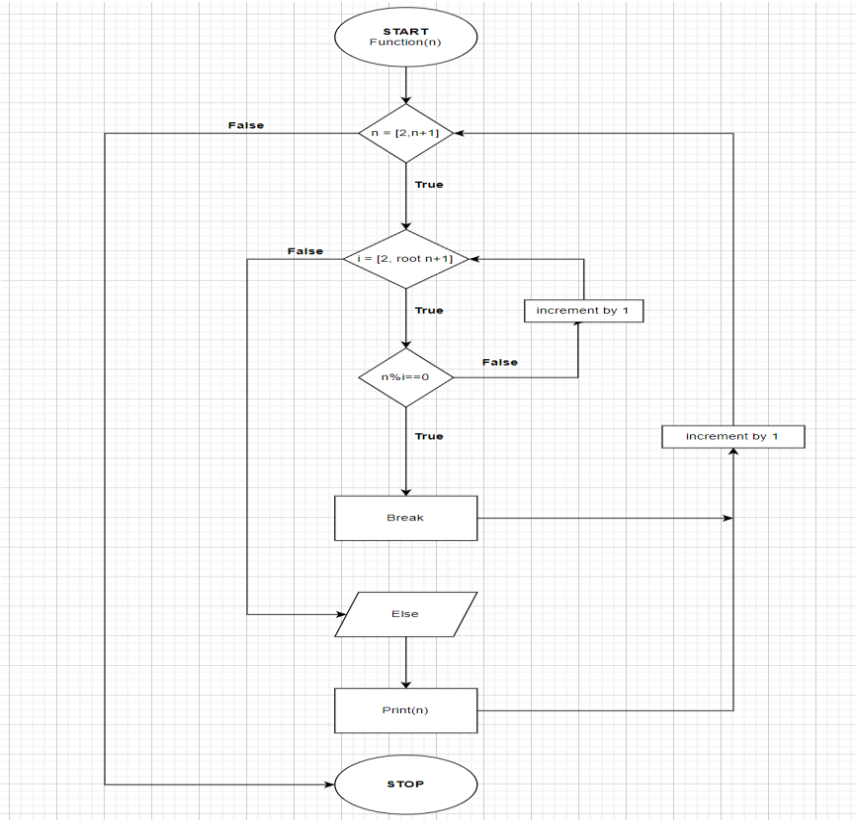


Figure 12 : Optimized Trial Division Flowchart (Diagrams 2005)
Figure 12 is used to examine whether a given integer is 0 or 1, since they are not prime numbers. However given that the lowest prime number is 2. Both lines could be eliminated and replaced with: for n in range[2,n+1]

The purpose of this, is executed more efficiently because the code does not use 0 and 1 through the program, causing these integers, redundant. The nature of conditional for loops automatically checks if a given integer is True or False. Therefore by starting at [2,n+1], it naturally checks if the number is within it's constraints, for the range of possible primes. Ultimately when both these integers, 0 and 1 are declared, the program's code will never be executed. The bottom images, clearly shows that the program has returned nothing(on the right hand side), resulting the algorithm to stop.

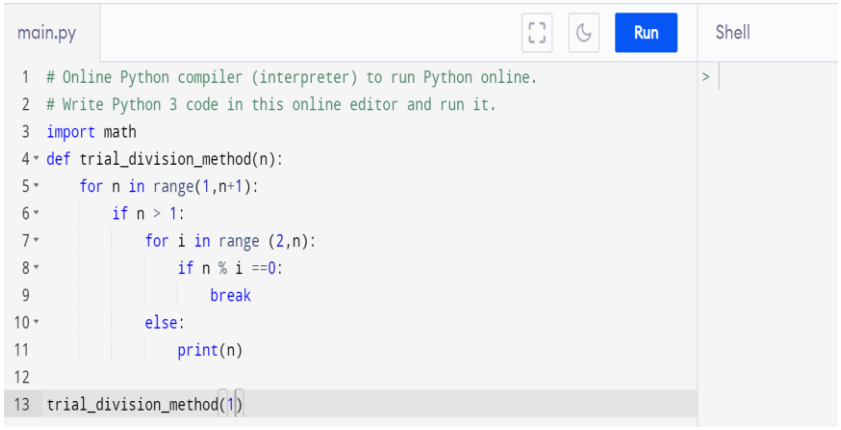


Figure 13: Trial Division Code(Programiz n.d)

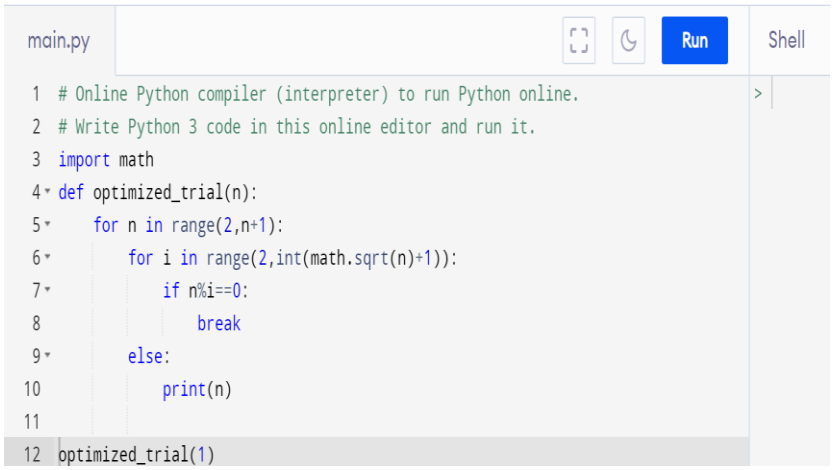


Figure 14: Optimized Trial Division Code (Programiz 2021)

Similarly, the sieve of Eratosthenes algorithm can also be reduced by removing unnecessary code.

```
if is_prime(p):
        primes.append(p)
```

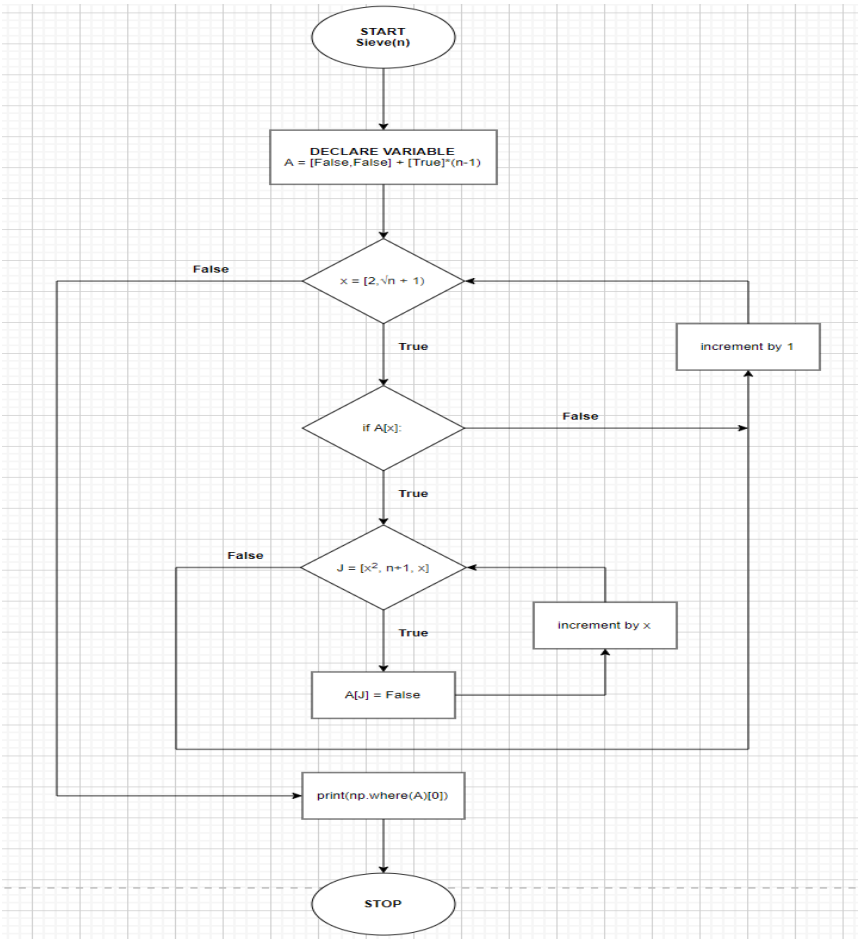


Figure 15: Opimized Sieve of Eratosthenes Flowchart (Diagrams 2005)

The table below, illustrates the concept of the declared empty list (Primes). In this situation, since there are no elements contained in the array, an output of that list will be [ ].

| Code: prime = [] | Primes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Index Values [A] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Elements | EMPTY | EMPTY | EMPTY | EMPTY | EMPTY | EMPTY | EMPTY | EMPTY |

Suppose n = 7 , leaving is_prime = [False, False] + [True]*(6). The table below, depicts the array of boolean values (is_prime).

| Code: is_prime = [False, False] + [True]*(n-1) | is_prime | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Index Values [A] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Elements | F | F | T | T | T | T | T | T |

The code primes.append(p) can be illustrated with another table below showing that the pth elements are added/updated to the empty list. The primes are now showing the indexed values of is_prime in the primes list with the condition that they are true.

| Code: primes.append(p) | Primes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Index Values [A] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Elements | 2 | 3 | 5 | 7 | EMPTY | EMPTY | EMPTY | EMPTY |

A removal of appending(p) to the primes list and declaring an empty list of variables primes = [] could be eliminated. Instead of adding true values of p to Primes, printing the indexed values of the Booleans in the variable is_primes will suffice.

A possible way of doing this, is importing the numpy module before the algorithm is called and adding the function np.where(A)[0], to return the indices. To further expand, the numpy function goes into a data structure and extracts all indices that match a logical condition. The function itself, naturally returns true values instead of false or 0.

| | is_prime | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Index Values [A]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Elements** | F | F | T | T | T | T | T | T |

The is_prime boolean list will still be declared in the algorithm, The table below ashows the updated values after the algorithm itereates through multiples. The red integers shows the indices are referenced if the np.where(is_primes) is called.Since it only outputs the true values it will thus be 2,3,5,7.

| | np.where(is_prime) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Index Values [A]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Elements** | F | F | T | T | F | T | F | T |

Coming back to the actual code, the table below depicts how the rest of the remainder instructions will exectute. The indices highlighted in yellow illustrates that the print function will print out the numbers 2,3,5,7.

| | print(np.where(is_prime)[0]) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Index Values [A]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Elements** | F | F | T | T | F | T | F | T |

Finally, in both algorithms they have not implemented the square root of n. As explained in 1.2, the use of a square root will be more efficient because it takes less steps to get to the other factors. Suppose, the algorithm wants to figure out primes up to a given number the use of the root function will filter out all the factors and numbers that are less than or equal to the square root of that integer. Any factor greater than root n will have a factor that is less than or equal to root n.

# 4. Benchmarking

## 4.1 Trial Division vs Optimised Trial Division

### 4.1.1 Trial Division

```
In [11]:
import math
import timeit
def trial_division_method(n):
    for n in range(1,n+1):
        if n > 1:
            for i in range (2,n):
                if n % i ==0:
                    break
            else:
                print(n)
```

```
In [15]:
timeit.timeit('trial_division_method(1000)', setup='from __main__ import trial_division_method', number = 100)
```

```
877
881
883
887
907
911
919
929
937
941
947
953
967
971
977
983
991
997

Out[15]: 1.918856699999992
```

Figure 16: Trial Division (Jupyter 2021)

### 4.1.2 Optimized Trial Division

```
In [8]:
import math
import timeit
def optimized_trial(n):
    for n in range(2,n+1):
        for i in range(2,int(math.sqrt(n)+1)):
            if n%i==0:
                break
        else:
            print(n)
```

```
In [13]:
timeit.timeit('optimized_trial(1000)', setup='from __main__ import optimized_trial', number = 100)
```

```
877
881
883
887
907
911
919
929
937
941
947
953
967
971
977
983
991
997

Out[13]: 1.4756213000000002
```

Figure 17: Optimized Trial Division(Jupyter 2021)

## 4.2 Sieve of Eratosthenes vs Optimised Sieve

### 4.2.1 Sieve :

```
In [35]:
import math
import timeit
import numpy as np
np.set_printoptions(threshold=np.inf)

def sieve(n):
    primes = []
    is_prime = [False,False]+[True]*(n-1)

    for p in range(2,n+1):
        if is_prime[p]:
            primes.append(p)
            for i in range(p**2,n+1,p):
                is_prime[i]=False


    print(primes)
```

```
In [42]:
timeit.timeit('sieve(10000)', setup='from __main__ import sieve', number = 100)
```

```
1, 6967, 6971, 6977, 6983, 6991, 6997, 7001, 7013, 7019, 7027, 7039, 7043, 7057, 7069, 7079, 7103, 7109, 7121, 7127, 712
9, 7151, 7159, 7177, 7187, 7193, 7207, 7211, 7213, 7219, 7229, 7237, 7243, 7247, 7253, 7283, 7297, 7307, 7309, 7321, 733
1, 7333, 7349, 7351, 7369, 7393, 7411, 7417, 7433, 7451, 7457, 7459, 7477, 7481, 7487, 7489, 7499, 7507, 7517, 7523, 752
9, 7537, 7541, 7547, 7549, 7559, 7561, 7573, 7577, 7583, 7589, 7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673, 768
1, 7687, 7691, 7699, 7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793, 7817, 7823, 7829, 7841, 7853, 7867, 787
3, 7877, 7879, 7883, 7901, 7907, 7919, 7927, 7933, 7937, 7949, 7951, 7963, 7993, 8009, 8011, 8017, 8039, 8053, 8059, 806
9, 8081, 8087, 8089, 8093, 8101, 8111, 8117, 8123, 8147, 8161, 8167, 8171, 8179, 8191, 8209, 8219, 8221, 8231, 8233, 823
7, 8243, 8263, 8269, 8273, 8287, 8291, 8293, 8297, 8311, 8317, 8329, 8353, 8363, 8369, 8377, 8387, 8389, 8419, 8423, 842
9, 8431, 8443, 8447, 8461, 8467, 8501, 8513, 8521, 8527, 8537, 8539, 8543, 8563, 8573, 8581, 8597, 8599, 8609, 8623, 862
7, 8629, 8641, 8647, 8663, 8669, 8677, 8681, 8689, 8693, 8699, 8707, 8713, 8719, 8731, 8737, 8741, 8747, 8753, 8761, 877
9, 8783, 8803, 8807, 8819, 8821, 8831, 8837, 8839, 8849, 8861, 8863, 8867, 8887, 8893, 8923, 8929, 8933, 8941, 8951, 896
3, 8969, 8971, 8999, 9001, 9007, 9011, 9013, 9029, 9041, 9043, 9049, 9059, 9067, 9091, 9103, 9109, 9127, 9133, 9137, 915
1, 9157, 9161, 9173, 9181, 9187, 9199, 9203, 9209, 9221, 9227, 9239, 9241, 9257, 9277, 9281, 9283, 9293, 9311, 9319, 932
3, 9337, 9341, 9343, 9349, 9371, 9377, 9391, 9397, 9403, 9413, 9419, 9421, 9431, 9433, 9437, 9439, 9461, 9463, 9467, 947
3, 9479, 9491, 9497, 9511, 9521, 9533, 9539, 9547, 9551, 9587, 9601, 9613, 9619, 9623, 9629, 9631, 9643, 9649, 9661, 967
7, 9679, 9689, 9697, 9719, 9721, 9733, 9739, 9743, 9749, 9767, 9769, 9781, 9787, 9791, 9803, 9811, 9817, 9829, 9833, 983
9, 9851, 9857, 9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929, 9931, 9941, 9949, 9967, 9973]

Out[42]: 0.19765060000008816
```

Figure 18: Sieve of Eratosthenes (Jupyter 2021)

### 4.2.2Optimized Sieve

```
In [30]:
import math
import timeit
import numpy as np
np.set_printoptions(threshold=np.inf)

def sieve_opt(n):
    A = [False, False]+ [True]*(n-1)
    for x in range (2, int(math.sqrt(n)+1)):
        if A[x]:
            for J in range(x**2, n+1, x):
                A[J] = False
    prime = np.where(A)[0]
    print(prime)
```

```
In [42]:
timeit.timeit('sieve_opt(10000)', setup='from __main__ import sieve_opt', number = 100)
```

```
7877 7879 7883 7901 7907 7919 7927 7933 7937 7949 7951 7963 7993 8009
8011 8017 8039 8053 8059 8069 8081 8087 8089 8093 8101 8111 8117 8123
8147 8161 8167 8171 8179 8191 8209 8219 8221 8231 8233 8237 8243 8263
8269 8273 8287 8291 8293 8297 8311 8317 8329 8353 8363 8369 8377 8387
8389 8419 8423 8429 8431 8443 8447 8461 8467 8501 8513 8521 8527 8537
8539 8543 8563 8573 8581 8597 8599 8609 8623 8627 8629 8641 8647 8663
8669 8677 8681 8689 8693 8699 8707 8713 8719 8731 8737 8741 8747 8753
8761 8779 8783 8803 8807 8819 8821 8831 8837 8839 8849 8861 8863 8867
8887 8893 8923 8929 8933 8941 8951 8963 8969 8971 8999 9001 9007 9011
9013 9029 9041 9043 9049 9059 9067 9091 9103 9109 9127 9133 9137 9151
9157 9161 9173 9181 9187 9199 9203 9209 9221 9227 9239 9241 9257 9277
9281 9283 9293 9311 9319 9323 9337 9341 9343 9349 9371 9377 9391 9397
9403 9413 9419 9421 9431 9433 9437 9439 9461 9463 9467 9473 9479 9491
9497 9511 9521 9533 9539 9547 9551 9587 9601 9613 9619 9623 9629 9631
9643 9649 9661 9677 9679 9689 9697 9719 9721 9733 9739 9743 9749 9767
9769 9781 9787 9791 9803 9811 9817 9829 9833 9839 9851 9857 9859 9871
9883 9887 9901 9907 9923 9929 9931 9941 9949 9967 9973]

Out[42]: 0.28804159999981493
```

Figure 19: Opimized Sieve of Eratosthenes (Jupyter 2021)

### 4.3 Findings

From the remaining figures above, the trial division optimization has worked as expected as it was 0.44 microseconds faster than the initial method. However, for the Sieve of Eratosthenes algorithm, although optimized, was actually slower than the initial one by 0.09 microseconds. A possible factor for making the optimized Sieve of Eratosthenes to be slower than the previous one, may be due to the addition of a numpy module, thus when the numpy function is in use it may cause delayed time for computer processor to run the algorithm.

## 5. Figures

a) Figures 1,2: Visnos 2011, *Sieve of Eratosthenes*, Michael Mcdaid, viewed 21st of June 2021, <https://www.visnos.com/demos/sieve-of-eratosthenes>

b) Figures 3, 7 Skinner, J. 2008, *Sublime Text*, software, version 3, Sublime HQ viewed 27 of July 2021, <https://www.sublimetext.com/3>

c) Figures 4,5,6,8,9,10,12,15 Benson, D. & Alder, G. 2005, *Diagrams.net*, software, version 14.9.2, Jgraph, viewed 27 of July 2021, <https://app.diagrams.net/>

d) Figures 11, 13, 14 Bhatta,R. n.d., *Python Programming*, software, version 3.2, Programiz, viewed 27 of July 2021, <https://www.programiz.com/python-programming/online-compiler/>

e) Figures 16,17,18,19 Fernando, P & Granger, B. 2015, *Jupyter Notebook*, software, version 3.0, viewed 27 of July 2021, <https://jupyter.org/>

## 6. References


a) Fibonacci, L., & Sigler, L. E. (2002). *Fibonacci's Liber abaci: a translation into modern English of Leonardo Pisano's Book of calculation*. Springer, New York.

b) GB 2012, *Prime or not: Determining primes through square roots,* Math and Multimedia, viewed 27 of July 2021, <http://mathandmultimedia.com/2012/06/02/determining-primes-through-square-root/>

c) Weisstein, E.W. n.d., *Prime Number*, website, Wolfram, viewed 27 of July 2021, <https://mathworld.wolfram.com/PrimeNumber.html>