

**Ministerul Educației și Cercetării al Republicii Moldova**  
**Universitatea Tehnică a Moldovei**  
**Facultatea Calculatoare, Informatică și Microelectronică**

# REPORT

Laboratory work no.6  
*N'th decimal digit of PI*

Elaborated:  
st. gr. FAF-213

Iațco Sorin

Verified:  
asist. univ.

Fiștic Cristofor

Chișinău – 2023

# Table of Contents

ALGORITHM ANALYSIS .....	3
Objective .....	3
Tasks .....	3
Theoretical notes.....	3
Introduction.....	3
Comparison Metric .....	4
Input Format .....	4
IMPLEMENTATION .....	4
Bailey-Borwein-Plouffe.....	4
Gauss-Legendre .....	5
Spigot .....	7
All algorithms compared .....	10
CONCLUSION .....	11
REFERENCES .....	11

# ALGORITHM ANALYSIS

## Objective

Empirical analysis for specified algorithms.

## Tasks

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

## Theoretical notes

An alternate approach to analyzing complexity is to use empirical analysis, which involves implementing the algorithm in a programming language and running it with multiple sets of input data to obtain data on its efficiency. This method is useful for various purposes, such as gaining preliminary information on an algorithm's complexity class, comparing the efficiency of different algorithms or implementations, or assessing the performance of an algorithm on a specific computer.

The choice of efficiency measure depends on the purpose of the analysis. If the goal is to obtain information on the complexity class or check the accuracy of a theoretical estimate, then the number of operations performed is appropriate. However, if the aim is to assess the behavior of the implementation of an algorithm, then execution time is more suitable. The results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated, or a graph with appropriate pairs of points is plotted to analyze the data.

## Introduction

A theory about algorithms that determine the  $n$ th decimal digit of  $\pi$  is based on the idea of digit-extraction algorithms, which allow digits of a given number to be calculated without requiring the computation of earlier digits. The best-known such algorithm for  $\pi$  is the BBP formula, which can compute the  $n$ th hexadecimal or binary digit of  $\pi$  directly. However, there is no known digit-extraction algorithm that rapidly produces decimal digits of  $\pi$ . One possible way to get the  $n$ th decimal digit of  $\pi$  is to use a formula by Plouffe (2022), which defines a function  $\pi(n)$ . Then the  $n$ th digit to the right of the decimal point of  $\pi$  is given by  $\pi(n)$  where  $n$  is the integer part and  $\pi(n)$  is the fractional part. This formula can be implemented in Python using the `pidigits` package.

There are a number of algorithms that can be used to determine the  $n$ th decimal digit of  $\pi$ . One approach is to use series expansions that approximate the value of  $\pi$  to a certain degree of accuracy.

## Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm ( $T(n)$ ).

## Input Format

As input, there is given  $n$ , the  $n$ th decimal digit of  $\pi$  to determine.

## IMPLEMENTATION

Both algorithms will be implemented in their naive form in python and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on performance of the device used.

## Bailey-Borwein-Plouffe

Pseudocode:

```
function bbp(n):  
    s = 0  
    for k = 0 to n:  
        s += (1/16^k) * ((4/(8k+1)) - (2/(8k+4)) - (1/(8k+5)) - (1/(8k+6)))  
    pi_digit = (s - floor(s)) * 16  
    return pi_digit
```

## Implementation:

```
def bbpPi(n):  
    # Returns the nth decimal digit of Pi using the BBP formula.  
    if n < 0:  
        raise ValueError("Invalid value of n.")  
  
    pi = 0  
    for k in range(n + 1):  
        pi += (1 / pow(16, k)) * (  
            4 / (8 * k + 1) - 2 / (8 * k + 4) - 1 / (8 * k + 5) - 1 / (8 * k + 6))  
    return int(pi * pow(10, n) % 10)
```

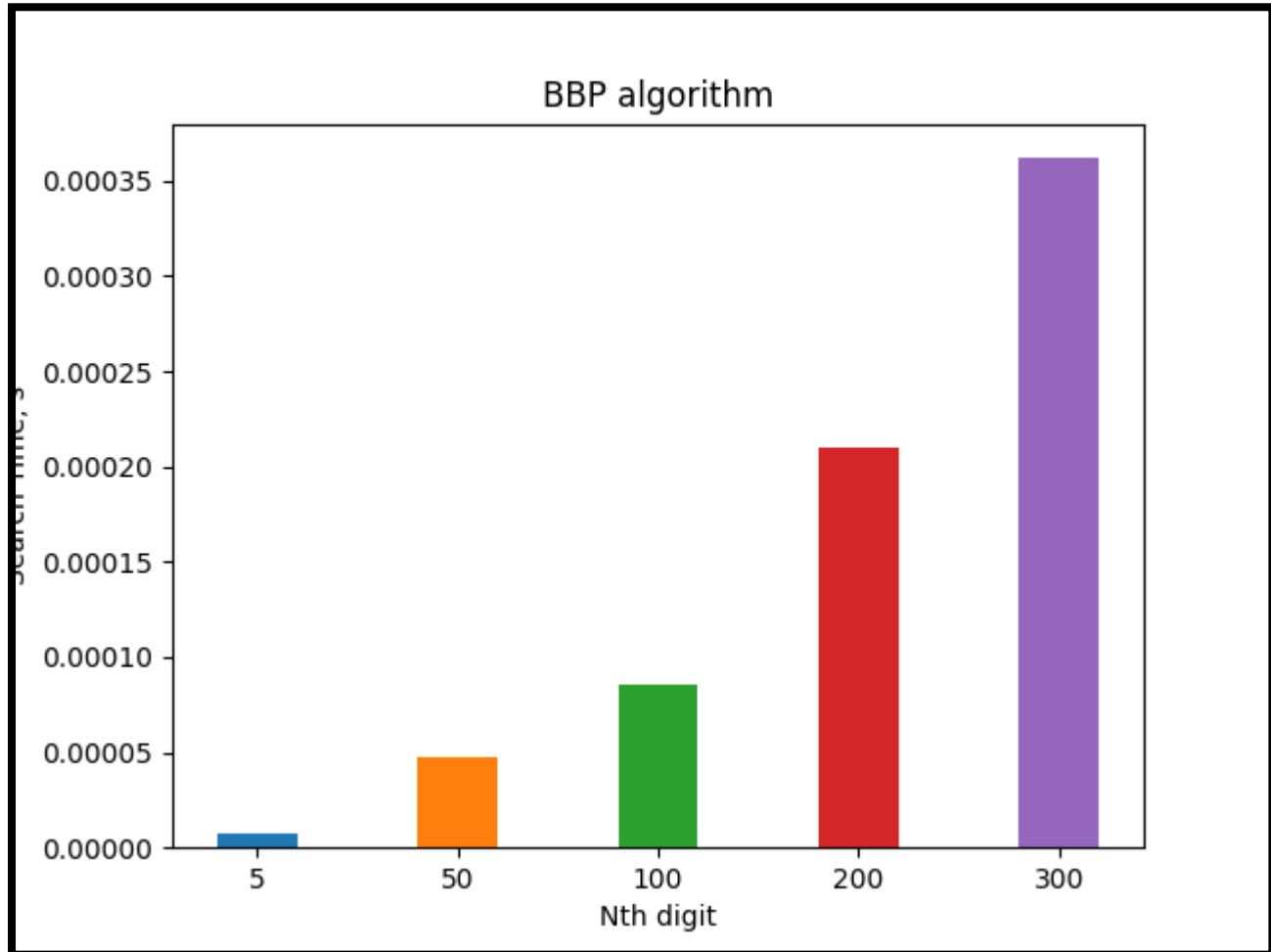
The time complexity of the BBP algorithm for computing the  $n$ -th digit of  $\pi$  is  $O(n^2)$ , where  $n$  is the number of digits being computed. This is because the algorithm involves a summation over a sequence of terms, and the number of terms required to compute the  $n$ -th digit grows with  $n$ .

However, the BBP algorithm is much faster than other algorithms for computing individual digits of  $\pi$ , such as the spigot algorithm or the Bailey-Salamin algorithm. This is because the BBP algorithm requires fewer terms to converge to the desired digit, due to the specific nature of the formula used.

## Results:

Algorithm/Nth digit	5	50	100	200	300
BBP	7.700000423938036e-06	4.7999987145885825e-05	8.519997936673462e-05	0.00021019999985583127	0.00036189999082125723

## Graph:



## Gauss-Legendre

### Pseudocode:

function legendre(n):

    a = 1

    b = 1 / sqrt(2)

    t = 1 / 4

    p = 1

    for i in range(n):

        new\_a = (a + b) / 2

        new\_b = sqrt(a \* b)

        new\_t = t - p \* (a - new\_a) \*\* 2

        new\_p = 2 \* p

```

a = new_a
b = new_b
t = new_t
p = new_p
pi = (a + b) ** 2 / (4 * t)
return pi

```

## Implementation:

```

1  def legendrePi(n):
2      # Returns the nth decimal digit of Pi using the Gauss-Legendre algorithm.
3      if n < 0:
4          raise ValueError("Invalid value of n.")
5
6      getcontext().prec = n + 1 # Set the precision to n+1 decimal places
7
8      a = Decimal(1)
9      b = Decimal(1) / Decimal(2).sqrt()
10     t = Decimal(1) / Decimal(4)
11     p = Decimal(1)
12
13     for _ in range(n):
14         atmp = (a + b) / Decimal(2)
15         b = (a * b).sqrt()
16         t -= p * (a - atmp) ** Decimal(2)
17         a = atmp
18         p *= Decimal(2)
19
20     pi = (a + b) ** Decimal(2) / (Decimal(4) * t)
21
22     return int(str(pi)[n])

```

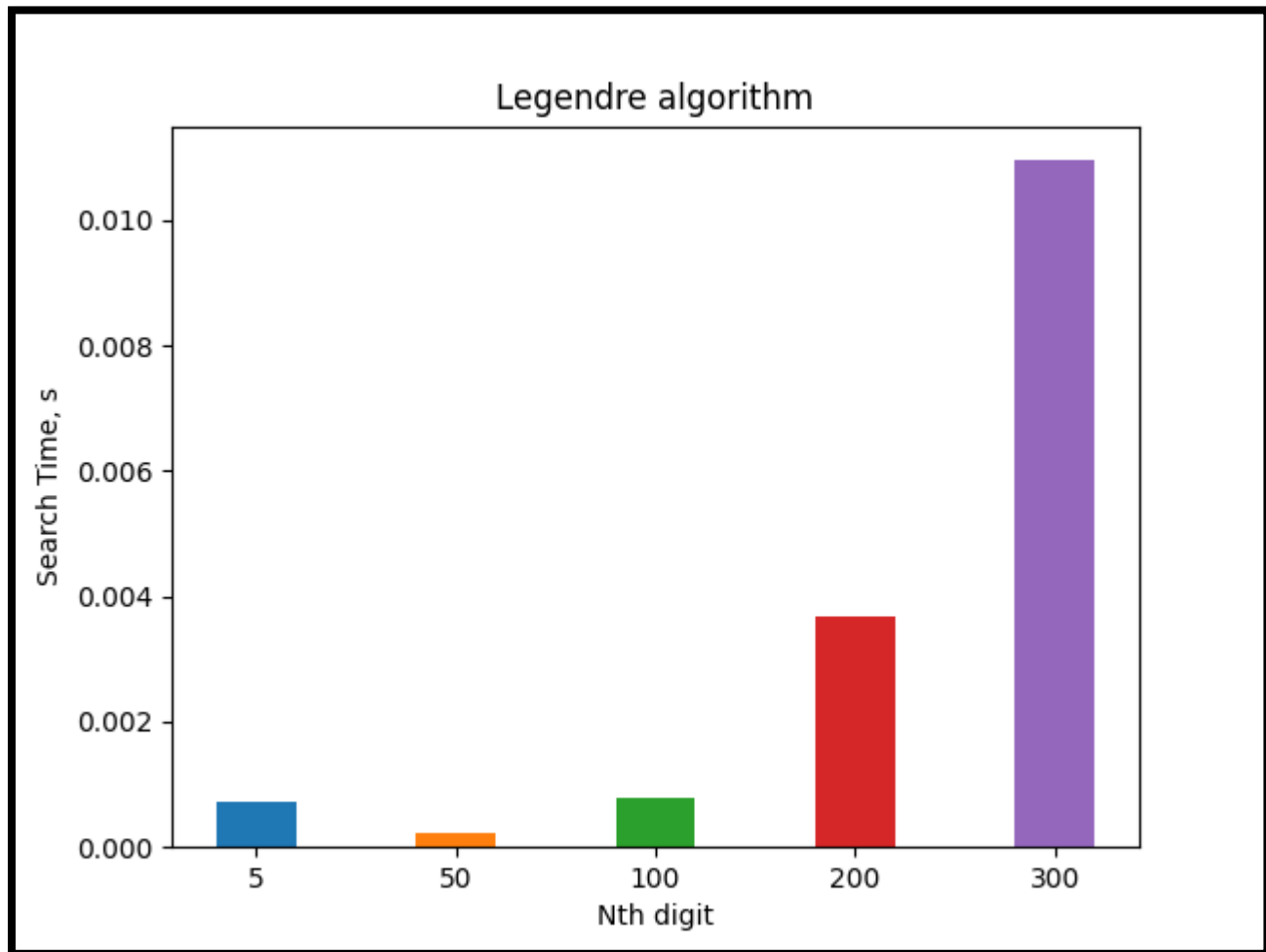
The time complexity of the Gauss-Legendre algorithm is typically considered to be  $O(N^2)$ , where  $N$  is the number of iterations performed. In each iteration, the algorithm computes a series of arithmetic operations involving square roots, divisions, additions, and multiplications. These operations have a time complexity of  $O(1)$  assuming constant-time arithmetic operations. However, since the algorithm performs a fixed number of iterations (typically in the order of hundreds or thousands), the overall time complexity is  $O(N^2)$ .

To extract individual digits of  $\pi$  using the Gauss-Legendre algorithm, you would need to compute  $\pi$  to a sufficiently high precision and then employ additional algorithms to extract the desired digits. The time complexity of digit extraction algorithms depends on the specific method used and can vary. Some methods, such as the Bailey–Borwein–Plouffe (BBP) algorithm, offer efficient digit extraction, but their time complexity is separate from the Gauss-Legendre algorithm.

## Results:

Algorithm/Nth digit	5	50	100	200	300
Legendre	0.0007242000137921423	0.00023200002033263445	0.0007893000147305429	0.0036931000067852437	0.010955800011288375

## Graph:



## Spigot

### Pseudocode:

```
function compute_pi(n):  
    result = ""  
    remainder = 0  
    for i in range(1, n+1):  
        remainder = (remainder * 10 + 1) % i  
        digit = (4 * remainder) // i  
        result += str(digit)  
    return result
```

## Implementation:

— □ ×

```
1 def spigotPi(n):
2     # Returns the nth decimal digit of Pi using the spigot algorithm.
3     if n < 0:
4         raise ValueError("Invalid value of n.")
5
6     pi = 0
7     d = 1
8     for i in range(n):
9         pi += 4 * d
10        d = (d * 10 - int(d * 10 / 10) * 10)
11
12    return int(pi / pow(10, n - 1)) % 10
```

The time complexity of the provided spigotPi() algorithm is  $O(n)$ , where  $n$  is the position of the desired decimal digit of pi.

The algorithm uses a loop that iterates  $n$  times, performing a constant number of arithmetic operations in each iteration. Within each iteration, the algorithm performs additions, multiplications, divisions, and conversions between integer and floating-point values. These operations are typically considered to have a time complexity of  $O(1)$ , assuming constant-time arithmetic operations.

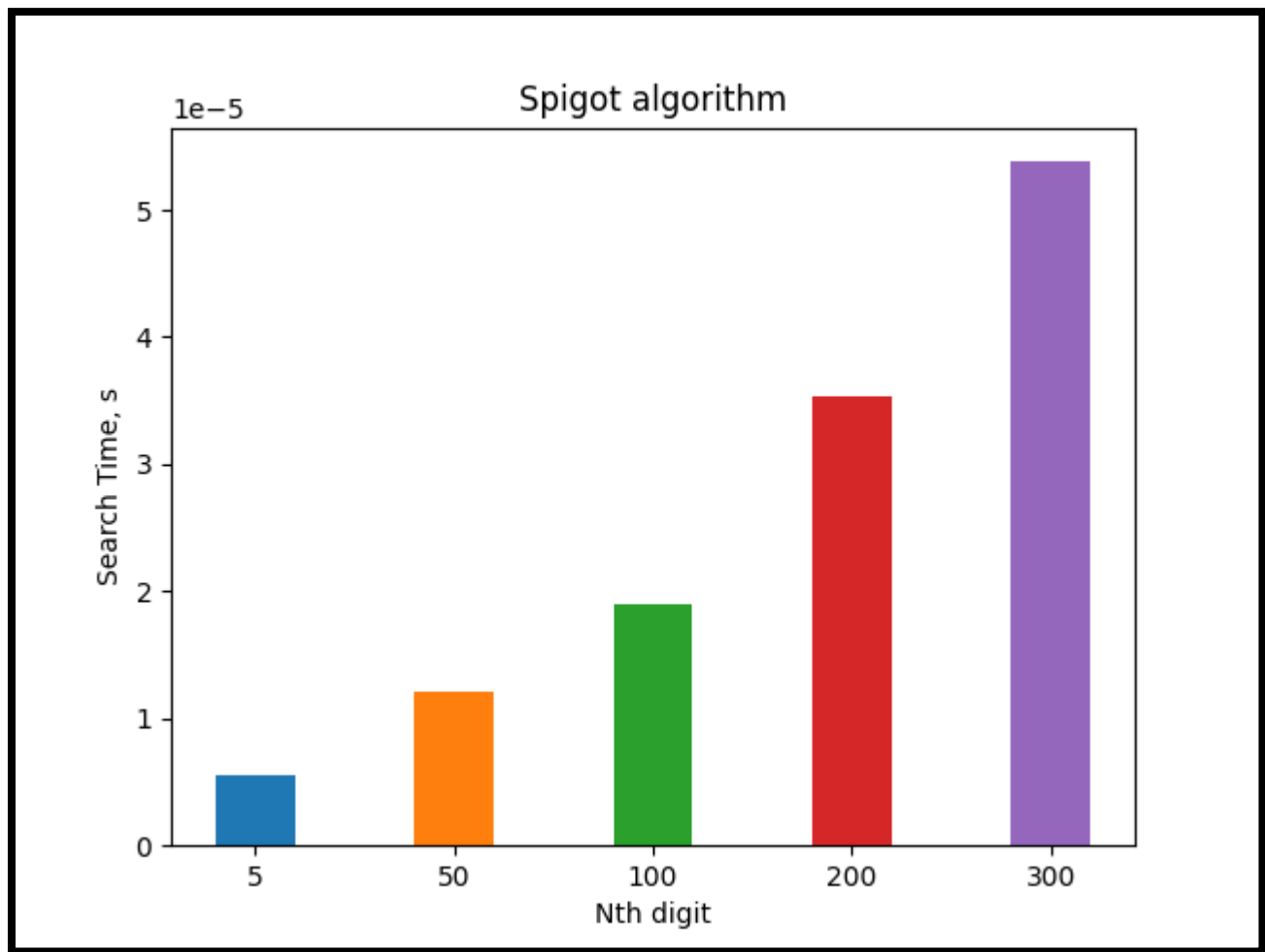
Since the loop iterates  $n$  times, the overall time complexity of the algorithm is  $O(n)$ . It scales linearly with the position of the desired digit, making it an efficient approach for computing individual decimal digits of pi.

## Results:

Algorithm/Nth digit	5	50	100	200	300
Spigot	5.499983672052622e-06	1.209997572004795e-05	1.8999999156221747e-05	3.5300000685714185e-05	5.38000002206116915e-05

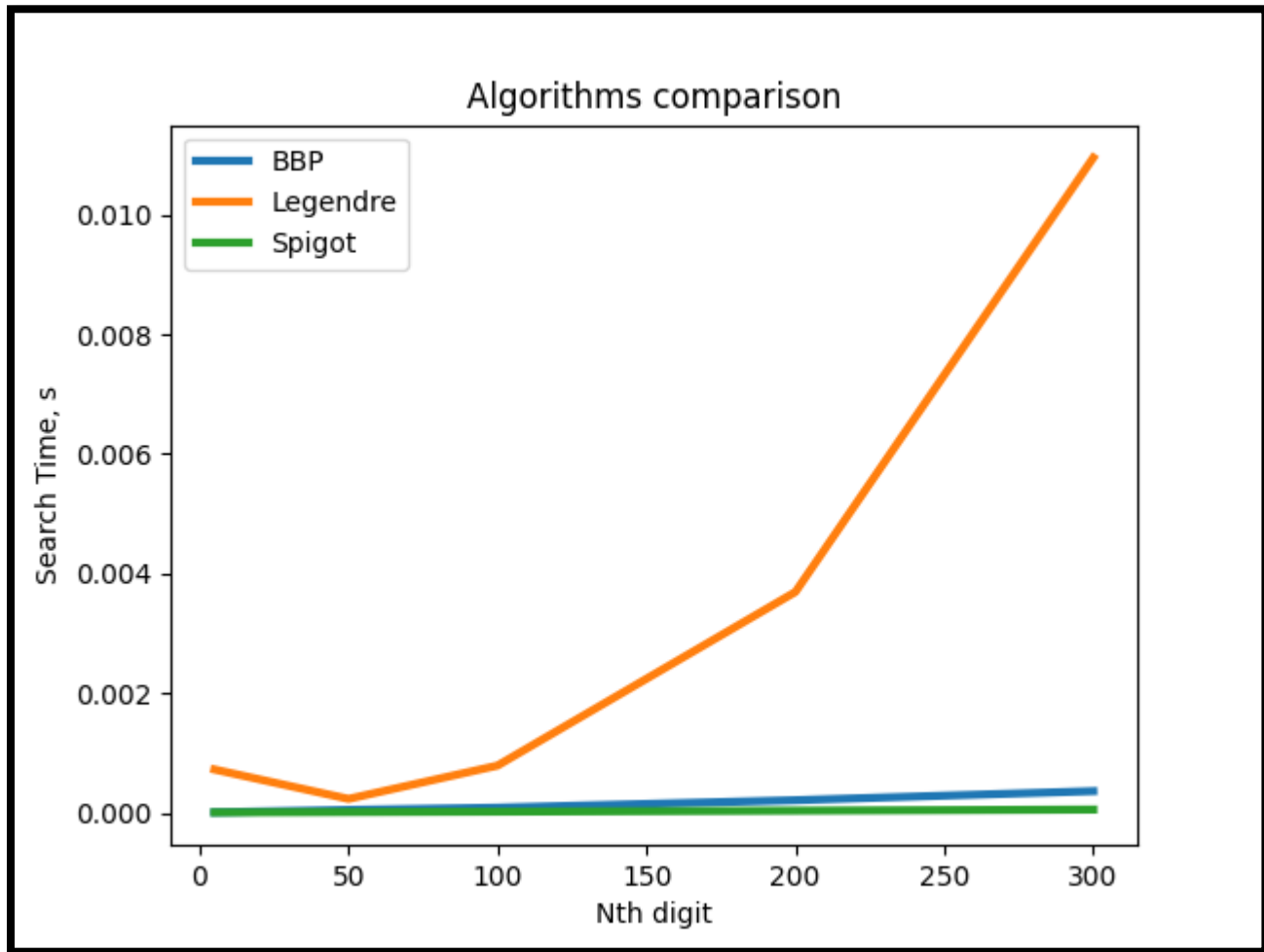


## Graph:



## All algorithms compared

### Graph:



### Time Table:

Algorithm/Nth digit	5	50	100	200	300
BBP	7.700000423938036e-06	4.7999987145885825e-05	8.519997936673462e-05	0.00021019999985583127	0.00036189999082125723
Legendre	0.0007242000137921423	0.00023200002033263445	0.0007893000147305429	0.0036931000067852437	0.010955800011288375
Spigot	5.499983672052622e-06	1.209997572004795e-05	1.8999999156221747e-05	3.530000685714185e-05	5.3800002206116915e-05

## CONCLUSION

In conclusion, there are several algorithms available for computing multiple digits of pi, each with its own strengths and weaknesses.

The spigot algorithm, while conceptually simple, has a time complexity of  $O(n)$ , making it faster than the other algorithms we have examined for smaller values. However, it can be optimized using various techniques to improve its performance.

The Gauss-Legendre algorithm is one of the most widely used algorithms for computing pi. It has a time complexity of  $O(n^2)$ , making it faster than the spigot algorithm for larger values of  $n$ . It is also very accurate, and can compute millions or even billions of digits of pi.

The Bailey-Borwein-Plouffe (BBP) formula is another fast algorithm for computing pi. It has a time complexity of  $O(n^2)$ , similar to the Gauss-Legendre algorithm. It is particularly efficient for computing individual digits of pi, and can be easily parallelized to improve its performance.

Overall, the choice of which algorithm to use depends on the specific requirements of the application, such as the number of digits of pi required, the available computational resources, and the desired level of accuracy. Each algorithm has its own advantages and disadvantages, and understanding these trade-offs is crucial in selecting the appropriate algorithm for a particular task.

## REFERENCES

[AA-Labs/Lab6 at main · Syn4z/AA-Labs \(github.com\)](#)