Ministerul Educaţiei 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

Chişinău – 2023

Table of Contents

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 nth 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 nth 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 nth decimal digit of pi is to use a formula by Plouffe (2022), which defines a function \n(1). Then the th digit to the right of the decimal point of for is given by \n(2) where is the integer part and 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 nth 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 nth decimal digit of pi to determine.

IMPLEMENTATION

Both algorithms will be implemented in their naive form in python an 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:

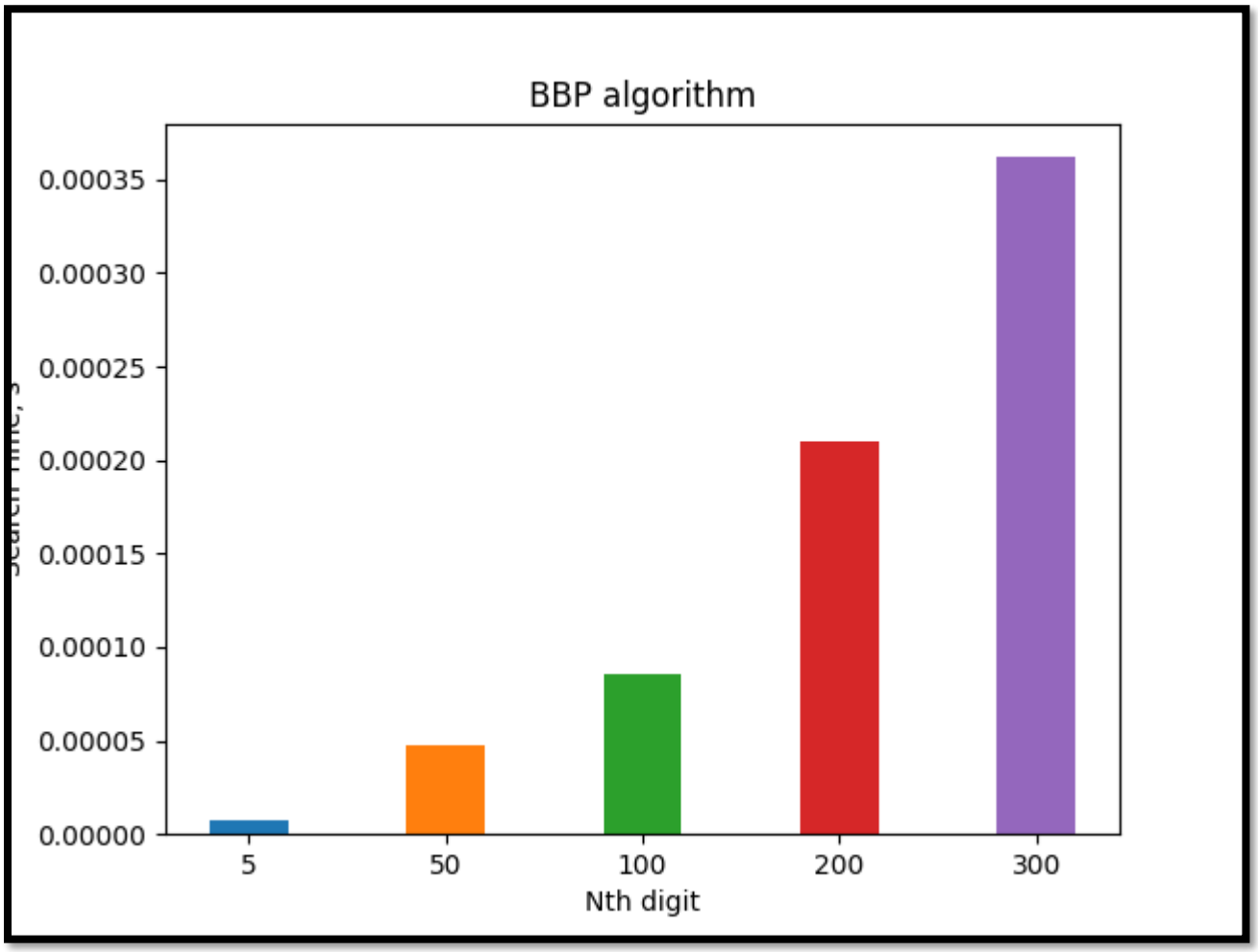
```python
def bbpPi(n):
    # Returns the nth decimal digit of Pi using the BBP formula.
    if n < 0:
        raise ValueError("Invalid value of n.")

    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	0.0003618999908212572 3

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:

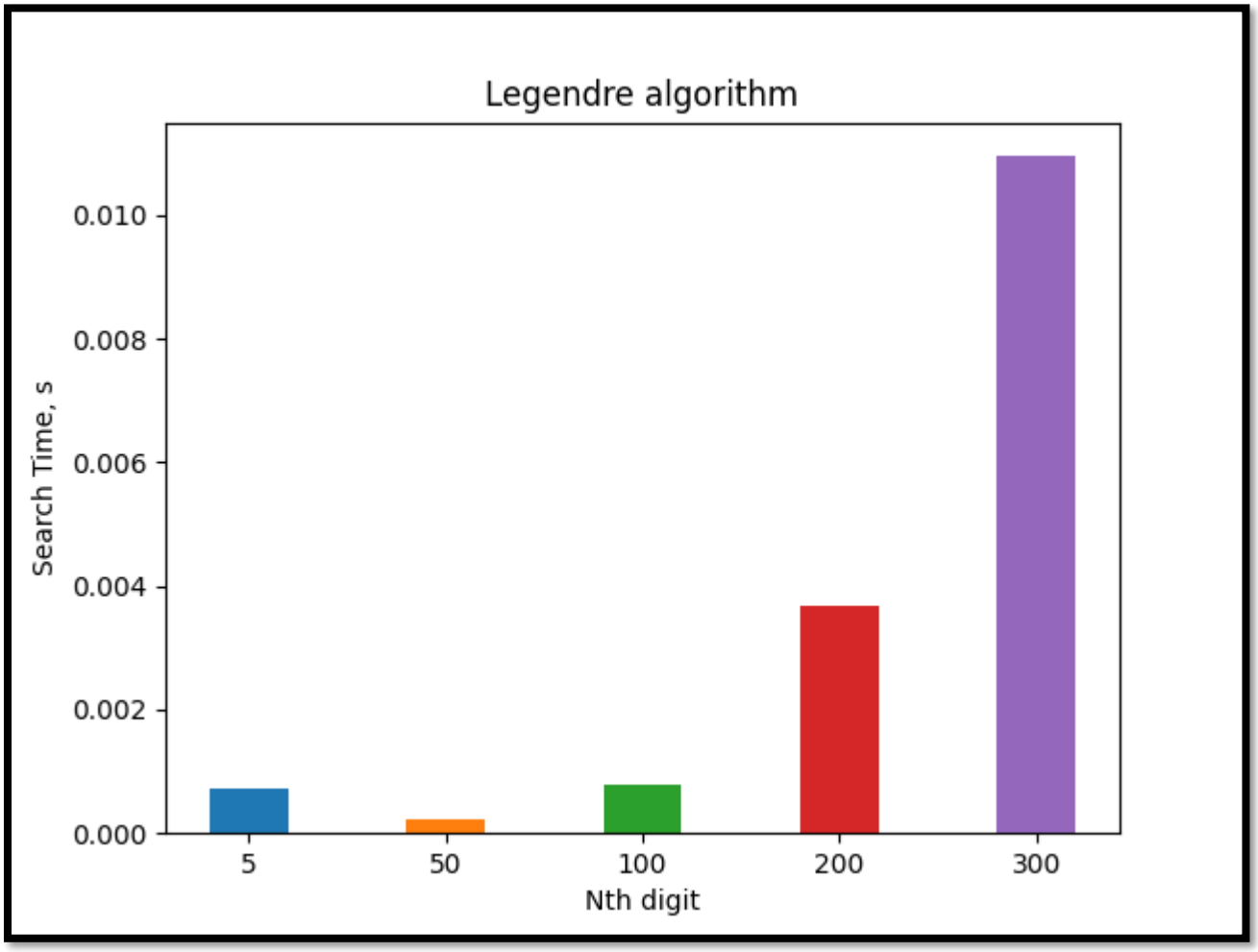
```python
def legendrePi(n):
    # Returns the nth decimal digit of Pi using the Gauss-Legendre algorithm.
    if n < 0:
        raise ValueError("Invalid value of n.")

    getcontext().prec = n + 1  # Set the precision to n+1 decimal places

    a = Decimal(1)
    b = Decimal(1) / Decimal(2).sqrt()
    t = Decimal(1) / Decimal(4)
    p = Decimal(1)

    for _ in range(n):
        atmp = (a + b) / Decimal(2)
        b = (a * b).sqrt()
        t -= p * (a - atmp) ** Decimal(2)
        a = atmp
        p *= Decimal(2)

    pi = (a + b) ** Decimal(2) / (Decimal(4) * t)

    return int(str(pi)[n])
```

The time complexity of the Gauss-Legendre algorithm for computing pi is O(n log n), where n is the number of iterations performed. This is because each iteration requires computing the square root of a number, which can be done in O(log n) time using algorithms like binary search or Newton's method.
Additionally, each iteration involves some basic arithmetic operations like addition, multiplication, and division, which can be performed in constant time.
Therefore, the overall time complexity of the algorithm is O(n log n), which means that as the number of iterations n increases, the runtime of the algorithm will grow roughly proportional to n log n. This makes the Gauss-Legendre algorithm faster than some other algorithms for computing pi, like the BBP formula which has a time complexity of O(n^2).

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:

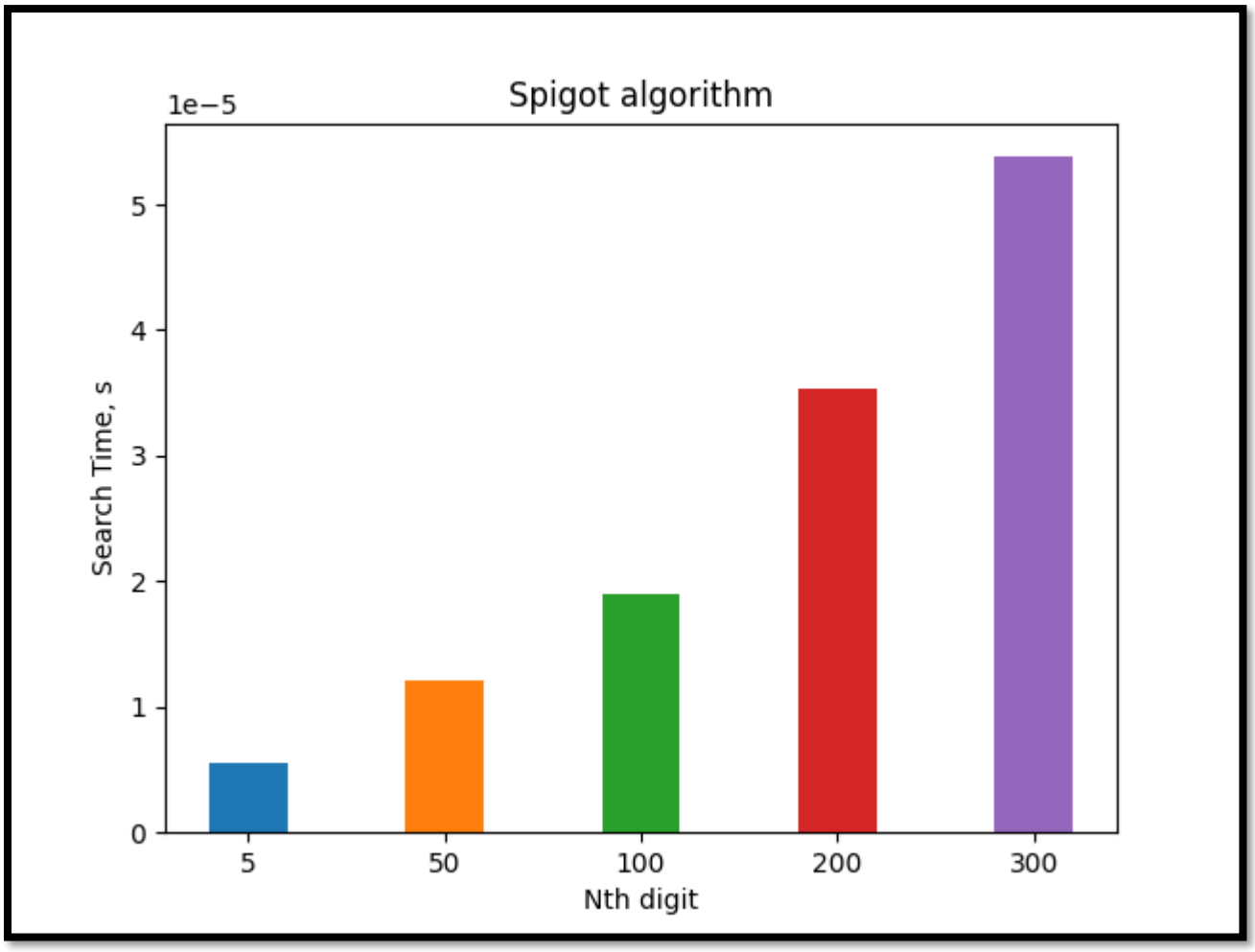
```python
def spigotPi(n):
    # Returns the nth decimal digit of Pi using the spigot algorithm.
    if n < 0:
        raise ValueError("Invalid value of n.")

    pi = 0
    d = 1
    for i in range(n):
        pi += 4 * d
        d = (d * 10 - int(d * 10 / 10) * 10)

    return int(pi / pow(10, n - 1)) % 10
```

The time complexity of the spigot algorithm to compute the n-th digit of pi is O(n^2), where n is the number of decimal places of pi to be computed. This is because the algorithm involves a nested loop structure where, for each decimal place of pi to be computed, the algorithm must compute the remainder for a series of division operations involving increasingly large integers. In each iteration of the loop, the algorithm performs a series of arithmetic operations that take constant time, and a division operation that takes O(n) time, since the size of the numbers involved in the division operation grows with each iteration. Therefore, the overall time complexity of the algorithm is proportional to the number of iterations, which is O(n).
Since each iteration involves a division operation that takes O(n) time, the overall time complexity of the algorithm is O(n^2). However, in practice, the spigot algorithm is much faster than this worst-case time complexity suggests, due to the use of various optimizations, such as precomputing values that can be reused in subsequent iterations, and avoiding redundant calculations.

Results:

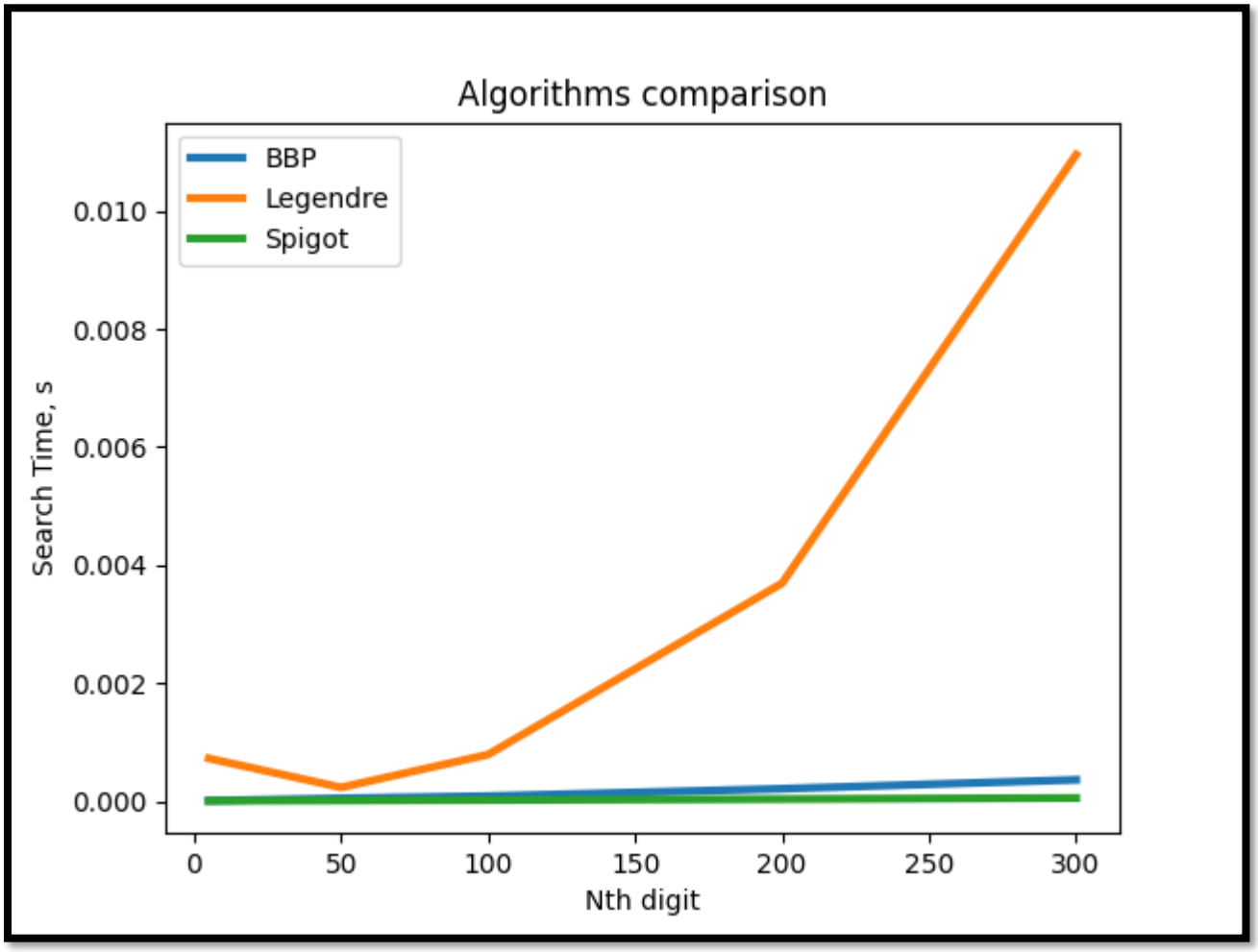
Algorithm/Nth digit	5	50	100	200	300
Spigot	5.499983672052622e-06	1.209997572004795e-05	1.8999999156221747e-05	3.530000685714185e-05	5.3800002206116915e-05

Graph:


Spigot algorithm
1e−5
Search Time, s
0
1
2
3
4
5
5
50
100
200
300
Nth digit

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^2), 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 log n), 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 log n), 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)