**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-212 Lupascu Felicia

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

**Table of Contents**

[ALGORITHM ANALYSIS 3](#_bookmark0)

[Objective 3](#_bookmark1)

[Tasks 3](#_bookmark2)

[Theoretical notes 3](#_bookmark3)

[Introduction 3](#_bookmark4)

[Comparison Metric 4](#_bookmark5)

[Input Format 4](#_bookmark6)

[IMPLEMENTATION 4](#_bookmark7)

[Bailey-Borwein-Plouffe 4](#_bookmark8)

[Gauss-Legendre 5](#_bookmark9)

[Spigot 7](#_bookmark10)

[All algorithms compared 10](#_bookmark11)

[CONCLUSION 11](#_bookmark12)

[REFERENCES 11](#_bookmark13)

**Objective**

# ALGORITHM ANALYSIS

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

A different approach to evaluating complexity involves empirical analysis, where the algorithm is implemented in a programming language and executed with various input data sets to gather information about its efficiency. This technique serves multiple purposes, including obtaining initial insights into the algorithm's complexity class, comparing different algorithms or implementations in terms of efficiency, and evaluating an algorithm's performance on a specific computer.

The choice of efficiency measurement depends on the specific goal of the analysis. If the aim is to gather information about the complexity class or validate a theoretical estimation, then the number of operations performed is a suitable metric. However, if the objective is to assess how well the algorithm performs in practice, then execution time is more appropriate. The obtained results are recorded, and synthetic quantities like mean and standard deviation may be calculated, or a graph with relevant data points can be plotted to analyze the information.

## Introduction

A theory concerning algorithms for calculating the nth decimal digit of pi is based on the concept of digit-extraction algorithms. These algorithms enable the computation of specific digits of a number without needing to calculate preceding digits. Among the well-known digit-extraction algorithms for pi is the BBP formula, which directly computes the nth hexadecimal or binary digit of pi. However, no digit-extraction algorithm is currently known to efficiently produce decimal digits of pi. Another option for obtaining the nth decimal digit of pi is by employing the Plouffe formula (2022), which involves a function denoted as \n(1). The formula then calculates the digit located at a given position, \n(2), by using the integer part, , and the fractional part,for determining the nth decimal digit of pi. One approach involves using series expansions that provide an approximation of pi with a specific level of precision.

## Comparison Metric

## In this laboratory experiment, the measure for comparing the algorithms will be the execution time (T(n)) of each algorithm.

## Input Format

The provided input is the value of "n," which represents the specific decimal digit of pi to be determined.

# IMPLEMENTATION

I implemented 3 algorithms: Spigot, Bailey-Borwein-Plouffe (BBP), Gauss-Legendre. All three algorithms will be implemented in their basic form using Python and analyzed through empirical means by measuring the time taken for their execution. Although the overall pattern of the results may resemble other experimental findings, the specific efficiency relative to the input will differ depending on the performance of the device used.

## Bailey-Borwein-Plouffe (BBP)

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):  
 if n < 0:  
 raise ValueError("n must be a non-negative integer")  
 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)

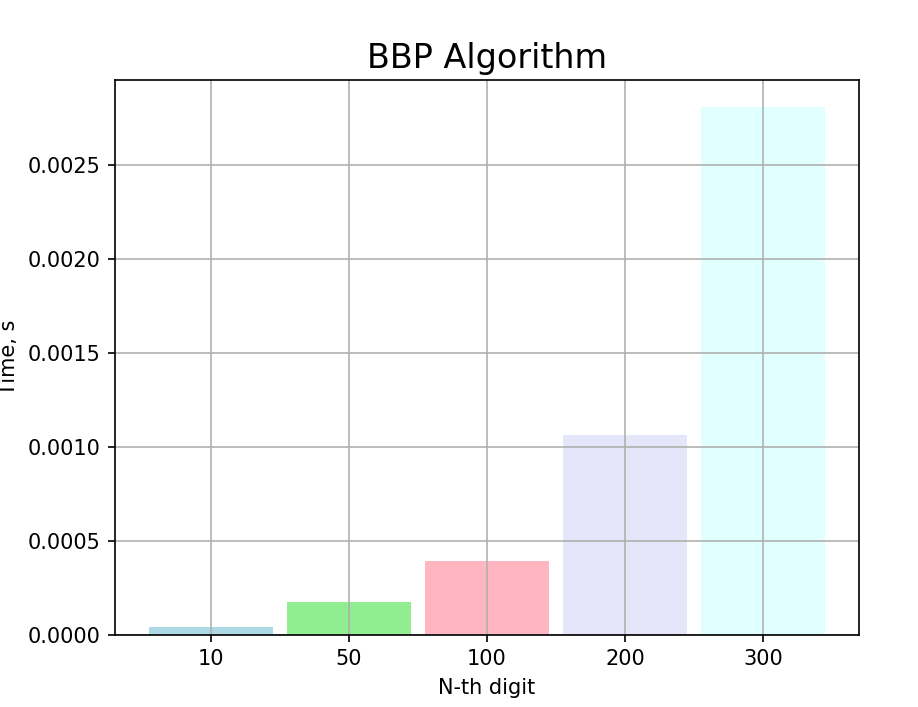
The given code defines a function named bbpPi(n) that calculates the nth decimal digit of pi using the Bailey–Borwein–Plouffe (BBP) formula.

Here's how the code works:

* First, it checks if the input n is a non-negative integer. If n is negative, it raises a ValueError with an appropriate error message.
* The variable pi is initialized to 0. This will store the calculated value of pi.
* A loop is set up to iterate from 0 to n (inclusive), represented by the variable k.
* Inside the loop, the BBP formula is used to calculate the partial sum for each value of k. The formula computes the kth term of the series expansion for pi.
* The computed partial sum is added to the variable pi.
* After the loop completes, the final value of pi is multiplied by 10 raised to the power of n and then modulo 10 is taken to extract the nth decimal digit.
* The calculated digit is then converted to an integer and returned as the result.

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:**

****

## 

## 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

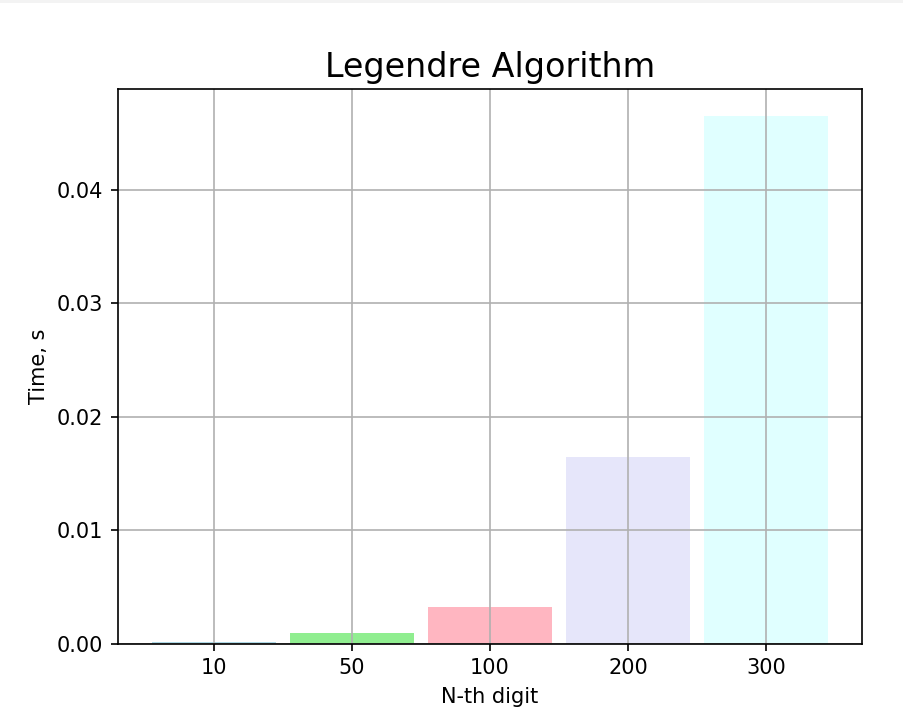
b = new\_b t = new\_t p = new\_p

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

**Implementation:**

def legendrePi(n):  
 if n < 0:  
 raise ValueError("n must be a non-negative integer")  
 getcontext().prec = n + 1  
 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 code defines a function named legendrePi(n) that calculates the nth decimal digit of pi using the Legendre algorithm. It sets the precision of decimal arithmetic, initializes variables, and iteratively updates their values. Finally, it calculates pi and returns the nth decimal digit of pi as an integer.



## Spigot

## A variant of the spigot approach uses an algorithm which can be used to compute a single arbitrary digit of the transcendental without computing the preceding digits: an example is the Bailey–Borwein–Plouffe formula, a digit extraction algorithm for π which produces base 16 digits . The inevitable truncation of the underlying infinite series of the algorithm means that the accuracy of the result may be limited by the number of terms calculated.

Pseudocode:

function compute\_pi(n): result = ""

remainder = 0

for i in range(1, n+1):

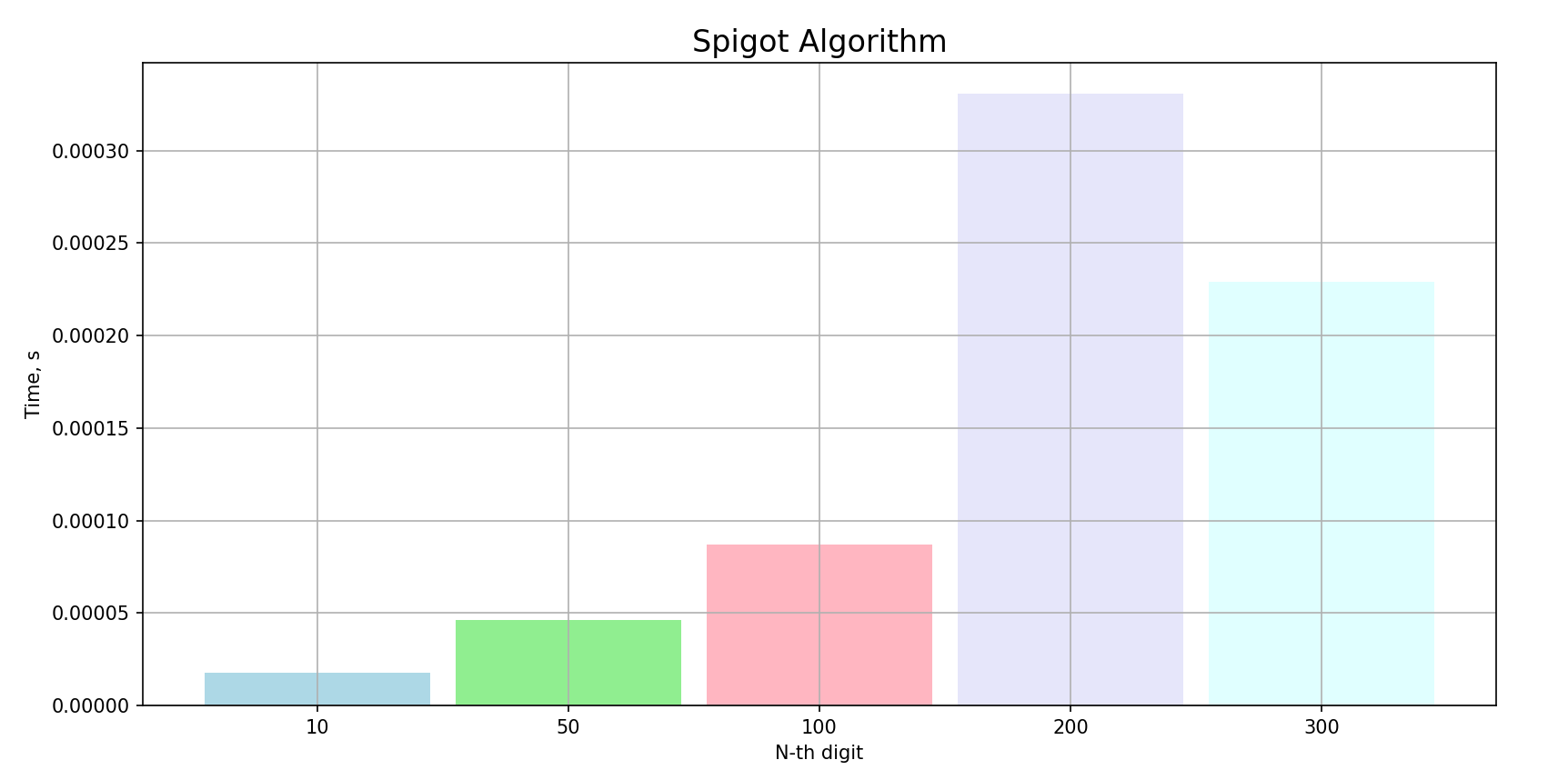
remainder = (remainder \* 10 + 1) % i digit = (4 \* remainder) // i

result += str(digit) return result

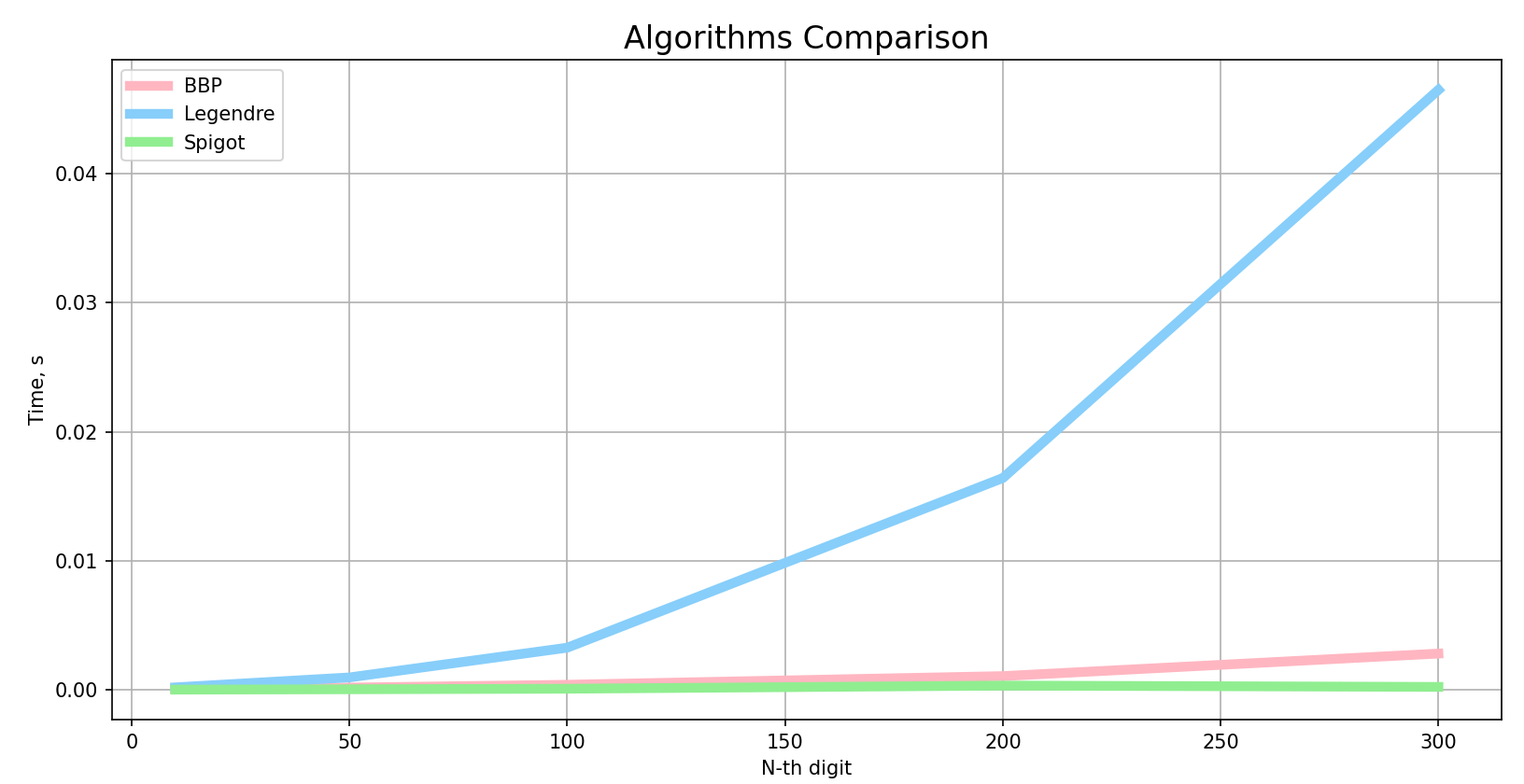
**Implementation:**

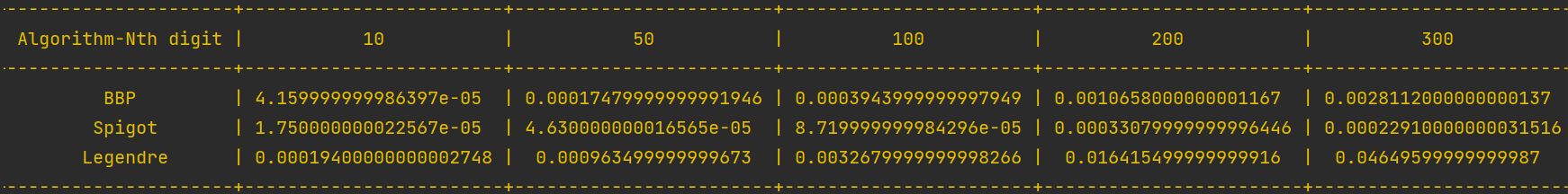
def spigotPi(n):  
 if n < 0:  
 raise ValueError("n must be a non-negative integer")  
 pi = 0  
 d = 1  
 for i in range(n):  
 pi += 4 \* d  
 d = (d \* 10 - int(d \* 10 / 10) \* 10)  
 return int(pi / pow(10, n - 1)) % 10

The code defines a function named spigotPi(n) that calculates the nth decimal digit of pi using the spigot algorithm. It checks if n is a non-negative integer, initializes variables, and iteratively updates their values. Finally, it calculates pi and returns the nth decimal digit of pi as an integer.



**Algorithms Comparison:**

****

****

The efficiency or superiority of a particular algorithm, such as the spigot algorithm, over others like the Legendre algorithm or Bailey-Borwein-Plouffe (BBP) formula, depends on various factors such as computational speed, accuracy, and memory usage.

While the spigot algorithm has some advantages, it may not necessarily be universally better than the Legendre or BBP algorithms in all aspects. Here are a few reasons why the spigot algorithm could be considered advantageous:

1. Digit-by-digit calculation: The spigot algorithm calculates pi digit by digit, which means it can produce a specific decimal digit without needing to compute preceding digits. This can be beneficial in situations where only a specific digit is required, as it avoids unnecessary calculations.
2. Efficient memory usage: The spigot algorithm typically uses a minimal amount of memory because it doesn't require storing large intermediate results. This can be advantageous in memory-constrained environments or when dealing with extremely large computations.
3. Simplicity and ease of implementation: The spigot algorithm is often relatively straightforward to implement compared to more complex formulas like the Legendre or BBP algorithms. Its simplicity can make it easier to understand, modify, or adapt for specific purposes.

However, it's important to note that the superiority of an algorithm can vary depending on the specific use case and requirements. Different algorithms may excel in different areas, such as speed of calculation, accuracy, or applicability to specific problem domains. Therefore, the choice of algorithm should be based on the specific needs and constraints of the problem at hand.

# 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.

<https://github.com/feliciaL3/APA_LABS/tree/main/LAB_6>