TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT: **SOFTWARE ENGINEERING AND AUTOMATION**

CC - LABORATORY WORK NR 1

Analiza algoritmilor

Ciuş Iurie FAF-203

Chişinău 2022

Contents

1 Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

1.1 Objectives

- Implement at least 3 algorithms that determine the N-th Fibonacci term.
- Establish the properties of the input data in relation to which the analysis is made.
- Choose the metric for comparing algorithms.
- Perform empirical analysis of the proposed algorithms.
- Make a conclusion on the work done.

1.2 Introduction

In mathematics, the **Fibonacci numbers**, commonly denoted F_n, form a sequence, the Fibonacci sequence, in which each number is the sum of the two preceding ones. The sequence commonly starts from 0 and 1, although some authors omit the initial terms and start the sequence from 1 and 1 or from 1 and 2. Starting from 0 and 1, the next few values in the sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$$

Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the Fibonacci Quarterly. Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems. They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern, and the arrangement of a pine cone's bracts.

Fibonacci numbers are strongly related to the golden ratio: Binet's formula expresses the *nth* Fibonacci number in terms of n and the golden ratio, and implies that the ratio of two consecutive Fibonacci numbers tends to the golden ratio as n increases. Fibonacci numbers are also closely related to Lucas numbers, which obey the same recurrence relation and with the Fibonacci numbers form a complementary pair of Lucas sequences.

1.3 Theoretical Notes

1.3.1 Algorithm execution time

Often, to solve a problem, one algorithm must be chosen from several possible ones, two main selection criteria being contradictory:

1. the algorithm should be easy to understand, code and troubleshoot.

2. the algorithm to efficiently use computer resources, to have a short execution time.

If the program being written has to be run a small number of times, the first requirement is more importance; In this situation, the time to set up the program is more important than its time so the simplest version of the program should be chosen.

If the program is to be run a large number of times, with a large number of data processed, the algorithm that leads to a faster execution must be chosen. Even in this situation, it should previously implemented the simplest algorithm and calculated the reduction of execution time that would bring it the implementation of the complex algorithm.

The running time of a program depends on the following factors:

- input data;

- the quality of the code generated by the compiler;

- the nature and speed of execution of the program instructions;

- the complexity of the algorithm that underlies the program.

So the running time is a function of its input, most of the time, not depending on the values from input, but the number of data.

Next we will denote by *T(n)* the execution time of an algorithm destined to solve a size problems n. In order to estimate the execution time, a calculation model must be established and a Unit. We will consider a computational model (also called a random access computing machine) characterized by:

- Processing is performed sequentially.
- Elementary operations are performed in constant time regardless of the value of the operands.
- The access time to the information does not depend on its position (there are no differences between the processing the first element and the last element of an array).

The importance of the worst case scenario. In the appreciation and comparison of algorithms, the most interesting unfavorable case because it provides the longest execution time relative to any size input data fixed. On the other hand, for some algorithms the worst case is relatively common

As for the analysis of the most favorable case, it provides a lower margin of time execution and can be useful for identifying inefficient algorithms (if an algorithm has a high cost in the most favorable case, then it cannot be considered an acceptable solution).

Average execution time. Sometimes extreme cases (the most unfavorable and the most favorable) are rare, so the analysis of these cases does not provide enough information about the algorithm.

In these situations, another measure of the complexity of the algorithms is useful, namely the average execution time. This represents an average value of the execution times calculated in relation to the probability distribution corresponding to the input data space.

1.3.2 Empirical analysis of the complexity of algorithms

An alternative to mathematical analysis of complexity is *empirical analysis.*

This may be useful for: (i) obtaining preliminary information on the complexity class of a algorithm; (ii) to compare the efficiency of two (or more) algorithms for solving the same problems; (iii) to compare the efficiency of several implementations of the same algorithm; (iv) to obtain information on the efficiency of implementing an algorithm on a particular computer.

The stages of empirical analysis. In the empirical analysis of an algorithm the following steps are usually followed:

1. The purpose of the analysis is established
2. Choose the efficiency metric to be used (number of executions of a / some operations or time execution of the whole algorithm or a part of the algorithm.
3. The properties of the input data are established in relation to which the analysis is performed (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Multiple sets of input data are generated.
6. The program runs for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. If the goal is to assess behavior implementation of an algorithm then is appropriate to the execution time.

To perform an empirical analysis is not enough a single set of input data but several, which highlight different features of the algorithm. It is generally good to choose data different sizes so as to cover the range of all dimensions that will appear in practice. On on the other hand, the analysis of different values or configurations of the input data is also important. If so analyzes an algorithm that verifies whether a number is prime or not and testing is done only for numbers which are not prime or only for numbers that are prime then you will not get a relevant result. The same thing can happen with a behavioral algorithm depending on the degree of sorting of one array (if you choose only the array almost sorted according to the desired criterion or arrays ordered in reverse analysis will not be relevant).

In order to empirically analyze the implementation of the algorithm in a programming language will need introduced sequences whose purpose is to monitor execution. If the efficiency metric is the number of executions of an operation then a counter is used which is incremented after each execution a that operation. If the metric is the execution time then the time of entry must be recorded the analyzed sequence and the time of exit. Most programming languages offer measurement functions the time elapsed between two moments. This is especially important if you are active on your computer several tasks, to count only the time allocated to the execution of the analyzed program. Especially if it is about measuring the time it is indicated to run the test program several times and to calculate the average value of time.

After the execution of the program for the test data the results are recorded and for the purpose of the analysis either calculates synthetic quantities (mean, standard deviation, etc.) or plot pairs of points shape (problem size, efficiency measure).

2 Code

The following pages show my code implementation and result of the choosen algorithms.

2.1 Implementation

FIBONACCI.py

```python
import math


class FIBONACCI:
    def __init__(self):
        pass

    # METHOD 1: Use Recursion

    def recursion(self, n):
        if n <= 1:
            return n
        else:
            return (self.recursion(n - 1) + self.recursion(n - 2))

    # METHOD 2: Use Dynamic Programming

    def dynamic(self, n):
        fibonacci_list = [0, 1]

        for i in range(2, n + 1):
            fibonacci_list.append(
                fibonacci_list[i - 1] + fibonacci_list[i - 2])

        return fibonacci_list[n]
```

```python

# METHOD 3: Space Optimized Method 2

def dynamic_optimized(self, n):
   a, b = 0, 1

   if n == 0:
      return a
   elif n == 1:
      return b
   else:
      for i in range(2, n + 1):
         c = a + b
         a = b
         b = c

      return b

# METHOD 4: Using formula
def formula(self, n):
   phi = (math.sqrt(5) + 1) / 2

   return round(phi**n / math.sqrt(5))
```

App.py

```
import time
import numpy as np
import matplotlib.pyplot as plt
from FIBONACCI import FIBONACCI


FB = FIBONACCI()


def compute_time(algorithm, numbers):
   elapsed_time = []
   for n in numbers:
      start = time.time()
      algorithm(n)
      end = time.time()
      elapsed_time.append(round(end - start, 6))
   return elapsed_time


a = np.arange(0, 35, 1)
b = np.arange(0, 100000, 3000)
c = np.arange(0, 100, 1)

recursion = compute_time(FB.recursion, a)
dynamic = compute_time(FB.dynamic, b)
dynamic_optimized = compute_time(FB.dynamic_optimized, b)
formula = compute_time(FB.formula, c)


plt.figure(figsize=(8, 6), dpi=80)
fig, axs = plt.subplots(2, 2)
```

```
axs[0, 0].set_title("Recursion")
axs[0, 0].plot(a, recursion)
axs[0, 1].set_title("Dynamic")
axs[0, 1].plot(b, dynamic, 'tab:orange')
axs[1, 0].set_title("Dynamic Optimized")
axs[1, 0].plot(b, dynamic_optimized, 'tab:green')
axs[1, 1].set_title("Golden Ration Formula")
axs[1, 1].plot(c, formula, 'tab:red')

for ax in axs.flat:
   ax.set(xlabel='Time in ms', ylabel='Nth Fibonacci\'s Number')

plt.show()
```

2.2 Time Complexity

Method 1 (Recursion)

Time Complexity: T(n) = T(n) which is linear.

If the original recursion tree were to be implemented then this would have been the tree but now for n times the recursion function is called

Original tree for recursion

```
                    fib(5)
                /            \
           fib(4)          fib(3)
          /      \          /     \
       fib(3)   fib(2)     fib(2) fib(1)
      /   \     /   \      /    \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
  /   \
fib(1) fib(0)
```

Optimized tree for recursion for code above

- fib(5)
- fib(4)
- fib(3)
- fib(2)
- fib(1)

O(n) if we consider the function call stack size, otherwise O(1).

Method 2 (Dynamic Programming)

For dynamic Programming, the time complexity would be O(n) since we only loop through it once. As you can see in the dynamic programming procedure chart, it is linear.

And the space complexity would be $O(N)$ since we need to store all intermediate values into our list. So the space we need is the same as n given.

Method 3 (Space Optimized Method 2)

Time Complexity: O(n)

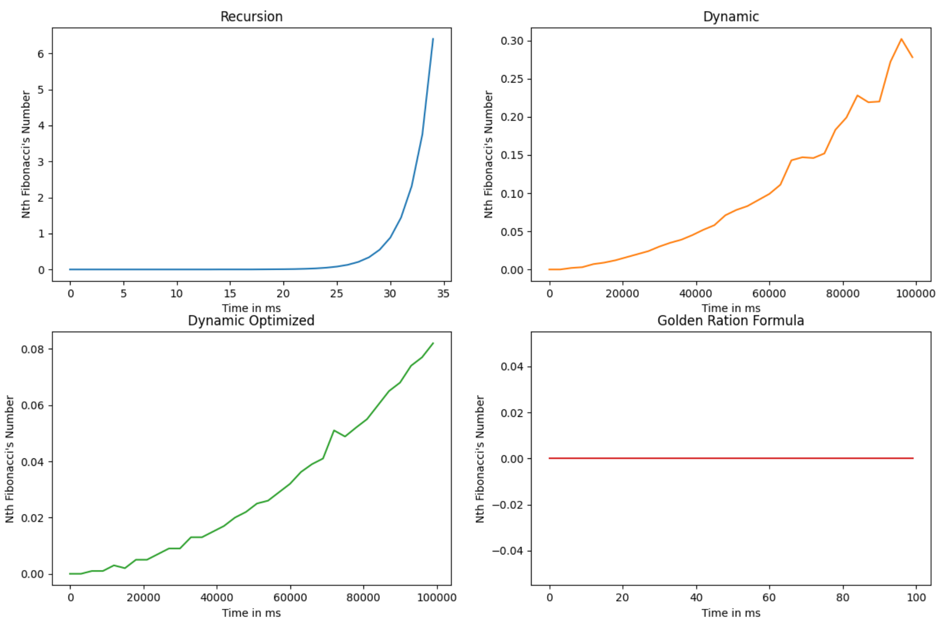
Extra Space: O(1)

Method 4 (Using formula)

Time Complexity: O(logn)

Extra Space: O(1)

2.3 Graphs



3 Conclusion

The Fibonacci sequence may not be the perfect example for an in-depth understanding of dynamic programming. But it shows us the steps to convert a recursive solution into a dynamic programming solution. To start with the idea of dynamic programming, it is a simple and easy-to-understand example. We can expand our understanding of dynamic programming by solving problems like —

- Longest Common Subsequence problem.
- Shortest Common Subsequence problem.
- 0/1 Knapsack problem.
- Matrix Chain Multiplication problem.
- Rod Cutting Problem.

We analyzed the time complexity of 4 different algorithms that find the nth value in the Fibonacci Sequence and plot the time versus the nth Fibonacci's number.

3.1 References

- https://github.com/IuraCPersonal/CC