# aula1-pratico

August 13, 2024

## 1 Class 01 - Practical exercise complexity analysis

GITHUB: https://github.com/pantro/doutoradoCS/blob/main/Analise%20de%20Algoritmos%20e%20Estruturas pratico.ipynb

#### 1.0.1 1) Propose a recursive algorithm to determine whether a number is prime or not. Propose an iterative algorithm for the same purpose.

- Execute both functions to find the largest prime number in a random vector of 100,000 elements (use the same vector in both codes).
- Check which of the two executed faster (use a time() function).
- Perform the time complexity analysis of both algorithms.

### 1.1 SOLUTION

For a number to be prime it must meet the following requirements: * Numbers less than 2 are not prime. * The number 2 is the only even prime number. * If a divisor is found that divides the number exactly (without a remainder), the algorithm must stop and conclude that the number is not prime. * The number must be divisible only by 1 and itself. This involves checking whether the number is divisible by any integer less than itself.

#### 1.1.1 Optimize the algorithm

- In order to optimize the search for prime numbers according to ChatGPT, we have to limit the checks to the square root of a number.

Explanation: * If a number has a divisor , If both and were greater than the square root of , then × would be greater than , which is a contradiction. This implies that if has divisors other than 1 and itself, then at least one of those divisors must be less than or equal to the square root of .

Example: Suppose you want to check if the number 29 is prime. * The square root of 29 is approximately 5.39. * To determine if 29 is prime, you only need to check if it is divisible by integers less than or equal to 5 (i.e., 2, 3, and 5). * If 29 is not divisible by any of these numbers, you can be sure that 29 has no additional divisors and is therefore prime.

### 1.1.2 Function Iterative

[8]:
```
def es_primo_iterativo(n):
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    # Check if the number  has any divisors between 3 and the square root of ␣
↪checking only the odd numbers.
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True
```

### 1.1.3 Complexity analysis of the function "es_primo_iterativo"

- if n <= 1 -> (1)
- if n == 2 -> (1)
- if n % 2 == 0 -> (1)
- for i in range(3, int(n**0.5) + 1, 2)->$\frac{\sqrt{n}}{2} - 2 = O(\sqrt{n})$
- if n % i == 0 -> (1)

Solution: The worst case time complexity is: $(\sqrt{n})$

### 1.1.4 Function Recursive

[9]:
```
def es_primo_recursivo(n, divisor=3):
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    if divisor > int(n**0.5):
        return True
    if n % divisor == 0:
        return False
    return es_primo_recursivo(n, divisor + 2)
```

### 1.1.5 Complexity analysis of the function "es_primo_recursivo"

- if n <= 1 -> (1)
- if n == 2 -> (1)
- if n % 2 == 0 -> (1)
- divisor > int(n**0.5) -> (1)
- if n % divisor == 0 -> (1)

- es_primo_recursivo(n, divisor + 2)->$\frac{\sqrt{n}}{2} - 2 = O(\sqrt{n})$

Solution: The worst case time complexity is: $(\sqrt{n})$

## 1.2 Test

- Run both functions to find the largest prime number in a random vector of 100,000 elements (use the same vector in both codes)

[13]:
```
# Library
import numpy as np
from datetime import datetime
# Semilla
np.random.seed(42)

# Create a vector with 100,000 random elements in the range 2 to 100,000
vector_aleatorio = np.random.randint(2, 100001, size=100000)

print("Size vector: ", len(vector_aleatorio))
vector_aleatorio[:10]
```

```
Size vector:  100000
```

[13]:
```
array([15797,   862, 76822, 54888,  6267, 82388, 37196, 87500, 44133,
       60265])
```

### 1.2.1 Iteration

[21]:
```
# Initial time
tiempo_inicial = datetime.now()
# Create vector of prime numbers
vector_primos = []

for n in vector_aleatorio:
    if (es_primo_iterativo(n)):
        vector_primos.append(n)

# Get the largest prime
if vector_primos:
    primo_mayor = max(vector_primos)
    # Print the largest prime number
    print("The largest prime number is:", primo_mayor)
else:
    print("There are no prime numbers in the vector")

# End time
tiempo_final = datetime.now()
diferencia_tiempo = tiempo_final - tiempo_inicial
```

```
print("Time: ", diferencia_tiempo)
```

```
The largest prime number is: 99989
Time:  0:00:00.292513
```

### 1.2.2 Recursive

[15]:
```
# Initial time
tiempo_inicial = datetime.now()
# Create vector of prime numbers
vector_primos = []

for n in vector_aleatorio:
    if (es_primo_recursivo(n)):
        vector_primos.append(n)

# Get the largest prime
if vector_primos:
    primo_mayor = max(vector_primos)
    # Print the largest prime number
    print("The largest prime number is:", primo_mayor)
else:
    print("There are no prime numbers in the vector")

# End time
tiempo_final = datetime.now()
diferencia_tiempo = tiempo_final - tiempo_inicial
print("Time: ", diferencia_tiempo)
```

```
The largest prime number is: 99989
Time:  0:00:03.434702
```

## 1.3 Conclusions

- We can see that both the iterative and recursive code have the same time complexity.
- When the test is run with 100,000 elements, the iterative algorithm is much faster than the recursive one. This may be because the recursive algorithm calls the entire function again and performs the IF comparisons again on each entry.
- Even though the iterative algorithm and the recursive algorithm have the same time complexity, the iterative algorithm is approximately 11 times faster than the recursive algorithm. This tells us that having the same time complexity does not necessarily ensure that they are close real times, but that the complexity value is a reference value.