**Análise empírica de algoritmos de ordenação**

**Nome: Matheus Henrique Scabia de Jesus**

O objetivo desta análise é comparar resultados obtidos de forma empírica, com os resultados encontrados de forma matemática, e ver se correspondem corretamente.

Vamos começar apresentando os algoritmos, tanto na linguagem Python quanto na linguagem C. E junto a eles, um gráfico relacionando o número N (que representa um tamanho de vetor), com o tempo para executar esta ordenação com determinado tamanho.

**Linguagem Python**

Algoritmo Insertion Sort:

import random

import matplotlib.pyplot as plt

import time

def insertSort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1

        while j >= 0 and key < arr[j]:

            arr[j+1] = arr[j]

            j = j-1

        arr[j+1] = key

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    insertSort(data)

    end\_time = time.time()

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 3), "segundos")

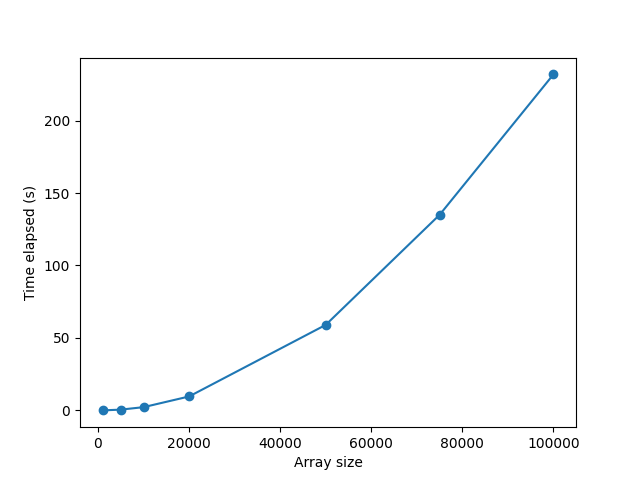
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()



Algoritmo QuickSort

import random

import time

import matplotlib.pyplot as plt

# function to partition the array on the basis of pivot element

def partition(array, low, high):

    i = (low - 1)

    pivot = array[high]

    for j in range(low, high):

        if array[j] <= pivot:

            i = i + 1

            # Swapping element at i with element at j

            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i

    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done

    return i + 1

# function to perform quicksort

def quickSort(array):

    # Create an empty stack

    stack = []

    # Push initial values of low and high to stack

    stack.append((0, len(array) - 1))

    # Loop until stack is empty

    while stack:

        # Pop low and high from stack

        low, high = stack.pop()

        # Set pivot element at its correct position in sorted array

        pi = partition(array, low, high)

        # If there are elements on left side of pivot,

        # then push left side to stack

        if pi - 1 > low:

            stack.append((low, pi - 1))

        # If there are elements on right side of pivot,

        # then push right side to stack

        if pi + 1 < high:

            stack.append((pi + 1, high))

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    quickSort(data)

    end\_time = time.time()

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 3), "segundos")

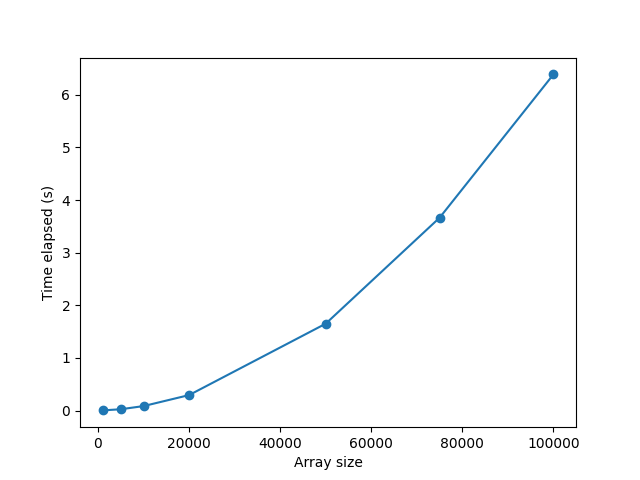
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()

****

Algoritmo BubbleSort:

import random

import matplotlib.pyplot as plt

import time

def bubbleSort(arr):

    for i in range(len(arr)):

        for j in range(len(arr) - 1):

            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    bubbleSort(data)

    end\_time = time.time()

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 3), "segundos")

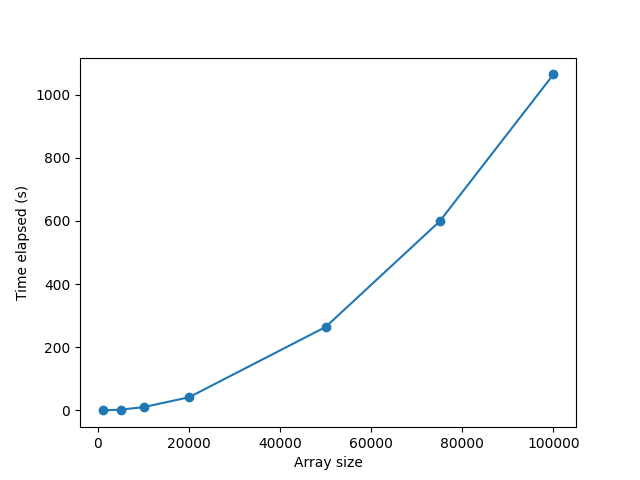
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()



Algoritmo de Busca sequencial:

import random

import re

import matplotlib.pyplot as plt

import time

def busca\_sequencial(lista, item):

    for i in range(len(lista)):

        if lista[i] == item:

            return i

    return None

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    retorno = busca\_sequencial(data, 53)

    end\_time = time.time()

    print(retorno)

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 6), "segundos")

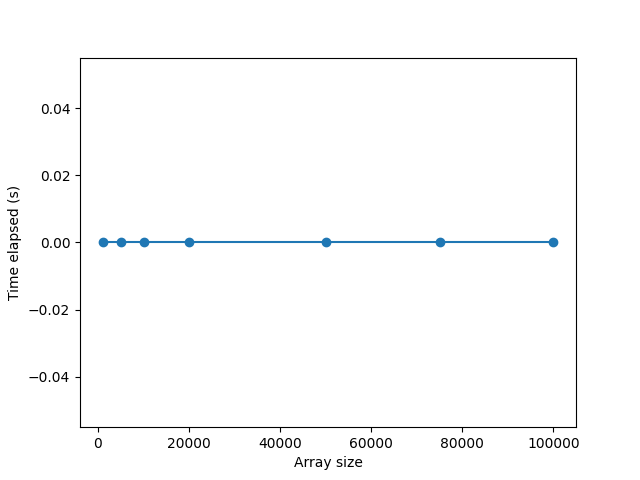
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()



Algoritmo de Busca binária não recursiva:

import random

import time

import matplotlib.pyplot as plt

def pesquisaBinariaNaoRecursiva(lista, item):

    lista\_ordenada = sorted(lista)

    primeiro = 0

    ultimo = len(lista\_ordenada) - 1

    while primeiro <= ultimo:

        meio = (primeiro + ultimo) // 2

        if lista\_ordenada[meio] == item:

            return meio

        else:

            if item < lista\_ordenada[meio]:

                ultimo = meio - 1

            else:

                primeiro = meio + 1

    return None

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    retorno = pesquisaBinariaNaoRecursiva(data, 50)

    end\_time = time.time()

    print(retorno)

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 3), "segundos")

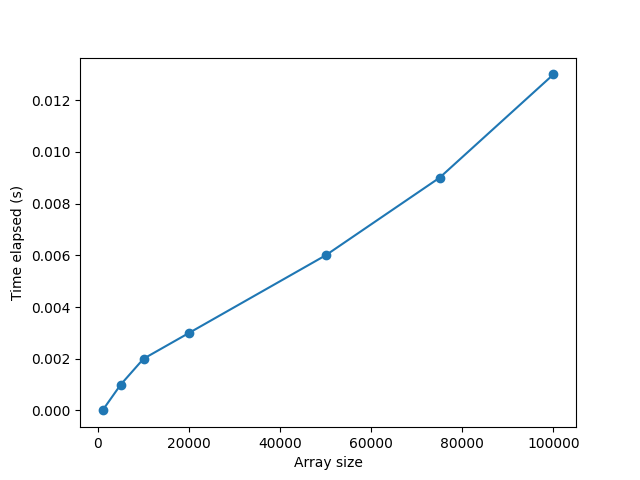
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()



Algoritmo de Busca Binaria Recursiva:

import random

import time

import matplotlib.pyplot as plt

def pesquisaBinariaRecursiva(lista, item):

    lista\_ordenada = sorted(lista)

    if len(lista\_ordenada) == 0:

        return False

    else:

        meio = len(lista\_ordenada) // 2

        if lista\_ordenada[meio] == item:

            return True

        else:

            if item < lista\_ordenada[meio]:

                return pesquisaBinariaRecursiva(lista\_ordenada[:meio], item)

            else:

                return pesquisaBinariaRecursiva(lista\_ordenada[meio + 1:], item)

# Generate random arrays of integers

sizes = [1000, 5000, 10000, 20000, 50000, 75000, 100000]

times = []

for size in sizes:

    data = [random.randint(0, 100) for \_ in range(size)]

    # Sort the array and measure the elapsed time

    start\_time = time.time()

    retorno = pesquisaBinariaRecursiva(data, 50)

    end\_time = time.time()

    print(retorno)

    # Append the elapsed time to the list of times

    times.append(end\_time - start\_time)

    print("Tempo decorrido no tamanho",size,":", round(end\_time - start\_time, 3), "segundos")

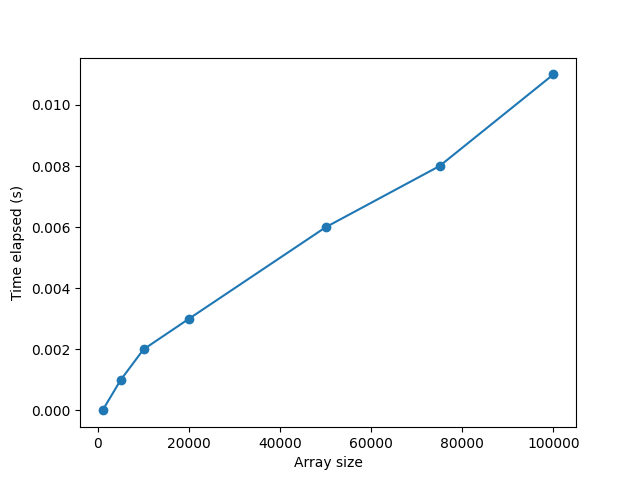
# Plot the graph

plt.plot(sizes, times, 'o-')

plt.xlabel('Array size')

plt.ylabel('Time elapsed (s)')

plt.show()



**Linguagem C**

Algoritmo de Insert Sort:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void insertionSort(int arr[], int n) {

    int i, key, j;

    for (i = 1; i < n; i++) {

        key = arr[i];

        j = i - 1;

        /\* Move os elementos do arr[0..i-1], que são maiores que a chave, para uma posição à frente de sua posição atual \*/

        while (j >= 0 && arr[j] > key) {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

int main() {

    int sizes[] = {1000, 5000, 10000, 20000, 50000, 75000, 100000}; /\* Tamanhos dos arrays a serem ordenados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 101; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        clock\_t inicio = clock(); /\* Marca o tempo de início da ordenação \*/

        insertionSort(arr, n);

        clock\_t fim = clock(); /\* Marca o tempo de fim da ordenação \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        printf("Tempo de execucao para tamanho %d foi: %.4f segundos\n", n, tempo);

    }

    return 0;

}

Algoritmo de bublesort:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void bubbleSort(int arr[], int n) {

    int i, j, temp;

    for (i = 0; i < n - 1; i++) {

        for (j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                /\* Troca os elementos adjacentes se o primeiro for maior que o segundo \*/

                temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

int main() {

    int sizes[] = {1000, 5000, 10000, 20000, 50000, 75000, 100000}; /\* Tamanhos dos arrays a serem ordenados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 101; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        clock\_t inicio = clock(); /\* Marca o tempo de início da ordenação \*/

        bubbleSort(arr, n);

        clock\_t fim = clock(); /\* Marca o tempo de fim da ordenação \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        printf("Tempo de execucao do tamanho %d foi: %.4f segundos\n", n, tempo);

    }

    return 0;

}

Algoritmo de QuickSort:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int\* a, int\* b) {

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

int partition(int arr[], int low, int high) {

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void quickSort(int arr[], int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

int main() {

    int sizes[] = {1000, 5000, 10000, 20000, 50000, 75000, 100000}; /\* Tamanhos dos arrays a serem ordenados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 101; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        clock\_t inicio = clock(); /\* Marca o tempo de início da ordenação \*/

        quickSort(arr, 0, n - 1);

        clock\_t fim = clock(); /\* Marca o tempo de fim da ordenação \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        printf("Tempo de execucao do tamanho %d foi: %.4f segundos\n", n, tempo);

    }

    return 0;

}

Algoritmo de Pesquisa sequencial:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int sequentialSearch(int arr[], int n, int x) {

    for (int i = 0; i < n; i++) {

        if (arr[i] == x) {

            return i;

        }

    }

    return -1;

}

int cmpfunc(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

int main() {

    int sizes[] = {1000, 5000, 10000, 20000, 50000, 75000, 100000}; /\* Tamanhos dos arrays a serem pesquisados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 101; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        int x = rand() % 101; /\* Gera um número aleatório para ser pesquisado \*/

        qsort(arr, n, sizeof(int), cmpfunc); /\* Ordena o array em ordem crescente \*/

        clock\_t inicio = clock(); /\* Marca o tempo de início da pesquisa \*/

        int pos = sequentialSearch(arr, n, x);

        clock\_t fim = clock(); /\* Marca o tempo de fim da pesquisa \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        if (pos == -1) {

            printf("O elemento %d nao foi encontrado no array de tamanho %d em %.4f segundos\n", x, n, tempo);

        } else {

            printf("O elemento %d foi encontrado na posicao %d do array de tamanho %d em %.4f segundos\n", x, pos, n, tempo);

        }

    }

    return 0;

}

Algoritmo de busca binária não recursivo:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int binarySearch(int arr[], int n, int x) {

    int left = 0, right = n - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (arr[mid] == x) {

            return mid;

        } else if (arr[mid] < x) {

            left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    return -1;

}

int cmpfunc(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

int main() {

    int sizes[] = {100000}; /\* Tamanhos dos arrays a serem pesquisados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 1001; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        int x = rand() % 1001; /\* Gera um número aleatório para ser pesquisado \*/

        qsort(arr, n, sizeof(int), cmpfunc); /\* Ordena o array em ordem crescente \*/

        clock\_t inicio = clock(); /\* Marca o tempo de início da pesquisa \*/

        int pos = binarySearch(arr, n, x);

        clock\_t fim = clock(); /\* Marca o tempo de fim da pesquisa \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        if (pos == -1) {

            printf("O elemento %d nao foi encontrado no array de tamanho %d em %.4f segundos\n", x, n, tempo);

        } else {

            printf("O elemento %d foi encontrado na posicao %d do array de tamanho %d em %.4f segundos\n", x, pos, n, tempo);

        }

    }

    return 0;

}

Algoritmo de busca binária recursiva:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int binarySearchRecursive(int arr[], int left, int right, int x) {

    if (left > right) {

        return -1;

    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == x) {

        return mid;

    } else if (arr[mid] < x) {

        return binarySearchRecursive(arr, mid + 1, right, x);

    } else {

        return binarySearchRecursive(arr, left, mid - 1, x);

    }

}

int cmpfunc(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

int main() {

    int sizes[] = {1000, 5000, 10000, 20000, 50000, 75000, 100000}; /\* Tamanhos dos arrays a serem pesquisados \*/

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); /\* Número de tamanhos \*/

    srand(time(NULL)); /\* Inicializa o gerador de números aleatórios com o tempo atual \*/

    for (int i = 0; i < num\_sizes; i++) {

        int n = sizes[i];

        int arr[n];

        for (int j = 0; j < n; j++) {

            arr[j] = rand() % 101; /\* Gera um número aleatório entre 0 e 100 \*/

        }

        int x = rand() % 101; /\* Gera um número aleatório para ser pesquisado \*/

        qsort(arr, n, sizeof(int), cmpfunc); /\* Ordena o array em ordem crescente \*/

        clock\_t inicio = clock(); /\* Marca o tempo de início da pesquisa \*/

        int pos = binarySearchRecursive(arr, 0, n - 1, x);

        clock\_t fim = clock(); /\* Marca o tempo de fim da pesquisa \*/

        double tempo = (double)(fim - inicio) / CLOCKS\_PER\_SEC; /\* Calcula o tempo de execução em segundos \*/

        if (pos == -1) {

            printf("O elemento %d nao foi encontrado no array de tamanho %d em %.4f segundos\n", x, n, tempo);

        } else {

            printf("O elemento %d foi encontrado na posicao %d do array de tamanho %d em %.4f segundos\n", x, pos, n, tempo);

        }

    }

    return 0;

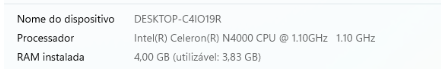
}

**Tabela comparando tempos de 2 diferentes máquinas:**

Máquina 1:



Máquina 2:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Tempo (segundos)** | | | | **N = 1000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | **Função custo** | **Classificação assintótica** |
| **Insertsort** | 0,003 | 0,004 | 0.025 | 0,04 | F(n) = n^2 | O(n^2) |
| **Bubblesort** | 0,003 | 0,003 | 0.096 | 0,11 | F(n) = n^2 | O(n^2) |
| **Quicksort** | 0 | 0 | 0.002 | 0,01 | F(n) = n log n | O(n log n) |
| **Busca sequencial** | 0 | 0 | 0.001 | 0,005 | F(n) = n | O(n) |
| **Busca binária Não recursiva** | 0 | 0 | 0.001 | 0,007 | F(n) = log n | O(log n) |
| **Busca Binária Recursiva** | 0 | 0 | 0.001 | 0,005 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 5000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 0,017 | 0,032 | 0.563 | 0,64 | F(n) = n^2 | O(n^2) |
| Bubblesort | 0,068 | 0,132 | 2.554 | 3,221 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,001 | 0,01 | 0.024 | 0,7 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0,003 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0.001 | 0,003 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0.001 | 0,002 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 10000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 0,08 | 0,129 | 2.258 | 2,669 | F(n) = n^2 | O(n^2) |
| Bubblesort | 0,279 | 0,4512 | 10.381 | 12,045 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,001 | 0,009 | 0.084 | 0,102 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0,004 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0.001 | 0,001 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0,002 | 0,002 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 20000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 0,329 | 0,756 | 9.638 | 10,978 | F(n) = n^2 | O(n^2) |
| Bubblesort | 1,056 | 2,873 | 41.785 | 43,67 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,008 | 0,005 | 0.293 | 0,413 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0,002 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0.002 | 0,005 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0,003 | 0,007 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 50000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 2,685 | 4,159 | 59.014 | 65,987 | F(n) = n^2 | O(n^2) |
| Bubblesort | 6,79 | 8,34 | 264.47 | 272,56 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,04 | 0,103 | 1.65 | 1,93 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0.003 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0,008 | 0,017 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0,006 | 0,013 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 75000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 3,875 | 4,472 | 134.931 | 150,876 | F(n) = n^2 | O(n^2) |
| Bubblesort | 15,986 | 17,82 | 598.552 | 621,264 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,084 | 0,278 | 3.633 | 4,2378 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0.004 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0,009 | 0,13 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0,008 | 0,019 | F(n) = log n | O(log n) |
|  |  |  |  |  |  |  |
|  | **Tempo (segundos)** | | | | **N = 100000** |  |
|  | **Linaguagem C Maquina 1** | **Linaguagem C Maquina 2** | **Python Maquina 1** | **Python Maquina 2** | Função custo | Classificação assintótica |
| Insertsort | 9,447 | 11,002 | 231.817 | 250,825 | F(n) = n^2 | O(n^2) |
| Bubblesort | 28,821 | 31,097 | 1.064,063 | 1164,3892 | F(n) = n^2 | O(n^2) |
| Quicksort | 0,151 | 0,378 | 6.387 | 7,421 | F(n) = n log n | O(n log n) |
| Busca sequencial | 0 | 0 | 0.001 | 0.003 | F(n) = n | O(n) |
| Busca binária Não recursiva | 0 | 0 | 0,011 | 0,011 | F(n) = log n | O(log n) |
| Busca Binária Recursiva | 0 | 0 | 0,011 | 0,011 | F(n) = log n | O(log n) |

**Conclusão**

Com base na análise empírica realizada, podemos concluir que os resultados obtidos foram consistentes com as análises matemáticas e esperadas dos algoritmos estudados.

Os algoritmos de ordenação Insertion Sort e Bubble Sort apresentaram um desempenho inferior em relação ao QuickSort, tanto em termos de tempo de execução quanto em relação à função custo. Isso ocorre porque esses algoritmos têm uma classificação assintótica de O(n^2), o que leva a um crescimento quadrático do tempo de execução com o tamanho da lista.

Por outro lado, o algoritmo Quick Sort apresentou um desempenho superior em relação aos outros algoritmos, tanto em termos de tempo de execução quanto em relação à função custo. Isso ocorre porque o Quick Sort tem uma classificação assintótica de O(n log n, o que leva a um crescimento mais suave do tempo de execução com o tamanho da lista.

Além disso, a análise empírica também mostrou que a implementação correta dos algoritmos é fundamental para obter um bom desempenho. Pequenas alterações no código podem ter um grande impacto no tempo de execução e na eficiência do algoritmo. Além também, do fato de duas máquinas diferentes poderem apresentarem resultado diferentes, dependendo da configuração de hardware.

Em resumo, a análise empírica dos algoritmos de ordenação nos permitiu entender melhor o desempenho e a eficiência de cada algoritmo, bem como a importância da implementação correta. Esses conhecimentos são fundamentais para a escolha do algoritmo mais adequado para cada situação e para a otimização do desempenho de sistemas computacionais.