Análise e Implementação do Algoritmo de Ordenação Quick Sort

1st Eduardo Henrique Queiroz Almeida
Dept. Engenharia da Computação
CEFET-MG
Divinópolis, Brasil
eduardo.almeida@aluno.cefetmg.br

2nd Lucas Cerqueira Portela
Dept. Engenharia de Computação
CEFET-MG
Divinópolis, Brasil
lucas.portela@aluno.cefetmg.br

3rd Bruno Prado Santos
Dept. Engenharia de Computação
CEFET-MG
Divinópolis, Brasil
bruno.santos@aluno.cefetmg.br

***Abstract*—Este documento apresenta uma análise detalhada do algoritmo de ordenação Quick Sort, abrangendo desde a explicação do algoritmo até a apresentação dos resultados dos testes realizados. Serão discutidas as conclusões obtidas a partir dos testes e as considerações finais sobre a eficiência e aplicabilidade do Quick Sort.**

***Index Terms*—Quick Sort, Algoritmos de Ordenação, Análise de Algoritmos, Estruturas de Dados**

I. Introdução

A. O que é Ordenação?

Ordenação é o processo de arranjar um conjunto de informações semelhantes em ordem crescente ou decrescente. Ou seja, é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma relação de ordem predefinida.

Dado uma sequência de n dados:

$$< a_1, a_2, \ldots, a_n >$$

O problema de ordenação é uma permutação dessa sequência:

$$< a'_1, a'_2, \ldots, a'_n >$$

Tal que se estabeleça a seguinte ordem:

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Algumas ordens são facilmente definidas. Por exemplo, a ordem numérica, ou a ordem alfabética. Contudo, existem ordens, especialmente de dados compostos, que podem ser não triviais de se estabelecer.

Os algoritmos que ordenam um conjunto, geralmente representado em um vetor, são chamados de algoritmos de ordenação. A função desses algoritmos é promover a ordenação completa ou parcial dos elementos presentes no vetor. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade de se acessar seus dados de modo mais eficiente.

Entre os mais importantes algoritmos de ordenação, podemos citar o *bubble sort* (ou ordenação por flutuação), *heap sort* (ou ordenação por heap), *insertion sort* (ou ordenação por inserção), *merge sort* (ou ordenação por mistura) e o *quicksort*. Nesta edição iremos abordar a história do *QuickSort*.

B. QuickSort

Quicksort é um algoritmo recursivo desenvolvido por Charles Antony Richard Hoare. Tony Hoare, como também era conhecido, nasceu em Colombo, no Sri Lanka, filho de pais britânicos e graduou-se na Universidade de Oxford em 1956. Estudou tradução computacional de linguagens humanas em visita à Universidade de Moscou, lugar o qual durante os estudos, foi preciso realizar a ordenação de palavras a serem traduzidas. O quicksort foi o algoritmo desenvolvido por Hoare para ordenar as palavras, em 1960, aos 26 anos.

Este algoritmo utiliza a técnica de divisão e conquista para ordenar elementos em um array ou lista, dividindo repetidamente a lista em sub-listas menores e ordenando essas sub-listas de maneira recursiva.

O processo de ordenação do Quick Sort começa com a escolha de um elemento pivô na lista. O objetivo é reorganizar os elementos da lista de forma que todos os elementos menores que o pivô fiquem à esquerda e todos os elementos maiores fiquem à direita. Este processo de particionamento é repetido recursivamente para as sub-listas à esquerda e à direita do pivô, até que a lista esteja completamente ordenada.

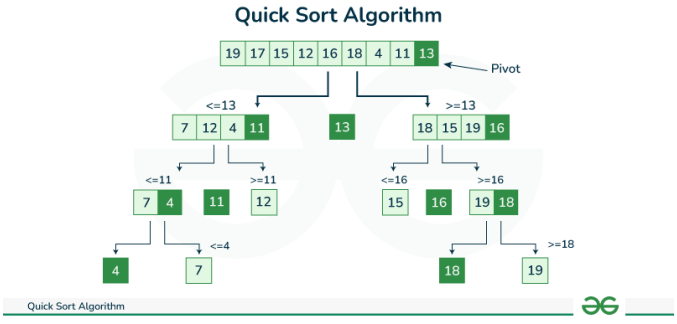

Fig. 1: Ilustração do algoritmo Quick Sort.

A eficiência do Quick Sort depende fortemente da escolha do pivô. No melhor caso, a lista é dividida de forma relativamente balanceada a cada particionamento, levando a uma complexidade de tempo de $O(n \log n)$. No pior caso, como quando a lista já está ordenada e o pivô é o menor ou o maior elemento, a complexidade pode se deteriorar para $O(n^2)$. No

entanto, com boas escolhas de pivô, o Quick Sort geralmente tem um desempenho muito bom na prática, com complexidade média de $O(n \log n)$.

II. Metodologia

O Quick Sort é um algoritmo de ordenação que segue a abordagem de divisão e conquista. Ele é conhecido por sua eficiência e é amplamente utilizado em diversos contextos. A seguir, detalharemos o funcionamento do algoritmo, ressaltando cada etapa do processo.

A. Escolha do Pivô

A escolha do pivô é a primeira etapa crucial do Quick Sort. O pivô é um elemento escolhido a partir da lista que será usado para dividir a lista em duas sub-listas. Existem várias estratégias para escolher o pivô:

- **Primeiro ou Último Elemento**: Escolher o primeiro ou o último elemento como pivô é uma abordagem simples, mas pode levar a um desempenho ruim se a lista já estiver ordenada ou quase ordenada.
- **Elemento Aleatório**: Escolher um elemento aleatório como pivô tende a fornecer um bom desempenho médio, pois distribui de forma mais uniforme as possibilidades de divisão da lista.
- **Mediana de Três**: Esta técnica envolve escolher a mediana entre o primeiro, o último e o elemento do meio da lista. Isso geralmente melhora o desempenho ao evitar piores cenários.

B. Particionamento

A etapa de particionamento reorganiza os elementos da lista de forma que todos os elementos menores que o pivô fiquem à esquerda e todos os elementos maiores fiquem à direita. Este processo é crucial para a eficiência do algoritmo. A função de particionamento pode ser descrita da seguinte forma:

1) **Inicialização**: Define-se o pivô (geralmente o último elemento da lista) e um índice para manter a posição do limite entre os elementos menores e maiores que o pivô.
2) **Iteração**: Percorre-se a lista comparando cada elemento com o pivô. Se um elemento for menor que o pivô, ele é trocado com o elemento na posição do índice de limite, e o índice é incrementado.
3) **Troca Final**: Após a iteração, o pivô é trocado com o elemento imediatamente após o último elemento menor que o pivô. Isso coloca o pivô na posição correta.

C. Estabilidade do Algoritmo

O Quick Sort não é um algoritmo estável de ordenação. Isso significa que ele pode alterar a ordem relativa de elementos com valores iguais. Em um algoritmo de ordenação estável, dois elementos com o mesmo valor permanecem na mesma ordem relativa que tinham antes da ordenação. Por exemplo, se você tiver uma lista de objetos onde o critério de ordenação é apenas um dos atributos e outros atributos determinam a ordem, uma ordenação estável preserva a ordem desses objetos com base nos atributos não utilizados.

No caso do Quick Sort, durante o particionamento, a troca de elementos não leva em conta a preservação da ordem dos elementos com o mesmo valor. Por isso, se a ordem original for importante para a aplicação, o Quick Sort pode não ser a melhor escolha, e algoritmos estáveis, como Merge Sort, podem ser preferíveis.

D. Recursão

Após o particionamento, o Quick Sort é aplicado recursivamente às sub-listas de elementos menores e maiores que o pivô. Este processo de divisão continua até que as sub-listas contenham zero ou um elemento, ponto em que elas estão naturalmente ordenadas. A recursão pode ser descrita assim:

1) **Chamada Recursiva à Esquerda**: Aplica-se o Quick Sort à sub-lista de elementos menores que o pivô.
2) **Chamada Recursiva à Direita**: Aplica-se o Quick Sort à sub-lista de elementos maiores que o pivô.

A recursão garante que, ao final do processo, a lista inteira estará ordenada. O processo é eficiente porque, em média, cada etapa de particionamento reduz a lista a metade do tamanho original.

E. Pseudo-código do Quick Sort

A seguir, apresentamos o pseudo-código do Quick Sort para ilustrar melhor seu funcionamento:

```
function quickSort(arr, low, high)
if low ¡ high then
   pivotIndex = partition(arr, low, high)
   quickSort(arr, low, pivotIndex - 1)
   quickSort(arr, pivotIndex + 1, high)
end if
end function
function partition(arr, low, high)
pivot = arr[high]
i = low - 1
for j = low to high - 1 do
   if arr[j] ¡ pivot then
      i = i + 1
      swap arr[i] and arr[j]
   end if
end for
swap arr[i + 1] and arr[high]
return i + 1
end function
```

F. Análise de Complexidade

A eficiência do Quick Sort pode ser avaliada pela sua complexidade de tempo. Em média, o Quick Sort apresenta uma complexidade de tempo de $O(n \log n)$, onde n é o número de elementos a serem ordenados. Essa eficiência média se deve ao fato de que, em cada etapa da recursão, a lista é dividida aproximadamente ao meio, resultando em um logaritmo no número total de divisões. No entanto, no pior caso, o tempo de execução pode degradar para $O(n^2)$, o que ocorre quando o

pivô escolhido resulta em partições altamente desbalanceadas, como no caso de uma lista já ordenada ou inversamente ordenada.

Para mitigar o impacto do pior caso, diversas estratégias de escolha de pivô foram desenvolvidas, como a escolha do pivô aleatório, a mediana de três elementos, ou outros métodos que tentam garantir uma divisão mais equilibrada das sub-listas. Além disso, o Quick Sort pode ser combinado com outros algoritmos de ordenação, como o Insertion Sort, para melhorar o desempenho em listas pequenas ou quase ordenadas.

A análise da complexidade do Quick Sort pode ser realizada utilizando a análise assintótica ou o Teorema Mestre. A recursão do Quick Sort é dada por:

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

onde k é o número de elementos em uma partição e $\Theta(n)$ é o tempo para a partição. No caso médio, assumimos que o pivô divide a lista em partes aproximadamente iguais, ou seja, $k \approx n/2$. Isso resulta na seguinte equação de recorrência:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Aplicando o Teorema Mestre, obtemos $T(n) = O(n \log n)$.

Além de sua eficiência, o Quick Sort é valorizado por ser um algoritmo *in-place*, ou seja, ele realiza a ordenação sem necessidade de espaço adicional significativo, o que o torna ideal para aplicações com restrições de memória. No entanto, o Quick Sort não é um algoritmo estável, o que significa que ele não preserva a ordem relativa de elementos iguais, uma característica importante em alguns contextos.

Este artigo explora o funcionamento do Quick Sort em profundidade, detalhando os aspectos teóricos e práticos do algoritmo. Serão apresentadas implementações do Quick Sort em diferentes linguagens de programação, testes realizados para avaliar sua performance em diferentes condições, e uma discussão sobre os resultados obtidos. Além disso, serão exploradas as vantagens e desvantagens do Quick Sort, fornecendo uma visão abrangente sobre um dos algoritmos de ordenação mais fundamentais e amplamente utilizados na computação.

G. Conclusão

O Quick Sort é uma técnica poderosa de ordenação que combina a simplicidade da abordagem de divisão e conquista com a eficiência na prática. Sua escolha cuidadosa do pivô e o eficiente processo de particionamento permitem que ele ordene grandes conjuntos de dados de forma rápida e eficaz.

III. Vantagens e Desvantagens do Quick Sort

O Quick Sort apresenta diversas vantagens e desvantagens, que variam conforme a aplicação específica. Entre as vantagens estão:

- **Desempenho Médio Elevado**: No caso médio, o Quick Sort possui uma complexidade de tempo de $O(n \log n)$, o que o torna eficiente para grandes volumes de dados.
- **Uso de Memória**: O Quick Sort é um algoritmo *in-place*, ou seja, ele ordena a lista sem a necessidade de espaço adicional significativo.
- **Simplicidade**: A implementação do Quick Sort é relativamente simples e pode ser otimizada de diversas formas, como a escolha adequada do pivô.
- **Paralelismo**: Devido à sua natureza recursiva, o Quick Sort pode ser paralelizado de forma eficiente, permitindo melhorias de desempenho em sistemas multi-core.

Entre as desvantagens estão:

- **Pior Caso Desfavorável**: No pior caso, a complexidade de tempo do Quick Sort é $O(n^2)$, o que pode ocorrer se o pivô escolhido resulta em partições altamente desbalanceadas.
- **Ordenação Instável**: O Quick Sort não é um algoritmo de ordenação estável, o que significa que ele não mantém a ordem relativa dos elementos com valores iguais.
- **Sensibilidade ao Pivô**: A escolha do pivô é crucial para o desempenho do Quick Sort. Escolhas inadequadas do pivô podem levar a um desempenho subótimo.
- **Recursividade**: O Quick Sort utiliza recursão, o que pode causar problemas de estouro de pilha (*stack overflow*) em listas muito grandes ou em implementações com profundidade de recursão limitada.

IV. Implementação em C, C++, C#, Java, JavaScript e Python

A seguir, apresentamos implementações do Quick Sort em C, C++, C#, Java, JavaScript e Python, ilustrando as diferenças entre as seis linguagens.

A. Implementação em C

```c
#include <stdio.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);
    }
}
```

B. Implementação em C#

```
using System;

class QuickSort {
static void Swap(ref int a, ref int b) {
    int t = a;
    a = b;
    b = t;
    }

 static 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(ref arr[i], ref arr[j]);
            }
        }
    Swap(ref arr[i + 1], ref arr[high]);
    return (i + 1);
    }



public static 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);
        }
    }
```

C. Implementação em Java

```
public class QuickSort {

void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

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, j);                }
        }
    }
    swap(arr, i + 1, 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);
    }
}
```

D. Implementação em C++

```
#include <iostream>
using namespace std;

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);
    }
}
```

E. Implementação em JavaScript

```
function swap(arr, i, j) {
    let temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
```

```
function partition(arr, low, high) {
    let pivot = arr[high];
    let i = (low - 1);
    for (let j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}

function quickSort(arr, low, high) {
    if (low < high) {
        let pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

F. Implementação em Python

```
  def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

V. Explicação dos Testes Realizados

Para avaliar a performance do algoritmo Quick Sort, realizamos uma série de testes com diferentes condições. Os detalhes dos testes são os seguintes:

A. Tamanhos de Entrada

Testamos o algoritmo com sete tamanhos diferentes de entradas:

- 10.000 elementos
- 50.000 elementos
- 100.000 elementos
- 500.000 elementos
- 1.000.000 elementos
- 2.500.000 elementos
- 5.000.000 elementos

B. Tipos de Entrada

Utilizamos três tipos diferentes de entradas para os testes:

- Vetor aleatório: Elementos são organizados de forma aleatória.
- Vetor crescente: Elementos são organizados em ordem crescente.
- Vetor decrescente: Elementos são organizados em ordem decrescente.

C. Linguagens Testadas

O algoritmo foi implementado e testado em seis linguagens de programação diferentes:

- C
- C++
- C#
- Java
- JavaScript
- Python

D. Escolha do Pivô

Os testes foram realizados com diferentes escolhas de pivô:

- **Último Elemento:** O pivô é escolhido como o último elemento do array. Similar à escolha do primeiro elemento, pode ser ineficiente para certos casos.
- **Primeiro Elemento:** O pivô é escolhido como o primeiro elemento do array. Similar à escolha do último elemento, pode ser ineficiente para certos casos.
- **Mediana de Três:** O pivô é escolhido como a mediana de três elementos: o primeiro, o último e o elemento central do array. Esta estratégia tende a melhorar o desempenho ao evitar a escolha de um pivô muito grande ou muito pequeno.

E. Configurações da Máquina

As configurações da máquina utilizada para os testes são:

- Placa-mãe: B450M
- Placa de vídeo: RTX 4060
- Processador: Ryzen 5 5600X
- Memória RAM: 16GB 2660MHz
- Fonte: Corsair 550W
- Armazenamento: SSD 1.5TB Kingston

Os testes foram realizados executando o algoritmo Quick Sort com as diferentes condições descritas acima. O tempo de execução foi medido e comparado entre as diferentes configurações para avaliar a eficiência do algoritmo em diversos cenários.

VI. Apresentação dos Resultados

A. *Desempenho por Tamanho de Entrada*

Para avaliar o desempenho do Quick Sort em relação ao tamanho da entrada, realizamos testes com diferentes tamanhos de vetores: 10.000, 50.000, 100.000, 500.000, 1.000.000, 2.500.000 e 5.000.000 elementos. Os tempos de execução foram medidos e comparados, conforme ilustrado na Tabela I e nas Figuras 2 e 3.
Obs: Nesse caso, optamos pela escolha da mediana de 3 para escolha do pivô.

TABLE I: Média Geral dos Tempos de Execução do Quick Sort por Tamanho de Entrada

Tamanho da Entrada	C	C++	C#	Java	JavaScript	Python
10.000	6.1235	11.9504	24.7343	5.3733	30.73	134.69
50.000	35.5395	68.8767	79.7967	28.6833	79.1667	784.29
100.000	75.2049	145.296	148.8531	60.159	147.3333	1514.99
500.000	424.8163	500.976	755.6978	312.7833	1045.5667	7513.82
1.000.000	873.8015	1812.734	1689.3213	661.1833	1967.1667	19700.56
2.500.000	2267.2364	4728.74	4094.1227	1743.74	5306.867	56157.23
5.000.000	4526.4884	7615.937	7474.284	3589.53	10566.23	131048.42

Observamos que, conforme o tamanho da entrada aumenta, os tempos de execução também aumentam significativamente para todas as linguagens. Python apresentou o maior tempo de execução em todas as entradas, refletindo sua menor eficiência em comparação com as outras linguagens compiladas. Em contraste, C e Java apresentaram tempos de execução consistentemente baixos, destacando-se como as linguagens mais eficientes para a implementação do Quick Sort neste experimento. A diferença de desempenho entre as linguagens pode ser atribuída a vários fatores, incluindo a forma como cada linguagem gerencia a memória e otimiza a execução do código.

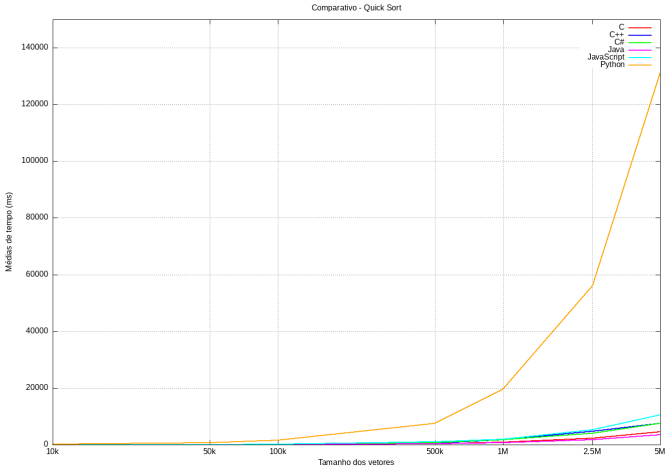

Fig. 2: Tempo de Execução do Quick Sort por Tamanho de Entrada

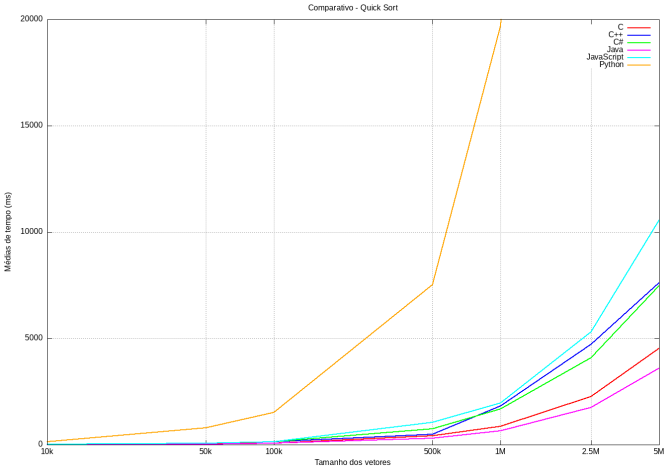

Fig. 3: Tempo de Execução do Quick Sort por Tamanho de Entrada com Zoom

B. Desempenho por Tipo de Entrada

Os testes foram realizados com três tipos diferentes de entrada: vetor aleatório, vetor em ordem crescente e vetor em ordem decrescente. Os resultados mostraram variações significativas no desempenho.
Obs: Nesse caso optamos pela mediana de 3 como escolha do pivô.

TABLE II: Tempo de Execução do Quick Sort por Tipo de Entrada (Crescente)

Tamanho da Entrada	C	C++	C#	Java	JavaScript	Python
10.000	1.1499	3.2215	43.84	1.1865	1.6742	49.27
50.000	6.6073	19.2881	101.17	6.5755	74.7294	303.98
100.000	14.189	39.805	152.9	12.3157	81.4378	649.99
500.000	87.4817	231.725	660.78	92.8922	174.1392	3893.59
1.000.000	201.9419	523.587	1241.52	168.5312	336.0143	10388.64
2.500.000	570.816	1465.86	3413.04	478.077	813.2468	36172.41
5.000.000	1255.316	3071.88	6969.6	1021.3195	1709.9804	82266.44

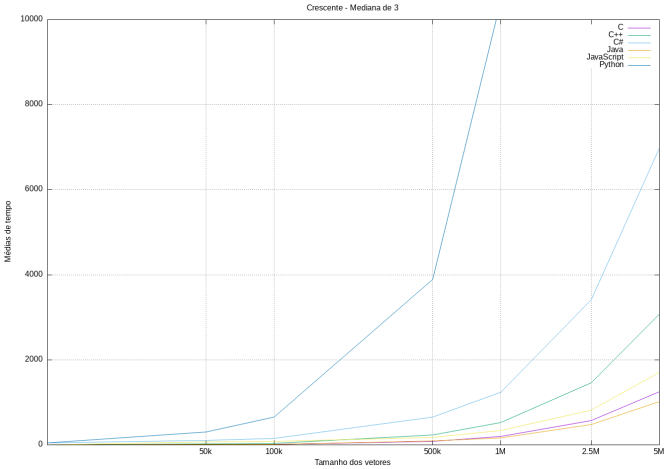

Fig. 4: Tempo de Execução do Quick Sort por Tipo de Entrada (Crescente)

TABLE III: Tempo de Execução do Quick Sort por Tipo de Entrada (Decrescente)

Tamanho da Entrada	C	C++	C#	Java	JavaScript	Python
10.000	1.207	3.757	37.14	1.0441	1.7358	50.67
50.000	7.1162	22.5035	93.19	6.2399	70.8074	298.25
100.000	14.6536	48.752	154.18	12.3558	84.6442	652.31
500.000	90.2157	283.982	583.58	86.5467	184.681	3902.31
1.000.000	206.4362	630.471	1410.76	167.6188	344.283	10237.47
2.500.000	599.8243	1738.17	3577.71	484.6027	843.0705	35551.24
5.000.000	1271.2741	3684.4	7224.61	942.5659	1742.7324	81433.27

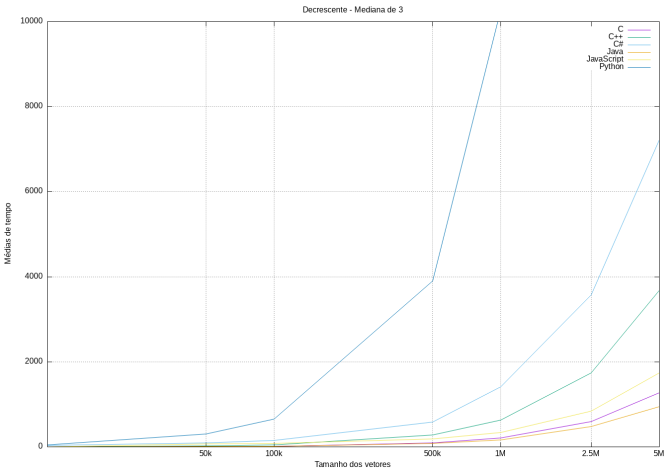

Fig. 5: Tempo de Execução do Quick Sort por Tipo de Entrada (Decrescente)

TABLE IV: Tempo de Execução do Quick Sort por Tipo de Entrada (Aleatório)

Tamanho da Entrada	C (ms)	C++ (ms)	C# (ms)	Java (ms)	JavaScript (ms)	Python (ms)
10.000	6.1168	10.8824	58.6	4.16	6.1879	208.2
50.000	35.5361	65.2536	149.4	24.97	33.2	1243.13
100.000	74.8063	135.555	260.9	65.11	129.7213	2522.29
500.000	427.782	761.097	1219.9	307.49	448.4903	14411.74
1.000.000	879.1351	1610.62	2559.7	653.61	869.5241	30191.25
2.500.000	2260.0139	4119.92	6459.85	1644.03	2232.8291	81665.34
5.000.000	4576.1624	8550.41	12035.95	3424.11	4621.9317	170965.33

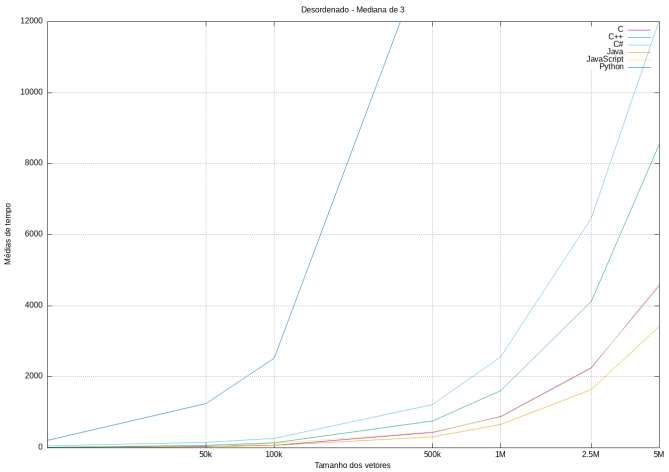

Fig. 6: Tempo de Execução do Quick Sort por Tipo de Entrada (Desordenado)

C. Comparação entre Linguagens

Implementamos o Quick Sort em seis linguagens de programação diferentes: C, C++, C#, Java, JavaScript e Python. Vamos mostrar os tempos e gráficos de cada linguagem em separado.
Obs: Nesses casos optamos por usar a mediana de 3 para escolha do Pivô.

TABLE V: Tempo de Execução do Quick Sort em C por Tipo de Entrada

Tamanho da Entrada	Desordenado (ms)	Crescente (ms)	Decrescente (ms)
10.000	6.1168	1.1499	1.2070
50.000	35.5361	6.6073	7.1162
100.000	74.8063	14.1890	14.6536
500.000	427.7820	87.4817	90.2157
1.000.000	879.1351	201.9419	206.4362
2.500.000	2260.0139	570.8160	599.8243
5.000.000	4576.1624	1255.3160	1271.2741

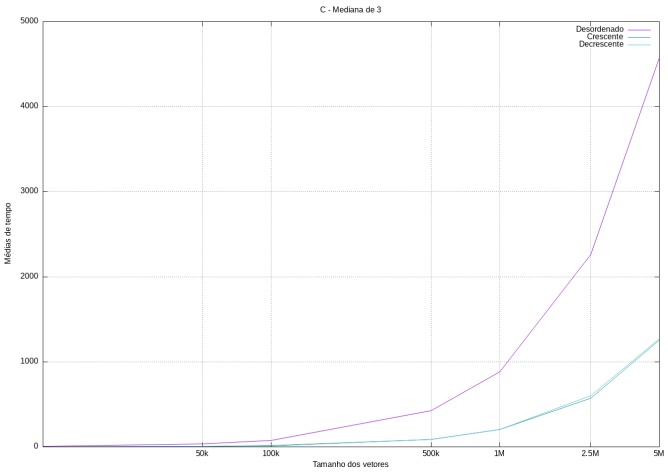

Fig. 7: Tempo de Execução do Quick Sort por Tipo de Entrada (C)

TABLE VI: Tempo de Execução do Quick Sort em C# por Tipo de Entrada

Tamanho da Entrada	Crescente (ms)	Desordenado (ms)	Decrescente (ms)
10.000	43.84	58.60	37.14
50.000	101.17	149.40	93.19
100.000	152.90	260.90	154.18
500.000	660.78	1219.90	583.58
1.000.000	1241.52	2559.70	1410.76
2.500.000	3413.04	6459.85	3577.71
5.000.000	6969.60	12035.95	7224.61

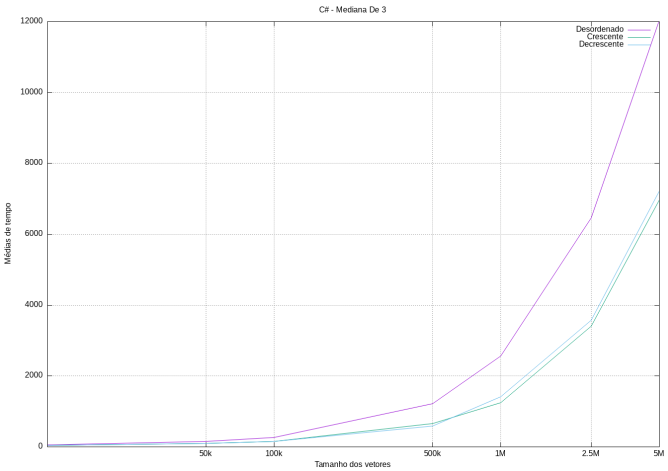

Fig. 8: Tempo de Execução do Quick Sort por Tipo de Entrada (C#)

TABLE VII: Tempo de Execução do Quick Sort em C++ por Tipo de Entrada

Tamanho da Entrada	Crescente (ms)	Desordenado (ms)	Decrescente (ms)
10.000	3.2215	10.8824	3.7570
50.000	19.2881	65.2536	22.5035
100.000	39.8050	135.5550	48.7520
500.000	231.7250	761.0970	283.9820
1.000.000	523.5870	1610.6200	630.4710
2.500.000	1465.8600	4119.9200	1738.1700
5.000.000	3071.8800	8550.4100	3684.4000

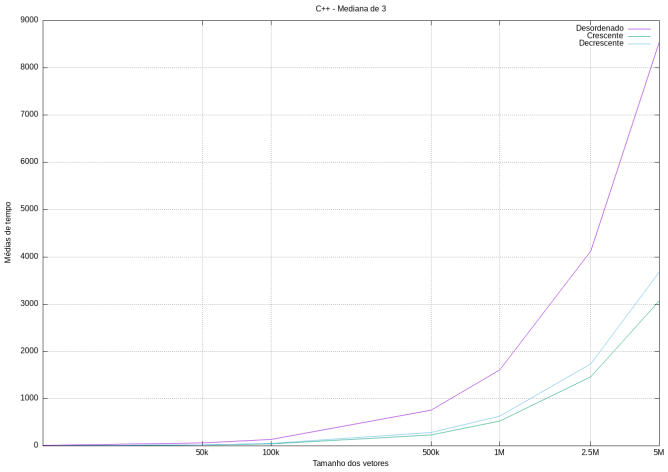

Fig. 9: Tempo de Execução do Quick Sort por Tipo de Entrada (C++)

TABLE VIII: Tempo de Execução do Quick Sort em Java por Tipo de Entrada

Tamanho da Entrada	Crescente (ms)	Desordenado (ms)	Decrescente (ms)
10.000	1,1865	4,16	1,0441
50.000	6,5755	24,97	6,2399
100.000	12,3157	65,11	12,3558
500.000	92,8922	307,49	86,5467
1.000.000	168,5312	653,61	167,6188
2.500.000	478,0770	1644,03	484,6027
5.000.000	1021,3195	3424,11	942,5659

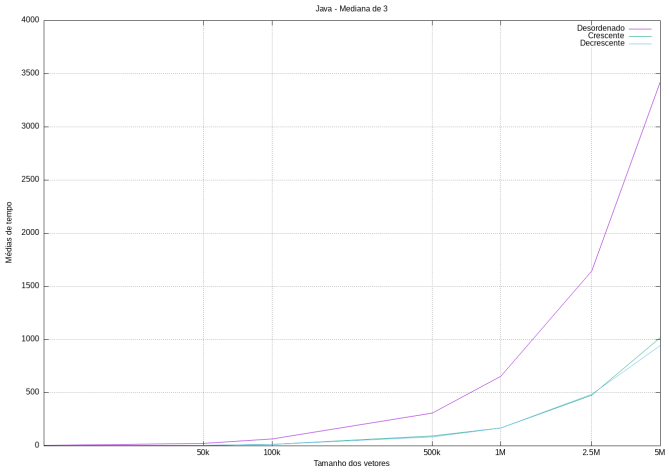

Fig. 10: Tempo de Execução do Quick Sort por Tipo de Entrada (Java)

TABLE IX: Tempo de Execução do Quick Sort em JavaScript por Tipo de Entrada

Tamanho da Entrada	Crescente (ms)	Desordenado (ms)	Decrescente (ms)
10.000	1.6742	6.1879	1.7358
50.000	74.7294	33.2	70.8074
100.000	81.4378	129.7213	84.6442
500.000	174.1392	448.4903	184.6810
1.000.000	336.0143	869.5241	344.2830
2.500.000	813.2468	2232.8291	843.0705
5.000.000	1709.9804	4621.9317	1742.7324

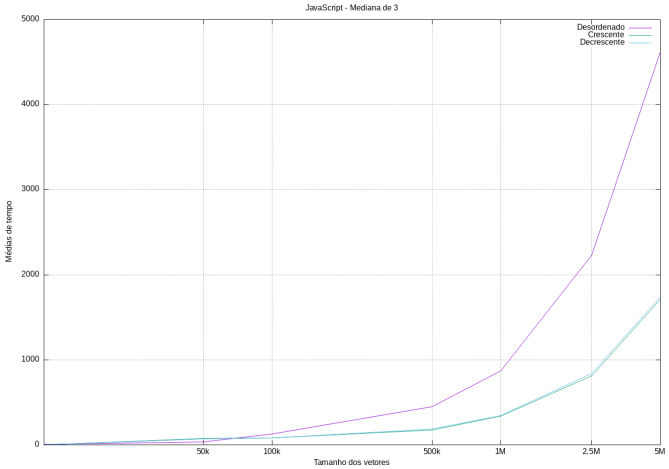

Fig. 11: Tempo de Execução do Quick Sort por Tipo de Entrada (JavaScript)

TABLE X: Tempo de Execução do Quick Sort em Python por Tipo de Entrada

Tamanho da Entrada	Crescente (ms)	Desordenado (ms)	Decrescente (ms)
10.000	49.27	208.20	50.67
50.000	303.98	1243.13	298.25
100.000	649.99	2522.29	652.31
500.000	3893.59	14411.74	3902.31
1.000.000	10388.64	30191.25	10237.47
2.500.000	36172.41	81665.34	35551.24
5.000.000	82266.44	170965.33	81433.27

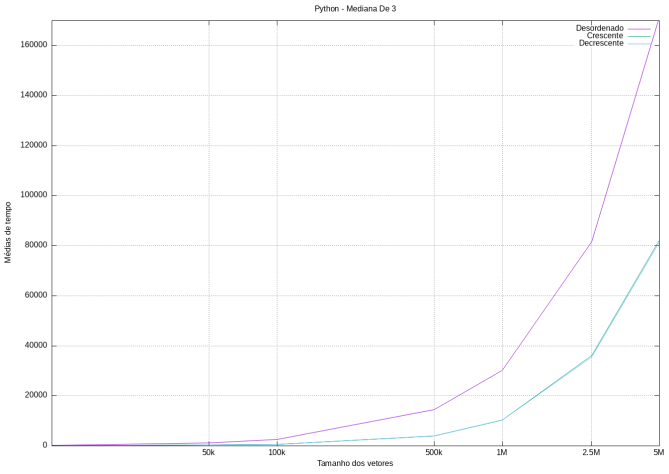

Fig. 12: Tempo de Execução do Quick Sort por Tipo de Entrada (Python)

D. Comparação de Métodos de Escolha do Pivô

Para avaliar o impacto da escolha do pivô no desempenho do Quick Sort, implementamos três métodos diferentes de seleção do pivô na linguagem C: pivô no início, pivô como mediana de três elementos e pivô no final. Realizamos testes com entradas desordenadas, crescentes e decrescentes, medindo os tempos de execução para cada método. Os resultados são apresentados nas Figuras 16, 17, e 18.

De acordo com os estudos teóricos sobre essa diferença de escolha do pivô, esparava-se o seguinte:

- **Pivô no Início:** Este método tende a ser menos eficiente, especialmente para entradas ordenadas (crescente ou decrescente), pois leva ao pior caso de complexidade O(n²) com maior frequência. No entanto, para entradas desordenadas, o desempenho é comparável aos outros métodos.
- **Pivô como Mediana de Três:** Este método geralmente oferece uma melhor performance, pois seleciona um pivô mais representativo do conjunto de dados. Isso reduz a probabilidade de ocorrer o pior caso, levando a uma complexidade média mais próxima de O(n log n). Observamos um desempenho mais consistente e eficiente em todos os tipos de entrada.
- **Pivô no Final:** Similar ao pivô no início, este método também pode levar ao pior caso de complexidade O(n²) para entradas ordenadas. Entretanto, para entradas desordenadas, o desempenho é aceitável. Em geral, a escolha do pivô no final mostra tempos de execução ligeiramente melhores do que o pivô no início, mas não tão bons quanto a mediana de três.

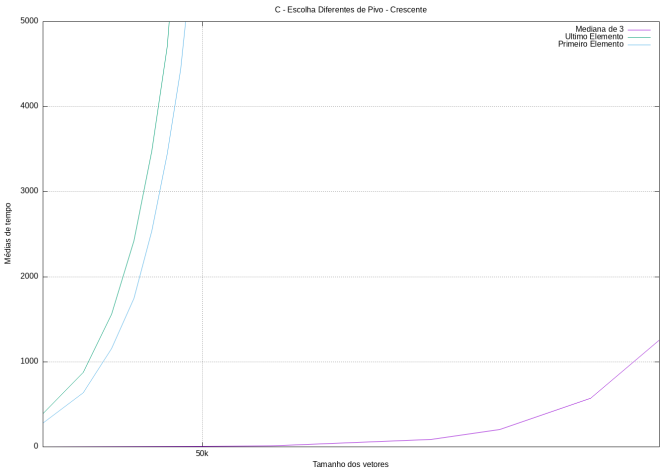

Fig. 13: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor Crescente (C)

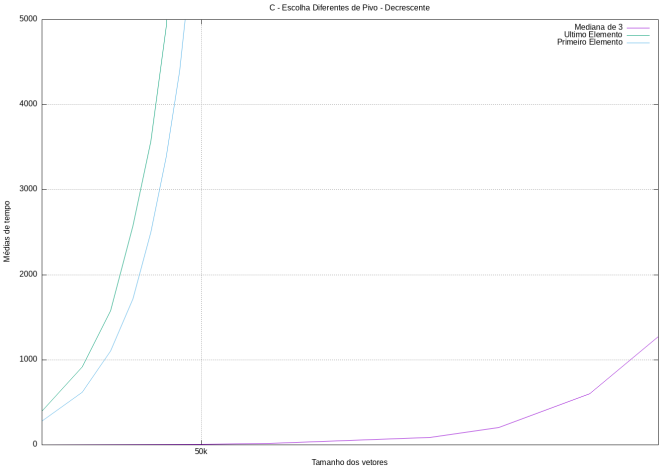

Fig. 14: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor Decrescente (C)

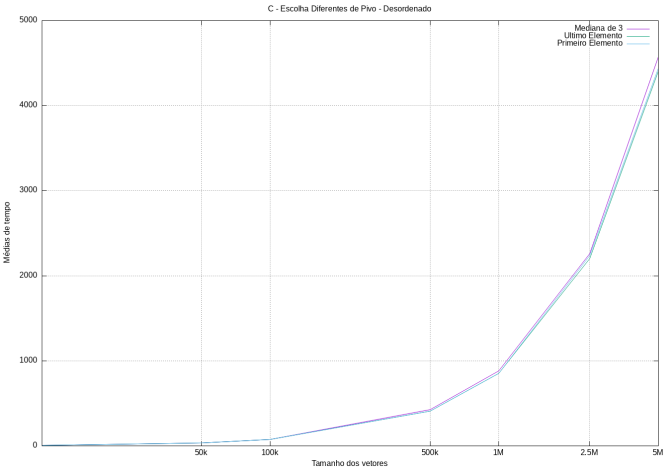

Fig. 15: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor desordenada (C)

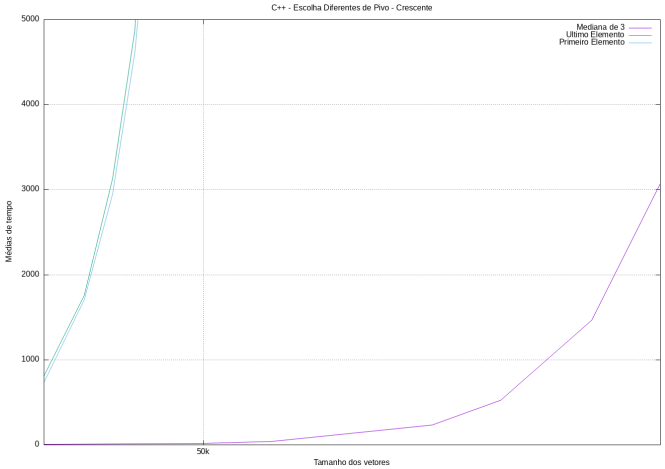

Fig. 16: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor Crescente (C++)

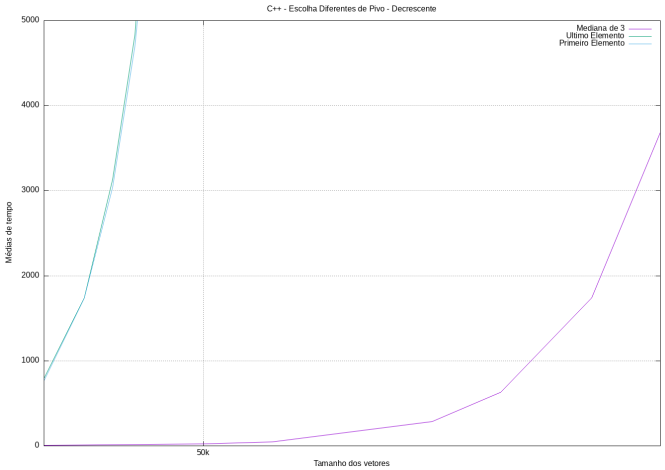

Fig. 17: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor Decrescente (C++)

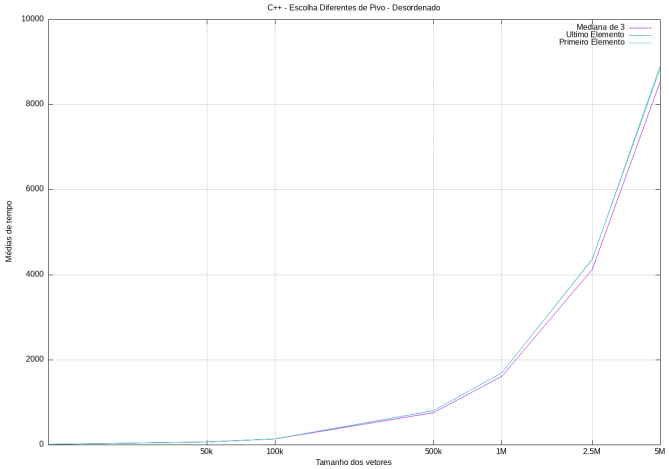

Fig. 18: Tempo de Execução do Quick Sort por Tipo de Entrada com os 3 Tipos de Pivô e o Vetor desordenada (C++)

Os gráficos apresentados corroboram a teoria previamente discutida, demonstrando que o algoritmo QuickSort alcança seu pior desempenho ao tentar ordenar vetores já ordenados, seja em ordem crescente ou decrescente. Assim, a seleção do pivô revela-se de extrema importância. Observa-se, nos gráficos, uma diferença substancial nos resultados, evidenciando o desempenho superior quando se utiliza a técnica da mediana de três para a escolha do pivô. Essa abordagem resulta em uma melhoria significativa, transformando o algoritmo de uma situação de pior caso para um desempenho equivalente ao caso médio ou até mesmo ao melhor caso. Quando se trata de um vetor desordenado, o comportamento das escolhas do pivô se torna bem semlhante, mas tendo visto todo o contexto, a mediana de 3 se torna de longe a escolha do pivô mais interessante.

VII. Conclusões sobre os Testes

Os testes realizados para avaliar o desempenho do algoritmo Quick Sort revelaram várias conclusões importantes:

- **Tamanho e Tipo de Entrada:** O desempenho do Quick Sort varia significativamente com o tipo e tamanho da entrada. Entradas já ordenadas, seja de forma crescente ou decrescente, resultam no pior caso de complexidade O(n²) para certas escolhas de pivô. Em contrapartida, entradas aleatórias tipicamente apresentam tempos de execução mais favoráveis, próximos da complexidade média O(n log n). Com o aumento do tamanho das entradas, essas diferenças de desempenho se tornam mais pronunciadas, enfatizando a importância de uma escolha criteriosa do pivô.
- **Escolha do Pivô:** A seleção do pivô geralmente mostra ter um impacto direto na eficiência do algoritmo. A técnica da mediana de três proporciona um desempenho mais consistente e eficiente, reduzindo a probabilidade do pior caso. Com isso, os nossos resultados práticos mostraram que as diferenças entre as várias técnicas de escolha do pivô foram fundamentais e mudam muito o resultado, quando se trata de vetores já ordenados.
- **Linguagem de Programação:** A linguagem de programação utilizada teve uma influência significativa nos tempos de execução. Linguagens compiladas como C e Java apresentaram os menores tempos de execução, enquanto Python teve o maior tempo, refletindo sua menor eficiência relativa. Diferenças na gestão de memória, otimizações do compilador e características específicas de cada linguagem contribuem para essas variações de desempenho. Por exemplo, as otimizações específicas do compilador em C podem explicar sua superioridade em termos de performance. Podemos ver isso muito bem, quando a maioria das linguagens varia bastante de acordo com o tipo do vetor, mas o C mesmo nesses casos conseguiu se manter muito constante, uusando tanto vetores ordenado quanto desordenados.
- **Eficiência Geral:** No geral, C e Java se destacaram como as linguagens mais eficientes para a implementação do Quick Sort neste experimento, enquanto Python apresentou o maior tempo de execução para todas as entradas. Isso ressalta a importância de escolher a linguagem de programação apropriada para aplicações que exigem alta performance, especialmente ao lidar com grandes volumes de dados.

Essas conclusões sublinham a importância de considerar múltiplos fatores ao avaliar a eficiência do Quick Sort. Futuros estudos podem explorar mais a fundo como diferentes combinações de fatores influenciam o desempenho do algoritmo. Investigações adicionais poderiam focar em testes com variações de hardware, cargas de trabalho mais diversificadas, e otimizações específicas para cada linguagem de programação para fornecer uma visão ainda mais detalhada e prática sobre a eficiência do Quick Sort em diferentes cenários.

VIII. Considerações Finais

O Quick Sort se mantém como um dos algoritmos de ordenação mais eficientes e rápidos, porém seu desempenho é influenciado por diversos fatores. Nossa análise demonstrou que o tipo e tamanho da entrada, a escolha do pivô e a linguagem de programação são determinantes para a eficiência do algoritmo.

Entradas ordenadas resultam no pior caso de complexidade $O(n^2)$ para certas escolhas de pivô, enquanto entradas desordenadas apresentam tempos de execução mais favoráveis, próximos de $O(n \log n)$. A estratégia da mediana de três pivôs geralmente se destaca por proporcionar um desempenho mais consistente e eficiente.

A linguagem de programação também impacta significativamente o desempenho, com linguagens compiladas como C e Java mostrando maior eficiência em comparação a linguagens interpretadas como Python. Além disso, os resultados sugerem que outros fatores, como a base de dados utilizada e as condições de execução, também influenciam o desempenho.

Em resumo, o Quick Sort é uma escolha excelente para ordenação, mas seu desempenho pode ser otimizado através da escolha cuidadosa do pivô e da linguagem de programação, considerando o contexto específico de uso.

Agradecimentos

Agradecemos a todos os colaboradores e financiadores que tornaram este trabalho possível.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[2] D. E. Knuth, *The Art of Computer Programming*, Vol. 3: Sorting and Searching, 2nd ed. Addison-Wesley, 1998.

[3] R. Sedgewick, *Algorithms in C*, 3rd ed. Addison-Wesley, 1998.

[4] R. S. Mello, *Algoritmos de Ordenação: QuickSort*. [Online]. Available: https://edisciplinas.usp.br/pluginfile.php/4123108/mod_resource/content/1/ICC2_Aula_QuickSort.pdf

[5] A. Bari, "Quick Sort Algorithm," YouTube, Jun. 20, 2018. [Online]. Available: https://www.youtube.com/watch?v=nV_WE8SEuGE&t=354s