Uma análise comparativa entre os principais algoritmos de ordenação

Henrique G. Oliveira[1], **Eduardo H. M. Cruz**[1]

[1]Instituto Federal do Paraná (IFPR) – Campus Paranavaí

henriquegon.o@hotmail.com, eduardo.cruz@ifpr.edu.br

Abstract. *Due to the need to have sorted data, several algorithms emerged with the objective of solving this problem, each with its particularity and time to reach the solution. Thus, this study aims to perform a comparative analysis of the main sorting algorithms, namely: Insertion Sort, Selection Sort, Bubble Sort, Quick Sort and the Merge Sort, observing the difference in the time that each algorithm takes to sort the same data.*

Resumo. *Devido a necessidade de ordenar dados, surgiram diversos algoritmos com o objetivo de solucionar esse problema, cada um com sua particularidade e tempo para alcançar a solução. Desta forma, este estudo objetiva realizar uma análise comparativa dos principais algoritmos de ordenação, sendo eles: Insertion Sort, Selection Sort, Bubble Sort, Quick Sort e o Merge Sort, observando a diferença no tempo que cada algoritmo demora para ordenar os mesmos dados.*

1. Introdução

A ordenação é considerada um dos principais problemas no que se refere à algoritmos [Cormen et al. 2022]. Os algoritmos de ordenação consistem em reorganizar os valores para que o resultado final tenha todos os dados de forma crescente ou decrescente. Pode acabar passando despercebido, porém, diversas vezes durante o dia a dia surge a necessidade de ter algum dado ordenado, desde simplesmente realizar uma busca na *internet* ordenando os dados pelos mais relevantes, a até mesmo sua utilização em bancos de dados, compiladores, interpretadores e sistemas operacionais [Schildt 2021]. Com isto, é perceptível a importância de se estudar algoritmos de ordenação, entendendo os prós e contras de cada algoritmo.

Neste contexto, o presente trabalho tem como objetivo realizar uma análise comparativa de alguns dos principais algoritmos de ordenação, sendo eles: *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Quick Sort* e o *Merge Sort*. Os algoritmos serão apresentados através de pseudocódigos e serão analisados por meio de diversos testes, com o objetivo de observar a diferença no tempo que cada algoritmo demora para ordenar a mesma quantidade de dados.

Este trabalho está estruturado da seguinte forma. Na Seção 2, são apresentados os trabalhos relacionados, da mesma forma que é introduzido a teoria da complexidade e as notações utilizadas nos pseudocódigos. Na Seção 3, é apresentado cada algoritmo de ordenação que será comparado neste trabalho. A Seção 4 descreve os métodos para obtenção e análise dos dados referente ao tempo que cada algoritmo demora para ordenar. Na Seção 5, são apresentados os resultados obtidos através dos testes. Por fim, na Seção 6, são apresentadas as conclusões acerca dos dados analisados.

2. Fundamentação teórica

Nesta seção, serão discutidos os trabalhos correlatos, contextualizando o presente trabalho e suas características diante das diversas outras pesquisas relacionadas. Além de introduzir a teoria da complexidade, entendendo o que é a complexidade de tempo e espaço, serão apresentadas as notações utilizadas no desenvolvimento dos pseudocódigos.

2.1. Teoria da complexidade

Diante da infinidade de algoritmos existentes, definir se um determinado algoritmo resolve o problema para o qual foi proposto, de forma que seja mais eficiente que outros algoritmos, acaba se tornando uma tarefa complicada. Desta forma, uma maneira para calcular a eficiência de um algoritmo é medir sua complexidade através da quantidade necessária de processamento e armazenamento para sua execução [Bovet et al. 1994]. Porém, quando falamos sobre medir a complexidade de um algoritmo, não nos referimos a quantidade demandada de processamento e armazenamento empiricamente, e sim de forma teórica.

Neste trabalho, serão apresentadas as complexidades tanto de espaço quanto de tempo.

2.1.1. Complexidade de tempo

A complexidade de tempo determina quanto tempo demora para um determinado algoritmo completar sua tarefa, baseado na quantidade de operações durante sua execução. Com isso, cada operação soma para a expressão da complexidade, de forma que até um código simples, dependendo do tamanho da sua entrada, começa a crescer significativamente o seu tempo de execução.

2.1.2. Complexidade de espaço

A complexidade de espaço funciona de forma semelhante à complexidade de tempo. Porém, neste critério o que é somado para a expressão da complexidade são as variáveis que são armazenadas.

2.1.3. Classificando as complexidades

Para classificar a eficiência de um algoritmo, deve-se analisar os casos do algoritmos: melhor caso, médio caso e pior caso. Para representar a complexidade de cada caso, neste trabalho, será utilizada a anotação Theta(θ). A notação θ estabelece ao mesmo tempo um limite assintótico inferior e superior para o algoritmo.

Seguem algumas complexidades que aparecem com bastante frequência:

- $\theta(1)$ – constante – o tempo/espaço necessário independe do tamanho da entrada;
- $\theta(n)$ – linear – o tempo/espaço cresce linearmente com o tamanho da entrada;
- $\theta(n^2)$ – quadrática – o tempo/espaço cresce quadraticamente com o tamanho da entrada;
- $\theta(log(n))$ – logarítmica;
- $\theta(n * log(n))$.

2.2. Notação dos pseudocódigos

Para os pseudocódigos, serão utilizadas as seguintes notações:

- **arr** – (*array*) é a sequência de elementos a ser ordenada.
- **arr.length** representa a quantidade de elementos do *array*.
- o laço de repetição e o arr iniciam com o valor 1.
- elementos de um *array* são acessados utilizando o nome do *array*, como arr, seguido pelo índice entre colchetes. Por exemplo, **arr[i]**, onde i pode ser qualquer valor de 1 até arr.length.
- e a estrutura para os laços de repetição **for**, **while**, e os condicionais **if-else** são semelhantes às utilizadas nas linguagens de programação como Python, C, C++, Java e diversas outras.

3. Estudo dos principais algoritmos de ordenação

Esta seção tem como objetivo apresentar os principais algoritmos propostos no trabalho, que são o: *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Quick Sort* e o *Merge Sort*. O objetivo é demonstrar como cada algoritmo resolve o problema da ordenação, através de pseudocódigos, para que os algoritmos sejam fáceis de compreender e explicar.

3.1. *Insertion sort*

O *Insertion Sort* é um dos algoritmos de ordenação mais simples e eficiente quando se é necessário organizar uma pequena quantidade de elementos [Cormen et al. 2022]. Segundo [Schildt 2021], a complexidade de tempo para está ordenação é $\theta(n^2)$ na maioria dos casos. Porém, em um caso raro, quando todos os elementos já estão ordenados, sua complexidade de tempo é $\theta(n)$. A complexidade de espaço é $\theta(1)$. A execução desta ordenação, presente no Algoritmo 1, assemelha-se à organização das cartas de um baralho, ou seja, para realizar a ordenação é selecionada uma carta por vez, e a mesma é colocada em seu devido lugar, mantendo sempre as cartas da mão em ordem.

Algorithm 1 *Insertion Sort*(arr)

```
for iteration = 2, 3, ... arr.length do
    value = arr[iteration]
    previous = iteration - 1
    while value > 0 and arr[previous] > value do
        arr[previous + 1] = arr[previous]
        previous = previous - 1
    end while
    arr[previous + 1] = value
end for
```

O algoritmo apresentado no pseudocódigo funciona da seguinte forma: quando a ordenação inicia, $value$ guarda o valor que irá ser ordenado no momento, já o $previous$ guarda o elemento na posição anterior a $value$, ou seja, a ordenação começa com o valor na segunda posição, então $previous$ guarda a primeira posição, e assim sucessivamente. Enquanto o $value$ for maior que 0 e o valor de $previous$ for maior que $value$, os números que são menores que $value$ vão ficando para esquerda dele, e os maiores são movidos para direita. Na Figura 1, é apresentado um exemplo da execução deste algoritmo.

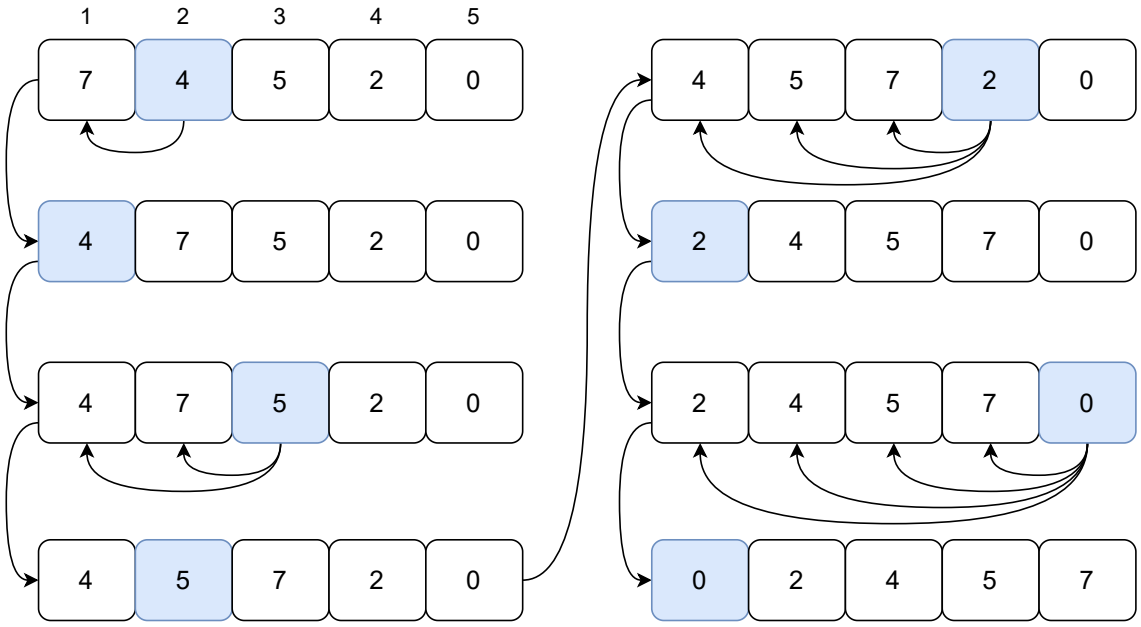

Figura 1. Exemplo de execução do *Insertion Sort*.

Na Figura 1, estão presentes 5 números, onde cada número ocupa uma posição, indo de 1 até 5 e, como foi visto anteriormente, o *Insertion Sort* funciona sempre comparando um valor com o seu anterior. Portanto, a ordenação começará na segunda posição que contém o valor 4 e irá comparar com a primeira posição que contem o valor 7, como 4 é menor que 7, os valores são trocados de lugar. Porém, como não existe nenhum outro valor antes dele para ser comparado, o algoritmo vai para a terceira posição que contem o valor 5, e assim como na *iteração* anterior, ele irá ser comparado com os valores anteriores que são 7 e 4, como 5 é menor que 7 eles irão ser trocados de posição, porém 5 não é menor que 4 então o algoritmo vai para a próxima *iteração*. Desta forma, será seguida a mesma lógica para o resto dos valores, sempre avançando uma posição no final da *iteração* e comparando o valor desta posição com os anteriores, de forma que os valores menores vão sempre se deslocando para o início.

3.2. *Selection Sort*

O *Selection Sort* realiza a ordenação selecionando o elemento com o menor valor em cada *iteração*, e vai movendo para o início. Sua complexidade de tempo para todos os casos acaba sendo $\theta(n^2)$, o que torna está ordenação ineficiente quando se é preciso ordenar uma grande quantidade de dados [Schildt 2021]. Sua complexidade de espaço é $\theta(1)$. O Algoritmo 2 apresenta como funciona a execução desta ordenação.

Este algoritmo ordena os dados guardando a posição do valor a ser movido em $minIndexValue$, e percorrendo o resto de todo o arr procurando o menor valor para preencher aquela posição. Portanto, para a primeira posição do arr que está sendo guardada em $minIndexValue$, ele irá encontrar o menor valor e realizará a troca. Para a segunda posição será feita a mesma coisa, e assim sucessivamente. A Figura 2 apresenta um exemplo da execução deste algoritmo.

Na Figura 2, o algoritmo pega a primeira posição que contém o valor 7, e passará

por todos os valores procurando o menor valor presente neste **arr**. Quando encontrado, que neste caso é o valor 0, eles são trocados de lugar. Com isso, é passado para próxima *iteração*. Como a primeira posição já tem o menor valor, será encontrado o segundo menor valor para a segunda posição, que atualmente contem o valor 4, ao percorrer novamente toda o **arr** é encontrado o valor 2 que é trocado com o 4, e passado para a próxima *iteração*. Desta forma, o algoritmo avança as posições e encontrando o menor valor para ela, até que todos os valores estejam ordenados.

Algorithm 2 *Selection Sort*(arr)

```
for iteration = 1, 2, ... arr.length do
    minIndexValue = iteration
    for value = interation + 1... arr.length do
        if arr[value] < arr[minIndexValue] then
            minIndexValue = value
        end if
    end for
    temporary = arr[iteration]
    arr[iteration] = arr[minIndexValue]
    arr[minIndexValue] = temporary
end for
```

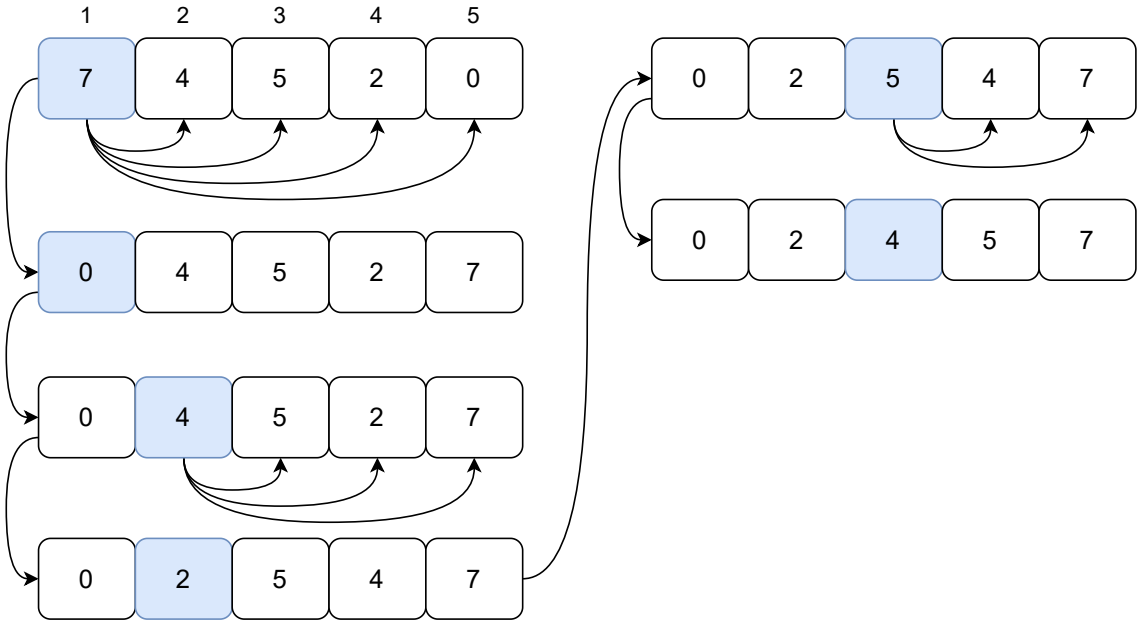

Figura 2. Exemplo de execução do *Selection Sort*

3.3. *Bubble Sort*

O *Bubble Sort* é um algoritmo de ordenação bem popular graças à sua simplicidade [Schildt 2021]. Contudo, ele é ineficiente [Cormen et al. 2022], devido a sua complexidade de tempo ser $\theta(n^2)$ para todos os casos. Sua complexidade de espaço é $\theta(1)$. A execução desta ordenação pode ser vista no Algoritmo 3 e na Figura 3.

Esta ordenação funciona sempre comparando e trocando dois valores de lugar, ou seja, é selecionado o primeiro valor e comparado com o segundo. Se o primeiro valor for maior que o segundo, eles são trocados, e vai para a próxima *iteração*, onde é comparado o segundo valor com o terceiro, e caso seja necessário é realizada a troca, e assim por diante. O efeito é que os valores seguem flutuando até o final do laço.

Algorithm 3 *Bubble Sort*(arr)

```
for i = 1, 2, ... arr.length do
    for j = 1, 2, ... arr.length - i - 1 do
        if arr[j] > arr[j + 1] then
            temporary = arr[j]
            arr[j] = arr[j + 1]
            arr[j + 1] = temporary
        end if
    end for
end for
```

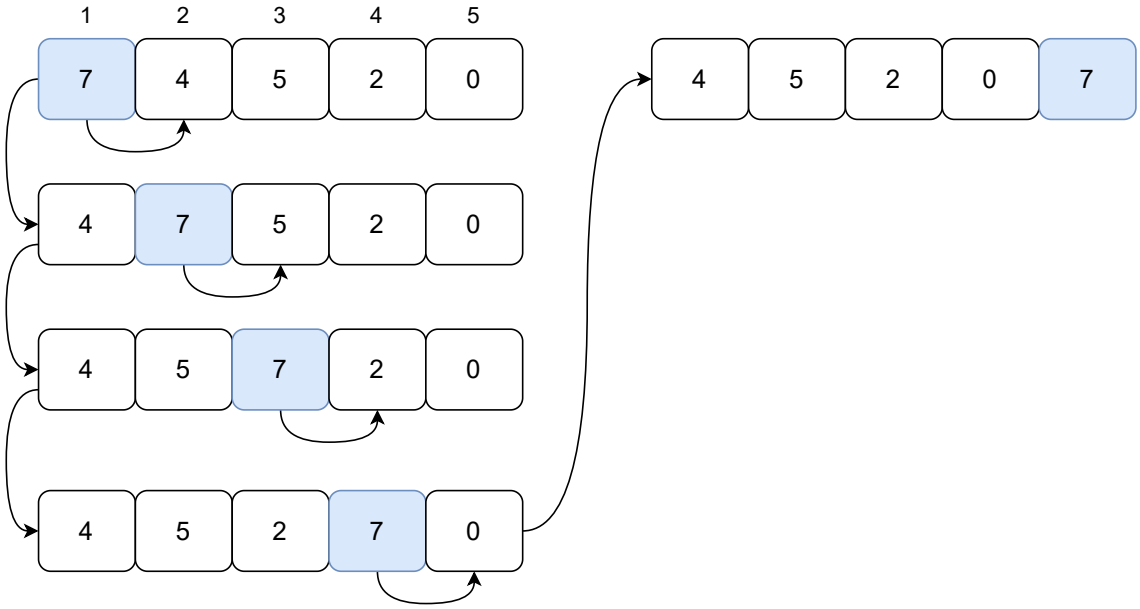

Figura 3. Exemplo de execução do *Bubble Sort*.

Na Figura 3, o algoritmo seleciona o valor da primeira posição, que é o 7, e compara com o da segunda posição que é o 4. Como 7 é maior que 4, eles são trocados de posição. Com isso, a *iteração* continua, comparando o 7, que está na terceira posição, com o 5. Como 7 é maior que 5, os valores são trocados, e assim sucessivamente, até que o 7 vire o último valor do **arr**. O algoritmo irá repetir o processo levando sempre os maiores valores pro final, deixando assim os valores ordenados.

3.4. *Quick Sort*

O *Quick sort* comumente é considerado o melhor algoritmo de ordenação com um propósito geral [Schildt 2021]. Está ordenação se caracteriza por se dividir em várias outras partes, que são chamadas de partições, até que cada parte tenha apenas um elemento,

então todas partes são combinadas recursivamente resultando nos elementos ordenados. Seu desempenho vária conforme a escolha do $pivot$. Se o $pivot$ for o elemento do meio do arr, ou próximo do meio, a complexidade de tempo no melhor caso será $\theta(n * log(n))$, já o pior caso que é $\theta(n^2)$, ocorre quando o $pivot$ escolhido se encontra em alguma extremidade. O caso médio é $\theta(n * log(n))$. Sua complexidade de espaço é $\theta(1)$. Os Algoritmos 4 e 5 apresentam o funcionamento desta ordenação. Na Figura 4, é demonstrado um exemplo da execução dos algoritmos.

Algorithm 4 partition(arr, start, end)

```
pivot = arr[start]
left = start
right = end
while left < right do
    while left ⩽ pivot do
        left = left + 1
    end while
    while right > pivot do
        right = right - 1
    end while
    if left < right then
        temporary = arr[left]
        arr[left] = arr[right]
        arr[right] = temporary
    end if
end while
arr[start] = arr[right]
arr[right] = pivot
return right
```

Algorithm 5 *Quick Sort*(arr, start, end)

```
if start < end then
    p = partition(arr, start, end)
    Quick Sort(arr, start, p - 1)
    Quick Sort(arr, p + 1, end)
end if
```

Para facilitar o entendimento do *Quick Sort*, sua execução foi separada em duas partes, uma responsável por separar recursivamente o **arr** em outras partes, e uma para ordenar os elementos. No início da ordenação, é calculado o $pivot$ chamando a segunda parte do algoritmo, que denominada de **partition**, passando como valor inicial $start = 0$ e $end = arr.length$. Com o $pivot$ definido, é chamado recursivamente mais duas vezes o *Quick Sort*, uma vez chamado para os elementos que estão antes do $pivot$, ou seja, de $start$ até $pivot - 1$, e à outra vez é chamado para os elementos depois do $pivot$, iniciando em $pivot + 1$ até o final do **arr**.

A função do **partition** é percorrer o **arr** enquanto o valor de $left$ for menor que $right$, incrementando $left$ toda vez que seu valor for menor que o $pivot$, assim como

decrementando o valor de $right$ enquanto seu valor for maior que $pivot$. Desta forma, irá passar por todo o **arr**, da esquerda até o $pivot$, e da direita até o $pivot$, para no final desse processo verificar se $left$ é menor que $right$. Caso seja, são trocados os valores de lugar, e também é feita a troca de valores onde $arr[start]$ recebe o último valor da direita, que é $right$, e a posição da direita se torna o $pivot$, para então ser retornado à posição da direita.

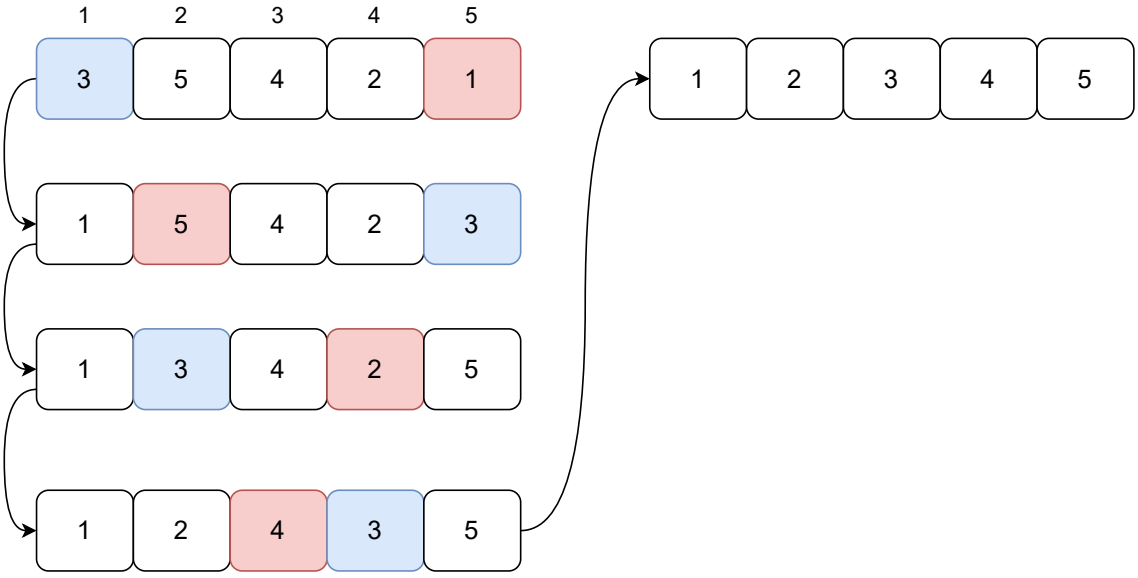

Figura 4. Exemplo de execução do *Quick Sort*, algoritmo partition.

Na Figura 4, para exemplificar a execução do algoritmo, os valores utilizados como base foram alterados para deixar a explicação mais simples e direta. Partindo para sua execução, o número 3 que está na primeira posição foi escolhido como *pivot*. Com isso, é procurado à sua direita um número menor que ele para ser passado para a sua esquerda. O primeiro número menor que o *pivot* encontrado foi o 1, então eles trocam de lugar. Agora é procurado um número à sua esquerda que seja maior que ele. O primeiro número maior encontrado foi o 5, portanto, eles trocam de lugar. O mesmo processo do início acontece, e o menor número encontrado foi o 2. Com isso, eles trocam de lugar e, em seguida, é repetido o processo do passo 2, onde o maior número encontrado é o 4, e eles trocam de lugar.

Este processo refere-se apenas à execução do Algoritmo 4. Após a execução dele, o algoritmo se divide recursivamente à esquerda e à direita do *pivot*.

3.5. *Merge Sort*

O *Merge Sort*, assim como o *Quick Sort*, é um algoritmo que divide recursivamente o conjunto de dados. O *Merge Sort* faz chamadas recursivas até que cada subconjunto possua apenas 1 elemento e, no final, cada subconjunto é reunido para formar os elementos ordenados. A complexidade de tempo deste algoritmo é $\theta(n * log(n))$ para todos os casos. Sua complexidade de espaço é $\theta(n)$, visto que ele precisa de um espaço de armazenamento adicional para combinar os sub-vetores. A execução desta ordenação pode ser vista nos Algoritmos 6 e 7, e um exemplo da sua execução pode ser visto na Figura 5.

A execução *Merge Sort* foi dividida em duas partes: uma responsável por separar

recursivamente os elementos, que pode ser vista no Algoritmo 7, e a outra responsável por ordenar, que está presente no Algoritmo 6. No início da ordenação, é calculado o valor para encontrar a posição do meio do **arr**. Com isso, é chamado novamente o algoritmo *Merge Sort*, porém, agora passando do começo do **arr** até o meio, e do meio + 1 até o fim, e por fim é chamado o **merge** para combinar as metades.

Algorithm 6 merge(arr, temp, start, mid, end)

```
length = end - start + 1
firstHalf = start
secondHalf = mid + 1
fhEnd = false, shEnd = false
for i = 1, 2, ... length do
    if !fhEnd && !shEnd then
        if arr[firstHalf] < arr[secondHalf] then
            temp[i] = arr[firstHalf++]
        end if
        temp[i] = arr[secondHalf++]
        if firstHalf > mid then
            fhEnd = true
        end if
        if secondHalf > end then
            shEnd = true
        end if
    end if
    if !fhEnd then
        temp[i] = arr[firstHalf++]
    end if
    temp[i] = arr[secondHalf++]
end for
for j = 1, 2, ... length do
    k = start
    arr[k] = temp[j]
    k++
end for
```

Algorithm 7 *Merge Sort*(arr, temp, start, end)

```
mid = null
if start < end then
    mid = start + end / 2
    Merge Sort(arr, temp, start, mid)
    Merge Sort(arr, temp, mid + 1, end)
    merge(arr, temp, start, mid, end)
end if
```

Dentro da função responsável por ordenar os elementos, é calculado o tamanho do **arr**, assim como definido em *firstHalf* e *secondHalf*, os conjuntos que serão combinados

no array temporário **temp**. Com isso, é percorrido todo o **temp**, e para cada posição é verificado quem é o menor elemento dentro de *firstHalf* e *secondHalf* para ser colocado nesta posição. Caso seja alcançado o final de uma das partes, que é identificada pelo *fhEnd* e *shEnd*, então não é mais necessário realizar a comparação, a parte que sobrou é copiada para **temp**. No final desta *iteração*, é percorrido todo o array temporário **temp**, que tem seus valores copiados para **arr**.

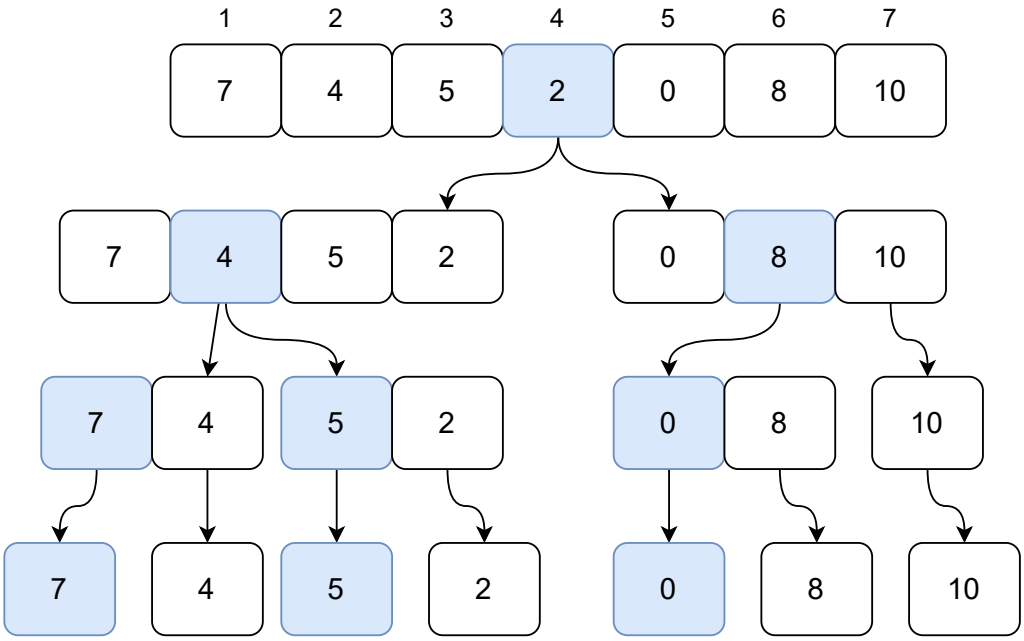

Figura 5. Exemplo de execução do *Merge Sort*.

Na Figura 5, é demonstrado como o *Merge Sort* vai calculando o meio do **arr**, que inicialmente foi a posição 4, deixando 3 elementos para cada lado, e com isso criando subconjuntos e recalculando o seu meio até que exista apenas um elemento por subconjunto. Ao final desse passo, começa a parte de **merge**, ou seja, combinar os elementos em pares de forma ordenada, resultando, assim em um conjunto ordenado no final.

4. Metodologia de avaliação

Para avaliação do tempo de ordenação de cada algoritmo, os códigos e testes foram realizados em uma máquina virtual utilizando a linguagem de programação C++. Na Tabela 1, é apresentada a configuração das máquinas física e virtual.

Além dos códigos também foram criados 30 arquivos com dados de entrada, onde cada arquivo contém os valores de um teste, totalizando 30 testes por algoritmo. Cada arquivo contém 5.000 números inteiros de valores aleatórios, desta forma, mantém-se os mesmos valores dos testes para todos os algoritmos, e consegue-se observar a diferença de tempo que cada algoritmo demora para ordenar os mesmos dados. Com isso, os resultados de cada algoritmo são salvos em um arquivo csv, e utilizando a biblioteca pandas [1], é gerado um gráfico que contém a média de cada algoritmo e o intervalo de confiança de 95% utilizando a distribuição t de Student.

[1] https://pandas.pydata.org/

Tabela 1. *Hardware* das máquinas utilizada para os testes.

Máquinas	Parâmetro	Valores
Física	Processador	4 cores, Intel Core, i5-6400, 2.7GHz, 14nm
	Cache L1	4 x 32 KB 8-way set associative instruction caches
		4 x 32 KB 8-way set associative data caches
	Cache L2	4 x 256 KB 4-way set associative caches
	Cache L3	6 MB 12-way set associative shared cache
	Memória Principal	16GB DDR4-2133MHz 15-15-15-35
Virtual	Processadores	4
	Memória Principal	8GB
	Ambiente	Ubuntu 20.04, Linux kernel 5.15, GCC 11.2
	Compilação	com otimização -O2

5. Resultado

Como visto na Seção 4, para realização dos testes, foram gerados 30 arquivos contendo 5.000 valores aleatórios, que foram utilizados como entrada para cada um dos 30 testes de cada algoritmo. O gráfico da Figura 6 apresenta os resultados destes testes em forma de linha, exibindo as médias de tempo em que cada algoritmo demora para ordenar de 1.000 à 5.000 elementos e o intervalo de confiança.

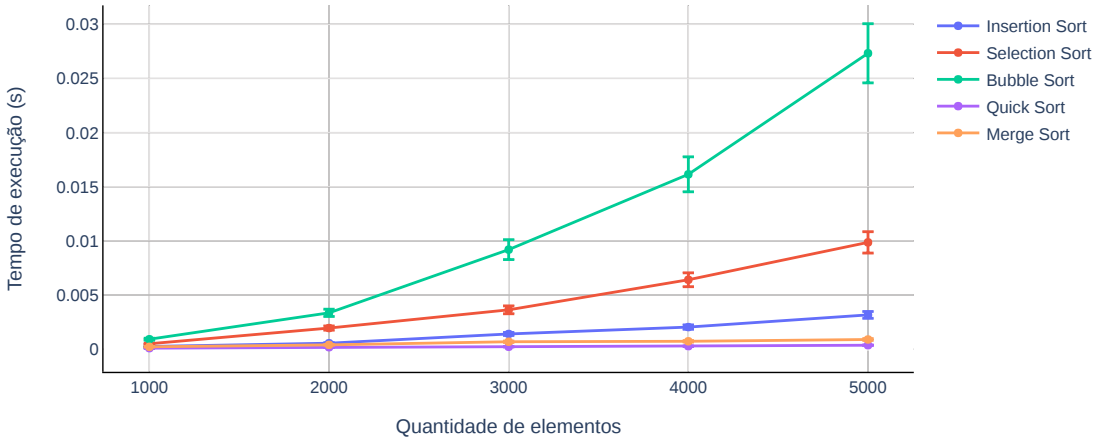

Figura 6. Tempo médio de execução dos algoritmos (em segundos).

No gráfico é possível observar que os algoritmos com uma complexidade quadrática, sendo eles: *Insertion Sort*, *Selection Sort* e *Bubble Sort*, crescem significativamente seu tempo de execução conforme a quantidade de elementos vai aumentando. Inicialmente todos os algoritmos ordenam até 1.000 elementos com tempos relativamente próximos, sendo o *Bubble Sort* o mais lento entre eles já de começo. Ainda assim, conforme os elementos aumentam para 2.000 e 3.000, é possível observar um crescimento

razoável do *Selection Sort* e do *Bubble Sort*, enquanto o *Insertion Sort* mostrou apenas um pequeno crescimento no seu tempo de execução.

Já para os 4.000 e 5.000 elementos o *Insertion Sort* se mostrou o mais rápido entre os algoritmos com a complexidade quadrática, devido ao seu pequeno número de comparações quando comparado com os outros algoritmos. Ao mesmo tempo que o *Bubble Sort* continuou sendo o algoritmo com o pior tempo e demonstrando o maior crescimento.

Enquanto esses 3 algoritmos demonstram um aumento significativo no tempo de execução conforme o número de elementos, os algoritmos *Quick Sort* e *Merge Sort* mantiveram seu tempo de execução com valores sempre próximos, não demonstrando uma grande diferença. Isto se da por utilizarem a estratégia de divisão e conquista, ao contrário dos outros algoritmos que vão sempre comparando os números em pares.

6. Conclusão

É notável a importância dos algoritmos de ordenação para a programação. Entender como cada algoritmo funciona e quais suas diferenças pode afetar positivamente em todo o desenvolvimento. Através de testes, é possível ter uma noção de qual algoritmo usar em determinada situação, pois cada algoritmo realiza a ordenação de uma forma, alguns utilizando mais processamento do que outros, assim como diferentes implementações de um mesmo algoritmo podem gerar resultados melhores ou piores. Como visto na Seção 5, algoritmos mais simples, com uma complexidade quadrática, se tornam extremamente lentos quando comparados com algoritmos que tem uma complexidade menor. Porém, o *hardware* utilizado na realização dos testes também deve ser levado em consideração, pois a configuração da máquina interfere no desempenho dos algoritmos. Os mesmos testes realizados em configurações melhores, gerariam resultados mais rápidos. Com isso, algoritmos como: *Insertion Sort*, *Selection Sort* e *Bubble Sort* são mais recomendados quando é necessário ordenar uma pequena quantidade de dados, pois seu tempo de execução cresce quadraticamente. Já algoritmos recursivos como o *Quick Sort* e o *Merge Sort* são recomendados para ordenar quantidades de valores na casa dos milhões/bilhões, devido a sua estratégia de divisão e conquista. Deste modo, um estudo que visa implementar e testar os diferentes algoritmos se mostra essencial para entender a importância e a diferença entre cada algoritmo de ordenação. Como trabalho futuro pode ser feito uma implementação e análise com outros algoritmos como: *Radix Sort*, *Bucket Sort*, *Shell Sort*, *Heap Sort*. Também pode ser realizado os mesmos testes utilizando uma configuração de *hardware* diferente, comparando os tempos dos algoritmos.

Referências

Bovet, D. P., Crescenzi, P., and Bovet, D. (1994). *Introduction to the Theory of Complexity*. Prentice Hall London.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

Schildt, H. (2021). C the complete reference.