Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Pedro Lemos Mariano - 6968

ALGORITMOS DE ORDENAÇÃO

Rio Paranaíba - MG

2024

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Pedro Lemos Mariano - 6968

ALGORITMOS DE ORDENAÇÃO

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2024

1 RESUMO

...

Sumário

2 INTRODUÇÃO

Os algoritmos são fundamentais na ciência da computação, pois fornecem métodos sistemáticos para resolver problemas e otimizar processos. Eles consistem em sequências de instruções que orientam a execução de tarefas em diversas áreas, como programação e análise de dados. Este trabalho visa explorar a importância do desenvolvimento e análise de algoritmos, com ênfase em técnicas clássicas, como o Bubble Sort, Insertion Sort, Merge Sort, Selection Sort, Shell Sort e Quick Sort que ilustram a busca por eficiência em operações computacionais [2].

A análise dos algoritmos abordará a funcionalidade básica de cada um, os contextos de aplicação mais comuns e as características que os tornam relevantes no estudo da ordenação. Além disso, será discutida a complexidade de cada algoritmo, considerando tanto o tempo de execução em seus melhores, piores e casos médios, quanto o espaço adicional necessário para sua execução. A complexidade de tempo e espaço dos algoritmos será abordada, com destaque para os seguintes pontos:

- **Bubble Sort**: Complexidade $O(n^2)$ no pior caso e caso médio, sendo ineficiente para grandes conjuntos de dados, mas simples de implementar.
- **Insertion Sort**: Complexidade $O(n^2)$ no pior caso, mas eficiente ($O(n)$) para entradas quase ordenadas, com baixo consumo de memória adicional.
- **Merge Sort**: Complexidade $O(n \log n)$ em todos os casos, destacando-se pela eficiência e estabilidade, porém com maior uso de memória.
- **Selection Sort**: Complexidade $O(n^2)$ em todos os casos, com um número fixo de trocas de elementos, o que pode ser vantajoso em alguns contextos.
- **Shell Sort**: Complexidade variável dependendo da sequência de incrementos, podendo variar entre $O(n^2)$ e $O(n \log^2 n)$, proporcionando melhorias sobre o Insertion Sort em vetores maiores.
- **Quick Sort**: Complexidade $O(n \log n)$ no melhor e no caso médio, mas $O(n^2)$ no pior caso, geralmente quando o pivô é mal escolhido. Apesar disso, é muito eficiente na

prática devido à sua abordagem de divisão e conquista e é amplamente usado para grandes conjuntos de dados.

O objetivo deste estudo é entender tanto a teoria por trás desses algoritmos quanto suas aplicações práticas, preparando-nos para os desafios da computação moderna. Além da aplicação de cada algoritmo em contextos reais, serão obtidos dados de tempo e complexidade para análise posterior. Os algoritmos serão submetidos a ordenações crescentes, decrescentes e aleatórias com vetores de tamanho 10, 100, 1.000, 10.000, 100.000 e 1.000.000 elementos, permitindo uma avaliação detalhada da eficiência de cada abordagem em diferentes cenários.

Através dos testes realizados, busca-se destacar não apenas a eficiência computacional de cada técnica, mas também suas limitações e possíveis contextos práticos de uso de cada um dos algoritmos.

3 ALGORITMOS

3.1 Insertion Sort

O algoritmo **Insertion Sort** é um método de ordenação de elementos de uma lista, ou vetor, inserindo cada item em sua posição correta em relação aos itens anteriormente ordenados. A ideia central é dividir o vetor em duas partes: a primeira é ordenada e começa com o primeiro elemento, enquanto a segunda parte contém os elementos ainda a serem classificados. A cada iteração, o próximo elemento da parte desordenada é comparado com os elementos da parte ordenada e inserido na posição adequada. Esse processo continua até que todos os elementos estejam organizados, como na imagem de exemplo [1].

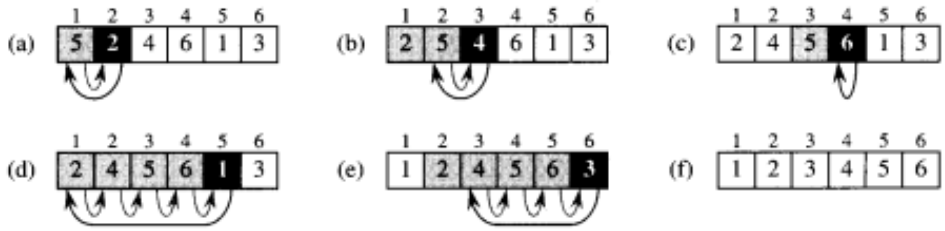

Figura 1: Exemplo de funcionamento [2]

O algoritmo é útil para listas pequenas e quase ordenadas, pois possui uma implementação simples, sendo um algoritmo in-place. Além disso, por ser estável, o Insertion Sort mantém a ordem relativa de elementos iguais, o que é vantajoso em contextos onde essa característica é importante. Como já é sabido o funcionamento do algoritmo e como suas comparações e trocas são feitas, uma exemplificação de um meio de implementação em linguagem C como na figura [2].

```c
for (int i = 1; i < tamanho; i++) {
    int chave = vetor[i];
    int j = i - 1;
    while (j >= 0 && vetor[j] > chave) {
        vetor[j + 1] = vetor[j];
        j--;
    }
    vetor[j + 1] = chave;
}
```

Figura 2: Estrutura do código em C [Feito pelo Autor]

3.2 Selection Sort

O algoritmo **Selection Sort** é um método de ordenação de elementos de uma lista, ou vetor, com a metodologia de comparação de todo o array, o algoritmo faz essa comparação geral, e como o próprio nome ja diz, seleciona o menor valor e coloca na posição inicial, a paritir disso, vai ordenando todo o resto, selecionando o 2° menor, e assim por diante. Caso seja necessário, o algoritmo também pode já fornecer o **maior** elemento, que deverá se encontrar na posição de maior extremidade (direita), como mosrtado na imagem de exemplo [3].

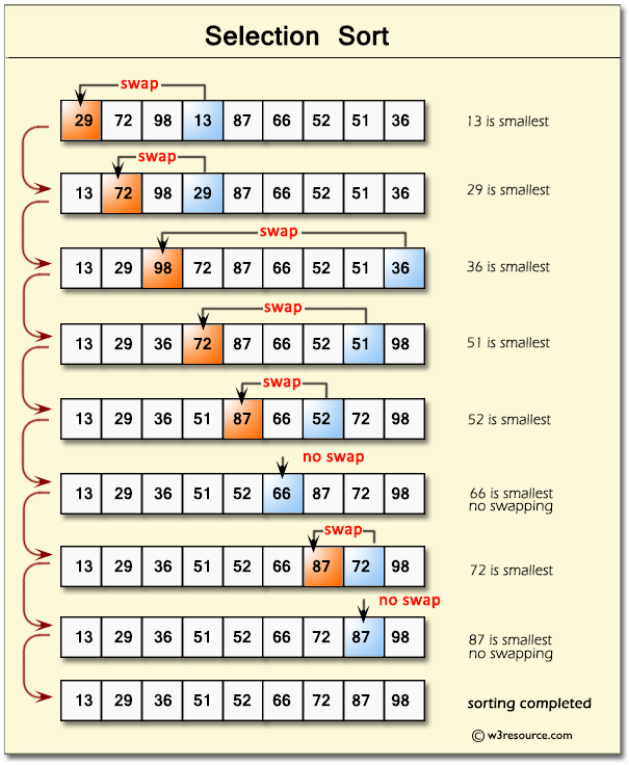

Figura 3: Exemplo de funcionamento [5]

O algoritmo também é útil para listas pequenas e quase ordenadas, pois possui uma implementação simples. No entanto, ao contrário do Insertion Sort, o Selection Sort não é estável, ou seja, ele pode alterar a ordem relativa de elementos iguais. Isso pode ser uma desvantagem em contextos onde a estabilidade é importante. Como já é sabido o funcionamento do algoritmo e como suas comparações e trocas são feitas, uma exemplificação de um meio de implementação em linguagem C é mostrada na figura [4]:

```c
for (int i = 0; i < tamanho - 1; i++) {
    int min_index = i;
    for (int j = i + 1; j < tamanho; j++) {
        if (vetor[j] < vetor[min_index]) {
            min_index = j;
        }
    }
    if (min_index != i) {
        int temp = vetor[i];
        vetor[i] = vetor[min_index];
        vetor[min_index] = temp;
    }
}
```

Figura 4: Estrutura do código em C [Feito pelo Autor]

3.3 Shell Sort

O algoritmo **Shell Sort**, é uma versão otimizada do Insertion Sort. [6] A ideia é que ele não compara e troca elementos apenas adjacentes, mas usa uma sequência de distâncias maiores, chamadas "gaps", que vão diminuindo ao longo do algoritmo até chegar a 1. Começa com uma distância maior, como metade do tamanho do array, e vamos comparando e trocando elementos que estão separados por essa distância. Depois, diminui-se o gap e repete o processo. Quando o gap chega a 1, o algoritmo faz uma espécie de Insertion Sort, mas agora em uma lista que já está quase ordenada, o que torna o processo bem mais rápido, como evidenciado na figura de exemplo [5].

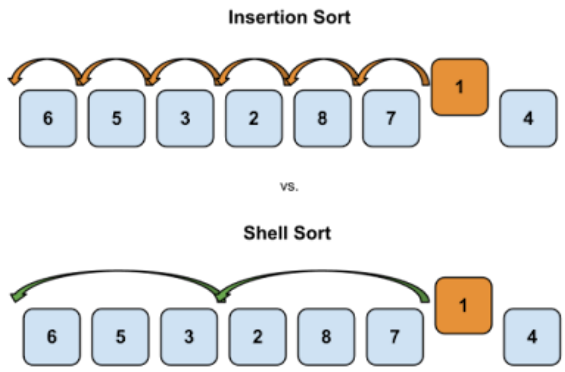

Figura 5: Exemplo de funcionamento [4]

O algoritmo é mais rápido que o Insertion Sort, principalmente em arrays grandes e com muitos elementos desordenados, por conta de seus gaps, que evitam movimentos pequenos de números que ja estão pertos de sua devida ordem. O Shell Sort é um algoritmo in-place, ou seja, não precisa de muita memória extra para funcionar, pois ordena os elementos dentro do próprio array. Como já é sabido o funcionamento do algoritmo e como suas comparações e trocas são feitas, uma exemplificação de um meio de implementação em linguagem C é mostrada na figura [6].

```c
for (int intervalo = tamanho / 2; intervalo > 0; intervalo /= 2) {
    for (int i = intervalo; i < tamanho; i++) {
        int temp = vetor[i];
        int j;
        for (j = i; j >= intervalo && vetor[j - intervalo] > temp; j -= intervalo) {
            vetor[j] = vetor[j - intervalo];
        }
        vetor[j] = temp;
    }
}
```

Figura 6: Estrutura do código em C [Feito pelo Autor]

3.4 Bubble Sort

O algoritmo **Bubble Sort**, é um dos mais básicos e é frequentemente usado para introduzir conceitos de ordenação. Ele trabalha através de comparações entre elementos adjacentes, trocando-os se estiverem fora de ordem e "empurrando" o maior elemento para o final da lista em cada passagem. [2] O algoritmo passa repetidamente pela lista até que nenhuma troca seja necessária, indicando que a lista está ordenada, [6] como é exemplificado na figura [7].

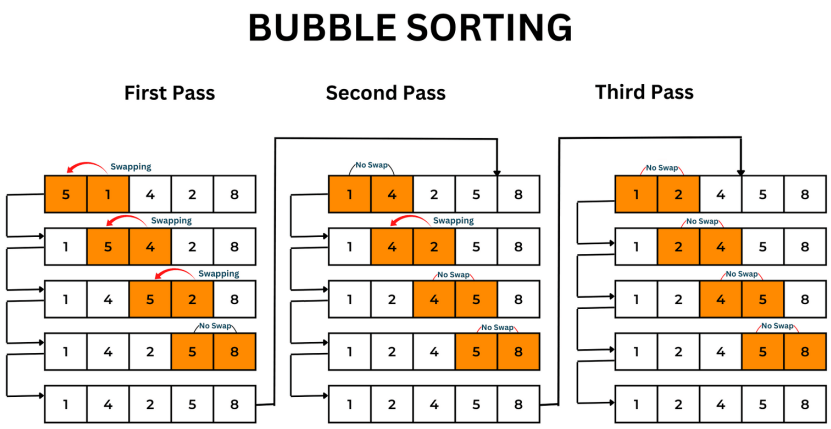

Figura 7: Exemplo de funcionamento [3]

O algoritmo é de uma fácil implementação , sendo um bom algoritmo para introdução no ensino desse tema, contudo, ineficiente para listas grandes e cenários com um volume muito grande de dados. Como já é sabido o funcionamento do algoritmo e como suas comparações e trocas são feitas, uma exemplificação de um meio de implementação em linguagem C é mostrada na figura [8].

```c
for (int i = 0; i < tamanho - 1; i++) {
    for (int j = 0; j < tamanho - i - 1; j++) {
        if (vetor[j] > vetor[j + 1]) {
            int temp = vetor[j];
            vetor[j] = vetor[j + 1];
            vetor[j + 1] = temp;
        }
    }
}
```

Figura 8: Estrutura do código em C [Feito pelo Autor]

Em um teste de mesa, nota-se todo o porcesso desse algoritmo, desde sua comparação em busca de um numero maior, até toda a ordenação do vetor. Para o teste de mesa em questão, usaremos a seguinte sequência: **5 - 3 - 8 - 4 - 6**

Teste de Mesa			
i	**j**	Se **A[j]** > **A[j+1]**	Então **A[j]** $\leftarrow\rightarrow$ **A[j+1]**
1	1	A[1] > A[2] (5>3)	A[1] $\leftarrow\rightarrow$ A[2] A[1] (5) $\leftarrow\rightarrow$ A[2] (3) Nova sequência: ***3 - 5 - 8 - 4 - 6***
1	2	A[2] > A[3] (5>8)	Mesma sequência anterior: ***3 - 5 - 8 - 4 - 6***, pois o ***if* = falso**
1	3	A[3] > A[4] (8>4)	A[3] $\leftarrow\rightarrow$ A[4] A[3] (8) $\leftarrow\rightarrow$ A[4] (4) Nova sequência: ***3 - 5 - 4 - 8 - 6***
1	4	A[4] > A[5] (8>6)	A[4] $\leftarrow\rightarrow$ A[5] A[4] (8) $\leftarrow\rightarrow$ A[5] (6) Nova sequência: ***3 - 5 - 4 - 6 - 8***
2	1	A[1] > A[2] (3>5)	Mesma sequência anterior: ***3 - 5 - 4 - 6 - 8***, pois o ***if* = falso**
2	2	A[2] > A[3] (5>4)	A[2] $\leftarrow\rightarrow$ A[3] A[2] (5) $\leftarrow\rightarrow$ A[3] (4) Nova sequência: ***3 - 4 - 5 - 6 - 8***
2	3	A[3] > A[4] (5>6)	Mesma sequência anterior: ***3 - 4 - 5 - 6 - 8***, pois o ***if* = falso**
2	4	A[4] > A[5] (6>8)	Mesma sequência anterior: ***3 - 4 - 5 - 6 - 8***, pois o ***if* = falso**

3.5 Merge Sort

O algoritmo **Merge Sort** é um dos mais eficientes entre os algoritmos de ordenação que utilizam a abordagem de divisão e conquista. Ele divide recursivamente a lista em sublistas menores até que cada sublista contenha apenas um elemento, momento em que elas são consideradas ordenadas. Em seguida, essas sublistas são mescladas de maneira ordenada para formar listas maiores e, finalmente, a lista completa. [2] Por sua natureza, o Merge Sort possui uma complexidade de tempo $O(n \log n)$ em todos os casos, o que o torna adequado para grandes conjuntos de dados, embora requeira espaço adicional para as operações de mesclagem [6], para melhor compreensão é exeplificado esse processo na figura [9].

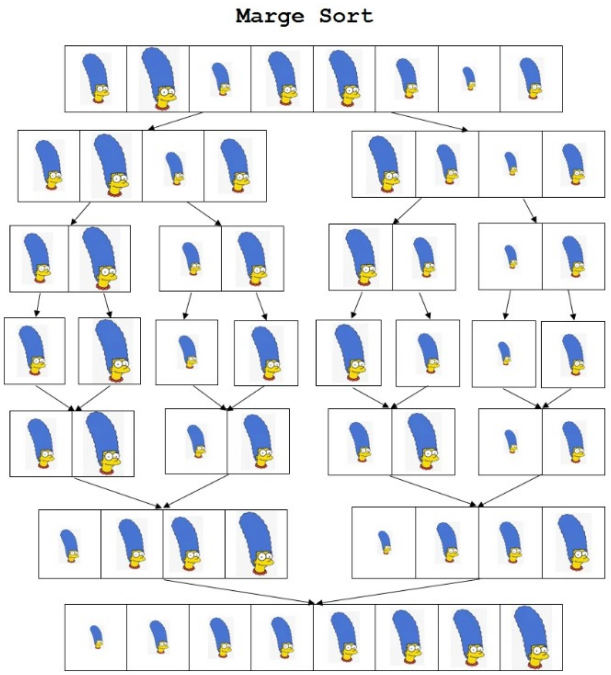

Figura 9: Analogia feita para o Merge Sort [Autor Desconhecido]

O **Merge Sort** é um algoritmo um pouco mais complexo de implementar, devido à sua abordagem estruturada de divisão e conquista. Ele é especialmente útil para listas grandes, pois mantém uma complexidade $O(n \log n)$ em todos os casos. No entanto, seu consumo de memória adicional pode torná-lo menos adequado em cenários com recursos limitados. Como já é conhecido o funcionamento do algoritmo e como a divisão, mesclagem e ordenação são realizadas, um exemplo de implementação em linguagem C é apresentado na figura [10].

```c
if (esquerda < direita) {
    int meio = esquerda + (direita - esquerda) / 2;

    mergeSort(vetor, esquerda, meio);
    mergeSort(vetor, meio + 1, direita);

    merge(vetor, esquerda, meio, direita);
}
```

Figura 10: Estrutura do código em C [Feito pelo Autor]

Em um teste de mesa, nota-se todo o porcesso desse algoritmo, desde sua divisão inicial em sublistas até chegar a somente 1 numero, até toda a ordenação do vetor. Para o teste de mesa em questão, usaremos a seguinte sequência: **8 - 3 - 5 - 4 - 6**

Teste de Mesa: Merge Sort			
Passo	**Divisão**	**Mesclagem**	**Resultado Parcial**
1	Lista inicial: *8, 3, 5, 4, 6*	-	*8, 3, 5, 4, 6*
2	Divide em: *8, 3, 5* e *4, 6*	-	*8, 3, 5 — 4, 6*
3	Sublista *8, 3, 5*: *8, 3* e *5*	-	*8, 3 — 5 — 4, 6*
4	Sublista *8, 3*: *8* e *3*	-	*8 — 3 — 5 — 4, 6*
5	-	Mescla *8* e *3*	*3, 8 — 5 — 4, 6*
6	-	Mescla *3, 8* e *5*	*3, 5, 8 — 4, 6*
7	-	Mescla *4* e *6*	*3, 5, 8 — 4, 6*
8	-	Mescla *3, 5, 8* e *4, 6*	*3, 4, 5, 6, 8*

3.6 Quick Sort

O algoritmo **Quick Sort** é um método de ordenação baseado no conceito de dividir e conquistar. Ele funciona escolhendo um **pivô** (elemento do vetor) e particionando os elementos em duas partes: aqueles menores que o pivô à esquerda e os maiores que o pivô à direita. Após essa separação, o mesmo processo é recursivamente aplicado às subpartições até que o vetor esteja ordenado.Uma característica importante do Quick Sort é que ele pode ser implementado de forma in-place, utilizando apenas uma quantidade constante de memória adicional. Na imagem de exemplo [11], observa-se uma simulação de seu funcionamento.

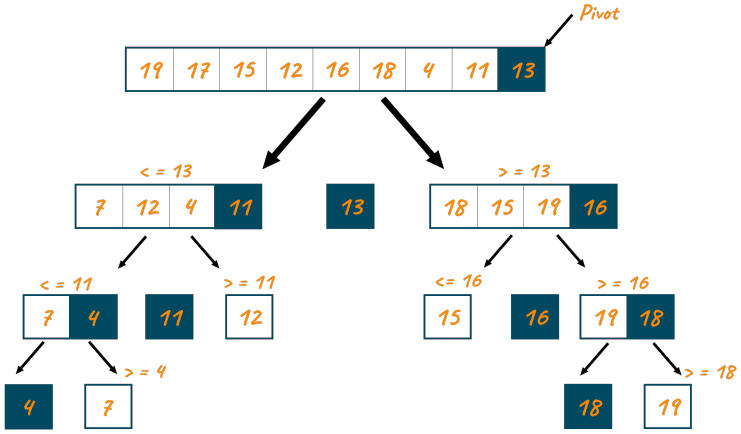

Figura 11: Exemplo de funcionamento Quick Sort [1]

O **Quick Sort** é conhecido pela sua eficiência, principalmente em grandes conjuntos de dados, mas apresenta um comportamento menos ideal em casos onde a escolha do pivô resulta em partições desbalanceadas. Como já se é conhecido o funcionamento do algoritmo e como a busca do pivô, divisão, mesclagem e ordenação são realizadas, um exemplo de implementação em linguagem C é apresentado na figura [12].

```
clock_t inicio = clock();

quick_sort_recursive(vetor, 0, tamanho - 1);

clock_t fim = clock();
double tempo_gasto = (double)(fim - inicio) / CLOCKS_PER_SEC;
printf("    Tempo gasto na ordenacao: %f segundos\n", tempo_gasto);
```

Figura 12: Estrutura do código em C [Feito pelo Autor]

Em um teste de mesa, nota-se todo o processo desse algoritmo, desde sua divisão inicial em sublistas até a ordenação completa do vetor. Para o teste de mesa em questão, usaremos a seguinte sequência: **8 - 3 - 5 - 4 - 6**.

Teste de Mesa: Quick Sort			
Passo	**Pivô**	**Particionamento**	**Resultado Parcial**
1	6	*[3, 5, 4] — 6 — [8]*	*3, 5, 4 — 6 — 8*
2	4	*[3] — 4 — [5] — 6 — 8*	*3 — 4 — 5 — 6 — 8*
3	-	Nenhuma mudança	*3, 4, 5, 6, 8*
4	-	Fim do algoritmo	*3, 4, 5, 6, 8*

3.7 Heap Sort

O algoritmo **Heap Sort** é um método de ordenação que utiliza a estrutura de dados conhecida como heap para organizar os elementos de um vetor. A ideia central é construir inicialmente um heap máximo (ou mínimo, dependendo do objetivo), onde o maior elemento está na raiz. Em seguida, o maior elemento é trocado com o último elemento do vetor, reduzindo o tamanho do heap e reorganizando-o para manter a propriedade de heap. Esse processo é repetido até que todos os elementos estejam ordenados, como ilustrado no exemplo da Figura [13].

3.7.1 Max_Heapify

Max_Heapify (*Heapify* ou **Verifica(A,x)**, sendo x um valor dentro dos parâmetros de quantitade referente ao *array*) é uma função que pode pertencer ao algoritmo de ordenação ***Heap Sort***. Ela possui o potencial de reduzir o tempo de complexidade da inserção em $O(n)$. Seu funcionamento ocorre da seguinte forma:

- Transformando o *array* em uma árvore binária completa, a mesma vai ser preenchida da **esquerda para a direita**, sendo que nenhum nódulo poderá ser parcialmente preenchido, a não ser pelas últimas folhas da árvore;

- Após o preenchimento, a mesma árvore é dividida em sub-árvores binárias (cada uma possui um nódulo-pai e um ou mais nódulos-filhos), onde as mesmas são subdivididas em diversas até que não sobre mais nenhuma não-detectada;

- Da sub-árvore binária mais baixa, compara-se os dois nódulos-filhos e o nódulo-pai e escolhe-se o maior dos três e colocam-o como nó pai (nódulo-pai). Após isso, é comparado o nó pai com o seu nó pai e continua-se a comparar e a trocar até à raíz. O objetivo é trocar os nós sempre que o **nó pai for maior do que o seu filho**;

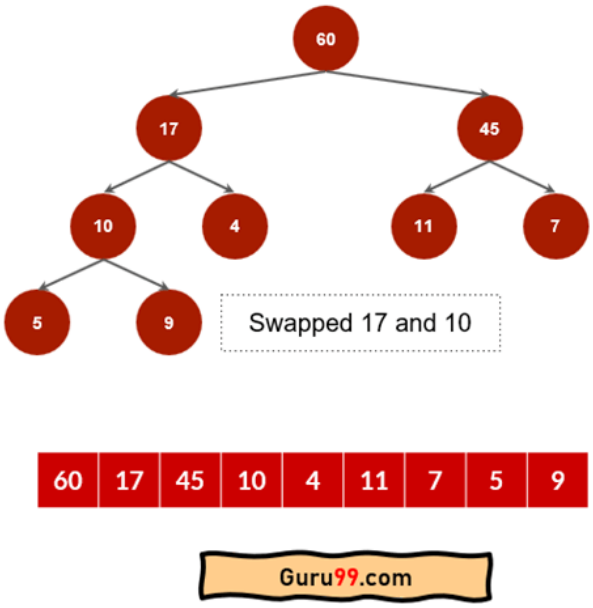

Figura 13: Exemplo de funcionamento [Sara Chen]

O **Heap Sort** é amplamente utilizado devido à sua robustez e previsibilidade, especialmente em situações onde é necessário garantir a eficiência no pior caso. O algoritmo utiliza a estrutura de dados heap para organizar os elementos de forma hierárquica e, em seguida, extrair os maiores (ou menores, dependendo da implementação) de maneira ordenada. Abaixo, é apresentado um exemplo de implementação em linguagem C, demonstrando a construção do heap e o processo de ordenação, conforme ilustrado na Figura [14].

```c
double heap_sort(int vetor[], int tamanho) {
    clock_t inicio = clock();

    for (int i = tamanho / 2 - 1; i >= 0; i--) {
        heapify(vetor, tamanho, i);
    }

    for (int i = tamanho - 1; i >= 0; i--) {
        int temp = vetor[0];
        vetor[0] = vetor[i];
        vetor[i] = temp;

        // Chama heapify no heap reduzido
        heapify(vetor, i, 0);
    }

    clock_t fim = clock();
    double tempo_gasto = (double)(fim - inicio) / CLOCKS_PER_SEC;
    printf("    Tempo gasto na ordenacao: %f segundos\n", tempo_gasto);

    return tempo_gasto;
}
```

Figura 14: Estrutura do código em C [Feito pelo Autor]

Em um teste de mesa, podemos observar todo o processo do algoritmo **Heap Sort**, desde a construção do **max-heap** até a ordenação completa do vetor. Para o teste de mesa, usaremos a sequência inicial: **8 - 3 - 5 - 4 - 6**.

Teste de Mesa: Heap Sort		
Passo	**Descrição**	**Estado do Heap**
1	Construção inicial do ***max-heap***	*8, 6, 5, 4, 3*
2	Troca da raiz com o último elemento	*3, 6, 5, 4 — 8*
3	Reestruturação do ***max-heap***	*6, 4, 5, 3 — 8*
4	Troca da raiz com o penúltimo elemento	*3, 4, 5 — 6, 8*
5	Reestruturação do ***max-heap***	*5, 4, 3 — 6, 8*
6	Troca da raiz com o antepenúltimo elemento	*3, 4 — 5, 6, 8*
7	Reestruturação do ***max-heap***	*4, 3 — 5, 6, 8*
8	Troca da raiz com o penúltimo elemento	*3 — 4, 5, 6, 8*
9	Fim do algoritmo	*3, 4, 5, 6, 8*

4 ANÁLISE DE COMPLEXIDADE

4.1 Insertion Sort

A analise de complexidade

Insertion Sort(A,n)	**Custo**	**Vezes**
1 for j ← 2 to comprimento	C_1	n
2 — do chave ← A[j] C2 n-1	C_2	n-1
3 — //inserir A[j] na sequência ordenada A(1...N)	$C_3 = 0$	n-1
4 — i ← j-1	C_4	n-1
5 — while i > 0 e A[i] > chave	C_5	$\sum_2^n tj$
6 —— do A[i+1] ← A[i]	C_6	$\sum_2^n (tj - 1)$
7 —— i ← i-1	C_7	$\sum_2^n (tj - 1)$
8 — A[i+1] ← chave	C_8	n-1

A tabela acima retirada do livro base [6], viabiliza calcular a complexidade do algoritmo **Insertion-Sort** a partir de certos cálculos, exemplificados a baixo:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) - c_8(n-1) \tag{1}$$

Com essa fómula, se torna possível o calculo de qual o melhor, médio e pior caso, de aplicação do algoritmo.

4.1.1 Melhor caso:

No ambiente no qual todos o numeros **já estão ordenamos de forma crescente**, a linha 5 de nosso pseudocódigo se torna não utilizável, assim, calculamos seu valor, mas retiramos o de cada parte de seu loop (linha 7 e 8):

$$t(n) = [C_1 n] + [C_2(n-1)] + [C_4(n-1)] + [C_5(n-1)] + [C_8(n-1)] \tag{2}$$

Resolve-se colocando n em evidência:

$$t(n) = n[C_1 + C_2 + C_4 + C_5] + [-C_2 - C_4 - C_5 - C_8] \tag{3}$$

Assim, a fórmula pode adotar a seguinte acertiva:

$$t(n) = A^{'}n + B \tag{4}$$

4.1.2 Médio caso:

Voltando a Fórmula base, adotamos:
$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) - c_8(n-1)$
Contudo, o ambiente agora está com **caracteristicas de aleatoriedade**, não estando nem em seu melhor caso(crescente), ne. em seu pior (descrescente). Assim tomamos:

$$t(n) = C_1n + C_2n + C_4n + C_5(\frac{n^2}{2} - 1) + C_6(\frac{n(n-1)}{2}) + C_7(\frac{n(n-1)}{2}) + C_8n + C_8 \tag{5}$$

Adotando $\sum_2^n j\frac{1}{2} = \frac{1}{2}\sum_2^n j$, temos:

- $C_5 = \frac{1}{2}(\frac{n^2+n}{2} - 1) \rightarrow \frac{n^2+n-2}{4}$
- C_6 e $C_7 = \frac{1}{2}(\frac{n(n-1)}{2}) - (n-1) \rightarrow \frac{n^2-n}{4} - (n+1) \rightarrow \frac{n^2-5n+4}{4}$

Substituindo os valores encontrados,em seus respectivos lugares:

$$t(n) = C_1n + C_2n - C_2 + C_4n - C_4 + \frac{C_5n^2+C_5-2C_5}{4} + \frac{C_6n^2-5C_6n-4C_6}{4} + \frac{C_7n^2-5C_7n+4C_6}{4} + C8n - C8$$

$$t(n) = n^2(\frac{C_5+C_6+C_7}{4} + n(C_1 + C_2 + C_4 + \frac{C_5-5C_5-5C_7}{4} + C_8) - (C_2 + C_4 + \frac{2C_5-4C_6-4C_7}{4} + C_8)$$

Assim, a fórmula pode adotar a seguinte acertiva:

$$t(n) = A^{''}n^2 + B^{''}n + C, \tag{6}$$

4.1.3 Pior caso:

Mais uma vez, voltando a Fórmula base, adota-se:
$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) - c_8(n-1)$

Agora, estamos na situação em que o vetor foi gerado, e tem-se a necessidade de coloca-lo em **órdem decrescente**, nesse caso será ultizado todos os valores de **Vezes** descritos na tabela, gerando uma Progressão Aritmética a ser calculada:

$$SPA = \frac{(a_1 + a_n)n}{2} \rightarrow \frac{(n-1)(2+n)}{2} \tag{7}$$

Resolvendo, abre-se a possibilidade de duas abordagens:

$$\frac{n + n^2}{2} - 1, \tag{8}$$

ou

$$\frac{n(n-1)}{2} - 1 \tag{9}$$

Usando a primeira abordagem (7), conseguimos:

$$t(n) = C_1n + C_2n - C_2 + C_4n - C_4 + \tag{10}$$
$$C_5\frac{n^2+n}{2} - 1 + C_6\frac{n^2+n}{2} - 1 + C_7\frac{n^2+n}{2} - 1 + C_8n - C_8 \tag{11}$$

Realizado as distributivas:

$$t(n) = C_1n + C_2n - C_2 + C_4n - C_4 + \tag{12}$$
$$\frac{C_5n}{2} + \frac{C_5n^2}{2} - C_5 + \frac{C_6n^2}{2} - \frac{C_6n}{2} + \frac{C_7n^2}{2} - \frac{C_7n}{2} + C_8n - C_8 \tag{13}$$

A partir da equação acima, é possível ver que colocar os valores em evidência é plausível,

da seguinte forma:

$$t(n) = n^2(\frac{C_5 + C_6 + C_7}{2}) + n(C_1 + C_2 + C_4 + \frac{C_5 - C_6 - C_7}{2} + C_8) + (-C_2 - C_4 - C_5 - C_8) \tag{14}$$

Assim, a fórmula pode adotar a seguinte acertiva:

$$t(n) = A^{''}n^2 + B^{''}n + C \tag{15}$$

4.2 Selection Sort

A analise de complexidade

Selection Sort(A,n)	**Custo**	**Vezes**
1 for (int i=0 ; i<tam-1 ; i++)	C_1	n
2 — int minIndex = i;	C_2	$n-1$
3 — for (**int** j=i+1 ; i<tam ; j++)	C_3	$\sum tj$
4 —— if (A[j] ¡ a[minIndex])	C_4	$\sum(tj-1)$
5 ——— minIndex = j;	C_5	$\sum(tj-1)$
6 —— if (minIndex != i)	C_6	$\sum(tj-1)$
7 ——— swap(A[minIndex], A[i])	C_7	$\sum(tj-1)$

A tabela acima retirada do livro base [6], viabiliza calcular a complexidade do algoritmo **Selection-Sort** a partir de certos cálculos, exemplificados a baixo:

$$T(n) = c_1 n + c_2 \sum_{i=1}^{n-1} i \tag{16}$$

Com essa fómula, se torna possível o calculo de qual o melhor, médio e pior caso de aplicação do algoritmo. Essa que pode ser calculada considerando que ele sempre faz uma quantidade fixa de operações para encontrar o menor elemento e trocá-lo de posição, percorrendo toda a lista.

No melhor, médio e pior caso, ele executa $\mathcal{O}(n^2)$ comparações, devido ao número de iterações necessárias para percorrer toda a lista. Assim, em todos os casos o algoritmo **Selction Sort**, adota uma mesma resolução de:

$$t(n) = An^2 + Bn - C$$

4.2.1 Melhor caso:

Lista já ordenada em ordem crescente.

$$T(n) = c_1 n + c_2 \frac{(n-1)n}{2} \Rightarrow \mathcal{O}(n^2) \tag{17}$$

Assim, a fórmula pode adotar a seguinte acertiva:

$$t(n) = An^2 + Bn - C$$

4.2.2 Médio caso:

Lista em ordem aleatória: $\sum \frac{(tj)}{2}$ e $\sum \frac{(tj-1)}{2}$, ao invés de valores inteiros: $\sum tj$ e $\sum \frac{(}{t}j-1)$:

$$t(n) = C1n + C2(n-1) + C3(\sum(tj)) + C4(\sum(tj-1)) +$$
$$+C5(\sum(tj-1)) + C6(\sum(tj-1)) + C7(\sum(tj-1))$$

Assim, a fórmula pode adotar a seguinte acertiva:

$$t(n) = An^2 + Bn - C$$

4.2.3 Pior caso:

Lista em ordem decrescente.

$$T(n) = c_1 n + c_2 \frac{(n-1)n}{2} \Rightarrow \mathcal{O}(n^2) \tag{18}$$

Assim, a fórmula podendo adotar também, a seguinte acertiva:

$$t(n) = An^2 + Bn - C$$

4.3 Bubble Sort

A analise de complexidade

Bubble Sort(A,n)	**Custo**	**Vezes**
1 for i=1 até comprimento [A]-1	C_1	n
2 — for j=1 até comprimento [A]-i	C_2	$\sum_1^{n-1} tj$
3 —— Se (A[j] > A[j+1]) então	C_3	$\sum_1^{n-1}(tj-1)$
4 ——— troca A[j] $\leftarrow\rightarrow$ A[j+1]	C_4	$\sum_1^{n-1}(tj-1)$

A tabela acima retirada do livro base [6], viabiliza calcular a complexidade do algoritmo **Bubble-Sort** a partir de certos cálculos, exemplificados a baixo:

$$t(n) = C1n + C2[\sum_{1}^{n-1} tj] + C3[\sum_{1}^{n-1}(tj-1)] + C4[\sum_{1}^{n-1}(tj-1)] \tag{19}$$

Com essa fómula, se torna possível o calculo de qual o melhor, médio e pior caso, de aplicação do algoritmo.

$$SPA = \frac{(a_1 + a_n)n}{2} \rightarrow \frac{(1+n-1)(n-1)}{2} \rightarrow \frac{n^2-n}{2} \tag{20}$$

Resultando em:

$$\sum_{1}^{n-1}(tj-1) = \frac{n^2-n}{2} - n - 1 \rightarrow \frac{n^2-n-2n+n}{2} \rightarrow \frac{n^2-3n}{2} + 1 \tag{21}$$

4.3.1 Melhor caso:

No ambiente no qual todos o numeros **já estão ordenamos de forma crescente**, a *linha 4* do pseudo-código **não irá ocorrer**, assim, a partir da equação base:

$$t(n) = C1n + C2\sum_{1}^{n-1} tj + C3(\sum_{1}^{n-1}(tj-1)) \tag{22}$$

Expandindo a equação geral:

$$t(n) = C1n + C2(\tfrac{n^2-n}{2} + C3(\tfrac{n^2-3n+2}{2})$$

$$t(n) = C1n + (\frac{C2n^2 - C2}{2}) + (\frac{C3n^2 - 3nC3 + C3}{2})$$
$$t(n) = n^2[\frac{C2 + C3}{2}] + n[\frac{-C2 - 3C3}{2}] + C3$$

Assim, a resolução pode adotar a seguinte acertiva:

$$t(n) = An^2 + Bn + C$$

4.3.2 Médio caso:

O cenário no qual o vetor foi gerado em ordem decrescente, teremos $j = \frac{1}{2}$ para a realizção dos cálculos de todos os somatórios:

$$\sum_{1}^{n-1}(\frac{tj}{2}) \tag{23}$$

A partir disso, é necessário apenas adaptar o novo valor, acrescentando o $\frac{1}{2}$ nos cálculos:

$$\sum_{1}^{n-1}(\frac{tj}{2}) = \frac{n^2 - n}{2} \times \frac{1}{2} = \frac{n^2 - n}{4} \tag{24}$$

Com o primeiro somatório modificado,partimos para o próximo e em seguida, abstraimos os valores já existentes: $\sum_{1}^{n-1}(\frac{tj}{2} - 1)$

$$\sum_{1}^{n-1}(\frac{tj - 1}{2}) = \sum_{1}^{n-1}\frac{tj}{2} - \sum_{1}^{n-1}\frac{1}{2} \tag{25}$$

$$\sum_{1}^{n-1}(\frac{tj - 1}{2}) = \frac{n^2 - n}{2} \times \frac{1}{2} - \frac{n - 1}{2} \tag{26}$$

Resolvendo as operações:

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n}{4} - \frac{n-1}{2}$$
$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n-(2n-2)}{4}$$
$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n-2n+2}{4}$$

O resultado final sendo:

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-3n+2}{4} \tag{27}$$

Por fim, com todas as substituição de valores e as resoluções, obtemos:

$$t(n) = C_1 n + \frac{C_2 n^2 - C_2 n}{4} + \frac{C_3 n^2 - 3C_3 n}{4} + C_3 + \frac{C_4 n^2 - 3C_4 n}{4} + C_4$$
$$t(n) = n^2(\frac{C_2+C_3+C_4}{4}) + n(C_1 - \frac{-C_2-3C_3-3C_4}{4}) + C_3 + C_4$$

Assim, a resolução pode adotar a seguinte acertiva:

$$t(n) = An^2 + Bn + C$$

4.3.3 Pior caso:

O cenário agora está com **caracteristicas de aleatoriedade**, não estando nem em seu melhor casoc(crescente), nem no médio (descrescente). Assim tomamos:

$$t(n) = C_1 n + C_2(\frac{n^2-n}{2} + C_3(\frac{n^2-3n+2}{2}) + C_4(\frac{n^2-3n+2}{2})$$
$$t(n) = C_1 n + (\frac{C_2 n^2 - C_2}{2}) + (\frac{C_3 n^2 - 3nC_3 + C_3}{2}) + (\frac{C_4 n^2 - 3nC_4 + C_4}{2})$$
$$t(n) = n^2[\frac{C_2+C_3+C_4}{2}] + n[\frac{-C_2-3C_3-3C_4}{2}] + C_3 + C_4$$

Assim, a resolução pode adotar a seguinte acertiva:

$$t(n) = An^2 + Bn + C$$

4.4 Merge Sort

A analise de complexidade

Merge e seus Casos				
Código	Custo	Melhor Caso	Pior Caso	Médio Caso
1 for (int i=0 ; i¡n1 ; i++)	C_1	n	n	n
2 — L[i] = arr[l+i]	C_2	$n-1$	$n-1$	$n-1$
3 for (int j=0 ; j¡n2 ; j++)	C_3	$n-1$	$n-1$	$n-1$
4 — R[j] = arr[m + 1 + j]	C_4	$n-1$	$n-1$	$n-1$
5 while (i¡n1 e j¡n2)	C_5	n	n	n
6 — if (L[i] ¡= R[j])	C_6	$n-1$	$n-1$	$n-1$
7 —— arr[k] = L[i]	C_7	$n-1$	$n-1$	$n-1$
8 —— i++	C_8	$n-1$	$n-1$	$n-1$
9 — else	C_9	$n-1$	$n-1$	$n-1$
10—— arr[k] = R[j]	C_10	$n-1$	$n-1$	$n-1$
11—— j++;	C_11	$n-1$	$n-1$	$n-1$
12—— k++	C_12	$n-1$	$n-1$	$n-1$
13 while (i ¡ n1)	C_13	n	n	n
14— arr[k] = L[i]	C_14	$n-1$	$n-1$	$n-1$
15— i++	C_15	$n-1$	$n-1$	$n-1$
16— k++	C_16	$n-1$	$n-1$	$n-1$
17 while (j ¡ n2)	C_7	n	n	n
18— arr[k] = R[j]	C_18	$n-1$	$n-1$	$n-1$
19— j++	C_19	$n-1$	$n-1$	$n-1$
20— k++	C_20	$n-1$	$n-1$	$n-1$

O merge sort por ser um algortimo com complexidade linear, apresenta tando no seu melhor caso, quanto no pior e médio, a mesma complexiade. Sendo um dos métodos mais rápidos de ordenação estudados até o momento, mostrando uma boa enficiência em todos os tamanos e casos do vetor. Sendo assim, a tabela acima, viabiliza calcular a complexidade do algoritmo **Merge Sort** a partir de certos cálculos, exemplificados abaixo:

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn \tag{28}$$

Expandindo a relação de recorrência pelo método da substituição:

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + C\frac{n}{2}\right] + Cn \tag{29}$$

$$= 4T\left(\frac{n}{4}\right) + 2C\frac{n}{2} + Cn \tag{30}$$

$$= 4\left[T\left(\frac{n}{8}\right) + C\frac{n}{4}\right] + 2C\frac{n}{2} + Cn \tag{31}$$

$$= 8T\left(\frac{n}{8}\right) + 4C\frac{n}{4} + 2C\frac{n}{2} + Cn \tag{32}$$

Generalizando, após k expansões:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + Cn\sum_{i=0}^{k-1}\frac{1}{2^i} \tag{33}$$

Quando $k = \log_2(n)$, temos:

$$T(1) = \Theta(1) \quad \text{(caso base)} \tag{34}$$

$$T(n) = n\log_2(n) + \Theta(n) \tag{35}$$

4.4.1 Melhor caso:

O melhor caso ocorre quando os elementos já estão particionados de forma ideal, minimizando as comparações, mas como a complexidade do Merge Sort depende da divisão e fusão, o tempo permanece:

$$T(n) = n\log_2(n) \tag{36}$$

4.4.2 Médio caso:

No médio caso, o vetor apresenta elementos em ordem aleatória, resultando em:

$$T(n) = n \log_2(n) \tag{37}$$

4.4.3 Pior caso:

No pior caso, mesmo com elementos em uma ordem específica (ex.: inversa), a divisão e a fusão mantêm a mesma complexidade:

$$T(n) = n \log_2(n) \tag{38}$$

Assim, para todos os cenários, a complexidade final do Merge Sort é:

$$T(n) = n \log_2(n)$$

O merge sort por ser um algortimo com complexidade linear, apresenta tando no seu melhor caso, quanto no pior e médio, a mesma complexiade. Sendo um dos métodos mais rápidos de ordenação estudados até o momento, mostrando uma boa enficiência em todos os tamanos e casos do vetor, até mesmo para o mais complexo e demorado (Decrescente de 1.000.000).

4.5 Quick Sort

A analise de complexidade

Quick Sort(A, low, high)	**Custo**	**Vezes**
1 if (low ¡ high)	C_1	n
2 — int pivotIndex = partition(A, low, high)	C_2	$n-1$
3 — QuickSort(A, low, pivotIndex-1)	C_3	$\sum tj$
4 — QuickSort(A, pivotIndex+1, high)	C_4	$\sum(tj-1)$

A tabela acima viabiliza calcular a complexidade do algoritmo **Quick Sort** a partir de certos cálculos, exemplificados a seguir:

$$t(n) = C_1 n + C_2[\sum_{1}^{n-1} tj] + C_3[\sum_{1}^{n-1}(tj-1)] + C_4[\sum_{1}^{n-1}(tj-1)] \tag{39}$$

Com essa fórmula, torna-se possível calcular o melhor, médio e pior caso de aplicação do algoritmo.

4.5.1 Melhor Caso:

No ambiente em que o **pivô sempre divide o vetor igualmente**, o número de comparações será mínimo. Partindo da equação base:

$$t(n) = C_1 n + C_2 \sum_{1}^{n-1} tj + C_3(\sum_{1}^{n-1}(tj-1)) + C_4(\sum_{1}^{n-1}(tj-1)) \tag{40}$$

Expandindo a equação geral:

$$t(n) = C_1 n + (\frac{C_2 n \log n}{2}) + (\frac{C_3 n \log n}{2}) + (\frac{C_4 n \log n}{2})$$
$$t(n) = n[\frac{C_2+C_3+C_4}{2} \log n] + C_1 n$$

Assim, a resolução pode adotar a seguinte forma:

$$t(n) = An \log n + Bn$$

4.5.2 Médio Caso:

Em uma execução média, o pivô **divide o vetor de forma balanceada**, mas não exatamente no meio. Assim, os cálculos consideram um fator de aleatoriedade no tamanho das partições:

$$t(n) = C_1 n + C_2[\frac{n \log n}{2}] + C_3[\frac{n \log n}{2}] + C_4[\frac{n \log n}{2}] \tag{41}$$

Expandindo os valores:

$$t(n) = n[\tfrac{C_2+C_3+C_4}{2} \log n] + C_1 n$$

Resultando na mesma forma do melhor caso:

$$t(n) = An \log n + Bn$$

4.5.3 Pior Caso:

O cenário mais desfavorável ocorre quando o pivô **é sempre o menor ou o maior elemento**, resultando em partições desbalanceadas. Nesse caso:

$$t(n) = C_1 n + C_2[\sum_{1}^{n-1} tj] + C_3[\sum_{1}^{n-1}(tj-1)] + C_4[\sum_{1}^{n-1}(tj-1)] \tag{42}$$

Expandindo para o caso linear:

$$t(n) = C_1 n + (\tfrac{C_2 n^2}{2}) + (\tfrac{C_3 n^2}{2}) + (\tfrac{C_4 n^2}{2})$$
$$t(n) = n^2[\tfrac{C_2+C_3+C_4}{2}] + C_1 n$$

A resolução final adota a seguinte forma:

$$t(n) = An^2 + Bn$$

4.6 Heap Sort

A análise de complexidade do algoritmo **Heap Sort** pode ser realizada a partir da tabela a seguir, que detalha os custos associados às principais operações do algoritmo:

Heap Sort(A)	**Custo**	**Vezes**
1 — BuildMaxHeap(A)	C_1	n
2 — for i = n downto 2 do	C_2	$n-1$
3 —— Swap(A[1], A[i])	C_3	$n-1$
4 —— MaxHeapify(A, 1, i-1)	C_4	$\sum_1^{n-1} \log j$

A tabela acima permite calcular a complexidade do **Heap Sort**, sendo que o custo total é dado pela seguinte fórmula:

$$t(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{1}^{n-1} \log j \tag{43}$$

4.6.1 Melhor Caso:

No melhor caso, a reorganização do heap é mínima, o que ocorre quando o vetor já está parcialmente ordenado (crescente). Expandindo a fórmula:

$$t(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4[\log n] \tag{44}$$

Isso resulta em uma complexidade de tempo:

$$t(n) = An \log n + Bn$$

4.6.2 Médio Caso:

No caso médio, o vetor não possui nenhuma ordem específica (random), e as operações de reorganização ocorrem em um heap balanceado. Assim, o somatório $\sum_1^{n-1} \log j$ permanece proporcional a $n \log n$. A fórmula geral se mantém:

$$t(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n \log n) \tag{45}$$

O que resulta em:

$$t(n) = An \log n + Bn$$

4.6.3 Pior Caso:

O pior caso ocorre quando o vetor está ordenado de forma decrescente, exigindo o máximo de operações de reorganização. Nesse caso, a complexidade também permanece $O(n \log n)$, pois as operações de construção e manutenção do heap são dominadas pelo custo logarítmico:

$$t(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n \log n) \tag{46}$$

Assim, mesmo no pior caso, temos:

$$t(n) = An \log n + Bn$$

O **Heap Sort**, portanto, destaca-se por sua estabilidade na complexidade de tempo $O(n \log n)$ em todos os cenários, sendo confiável para aplicações práticas que requerem consistência de desempenho.

5 TABELA E GRÁFICO

5.1 Insertion Sort

A seguir, serão apresentados o gráfico [15] e a tabela [1] que ilustram o desempenho do **Insertion Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

// TEMPO //	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.000s	0.000s	0.001s	0.003s
Decrescente	0.000s	0.000s	0.001s	0.088s	8.846s	928.356s
Random	0.000s	0.000s	0.000s	0.044s	4.404s	467.756s

Tabela 1: Tabela de tempo por segundo do algoritmo Insertion Sort [Feito pelo Autor]

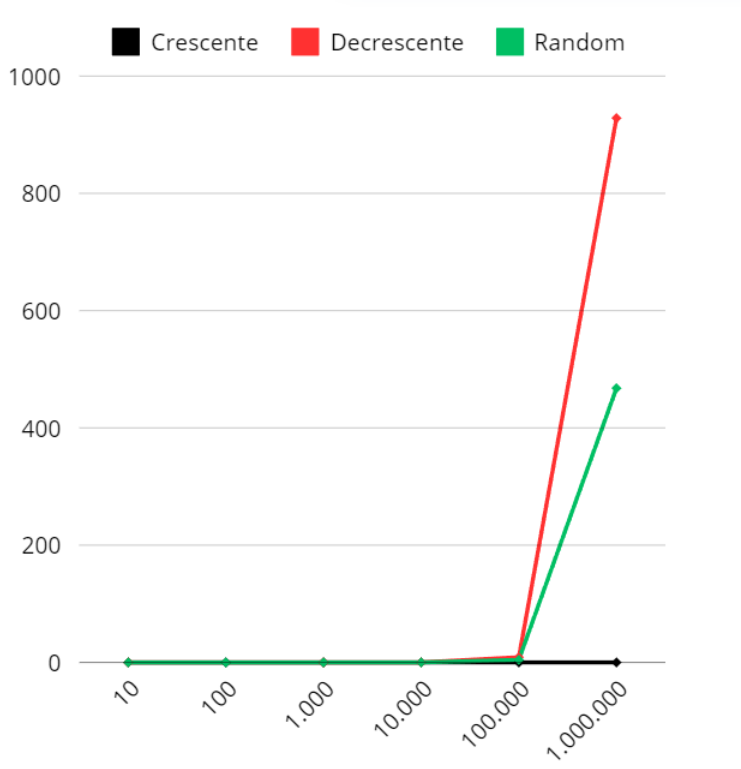

Figura 15: Gráfico de tempo por segundo do algoritmo Insertion Sort [Feito pelo Autor]

Como se é notado por meio da analise da tabela e do gráfico, o algoritmo **Insertion Sort**, é um algoritmo eficiente para ordenar um número pequeno de elementos [2]. A medida em que seu vetor se expande e a quantida de numeros cresce, ele enfrenta dificuldades para a ordenção, demandando mais tempo. Isso se dá por conta de independente de o vetor ja estar ordenado ou não, ele realizará sempre o mesmo número de comparações.

5.2 Selection Sort

A seguir, serão apresentados o gráfico [16] e a tabela [2] que ilustram o desempenho do **Selection Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.001s	0.102s	10.641s	963.770s
Decrescente	0.000s	0.000s	0.000s	0.107s	10.738s	1069.746s
Random	0.000s	0.000s	0.001s	0.097s	9.807s	1016.396s

Tabela 2: Tabela de tempo por segundo do algoritmo Selection Sort [Feito pelo Autor]

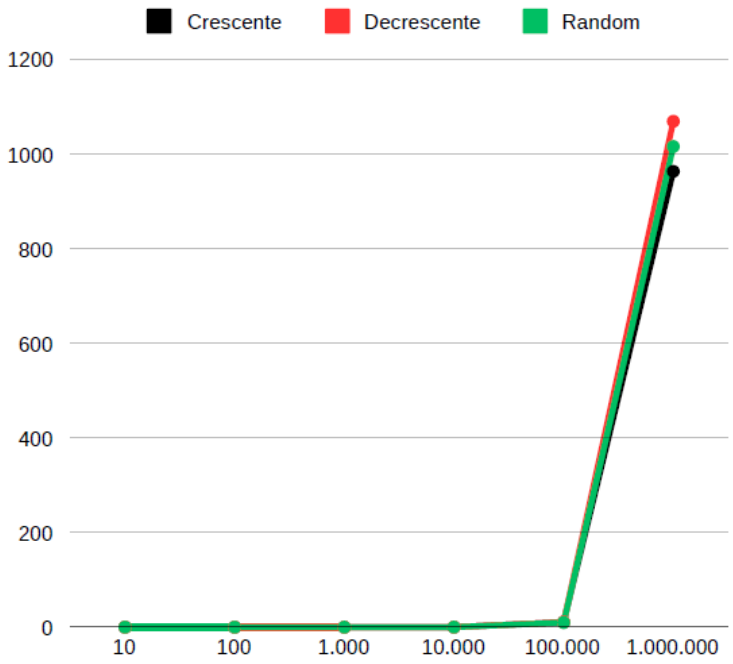

Figura 16: Gráfico de tempo por segundo do algoritmo Selection Sort [Feito pelo Autor]

Como se é notado por meio da analise da tabela e do gráfico, o algoritmo **Selection Sort** é eficiente para listas pequenas e para situações onde o consumo de memória é uma limitação, pois ele ordena o vetor no próprio array (*in-place*). Contudo, ele se torna lento e ineficiente para listas grandes [2].

5.3 Shell Sort

A seguir, serão apresentados o gráfico [17] e a tabela [3] que ilustram o desempenho do **Shell Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.000s	0.001s	0.004s	0.053s
Decrescente	0.000s	0.000s	0.000s	0.000s	0.006s	0.070s
Random	0.000s	0.000s	0.000s	0.001s	0.021s	0.251s

Tabela 3: Tabela de tempo por segundo do algoritmo Shell Sort [Feito pelo Autor]

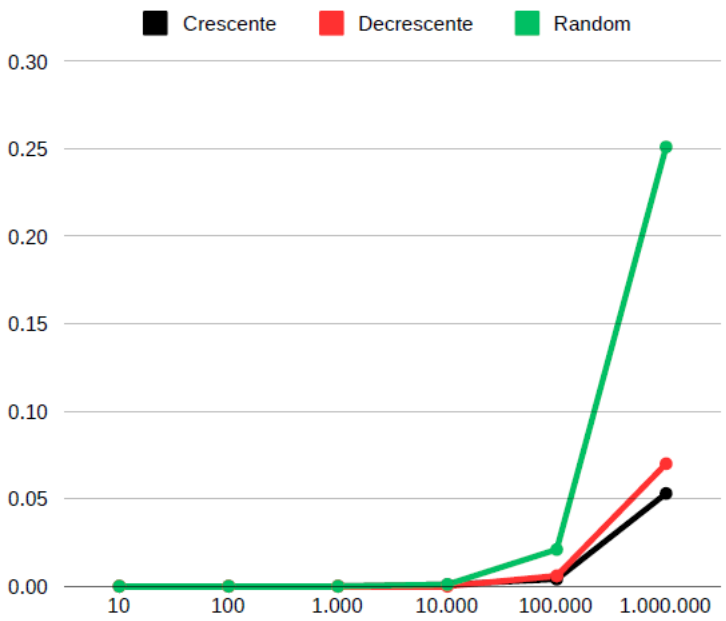

Figura 17: Gráfico de tempo por segundo do algoritmo Shell Sort [Feito pelo Autor]

Como se é notado por meio da analise da tabela e do gráfico, o algoritmo **Shell Sort** é uma versão aprimorada do Insertion Sort. Esse método permite uma redução significativa na quantidade de deslocamentos necessários, resultando em uma maior eficiência para listas de tamanho moderado, no entanto, para listas muito grandes, o Shell Sort perde em desempenho.

5.4 Bubble Sort

A seguir, serão apresentados o gráfico [18] e tabela [4] que ilustram o desempenho do **Bubble Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.001s	0.072s	7.052s	771.395s
Decrescente	0.000s	0.000s	0.002s	0.171s	17.417s	1749.219s
Random	0.000s	0.000s	0.002s	0.146s	21.830s	2196.778s

Tabela 4: Tabela de tempo por segundo do algoritmo Bubble Sort [Feito pelo Autor]

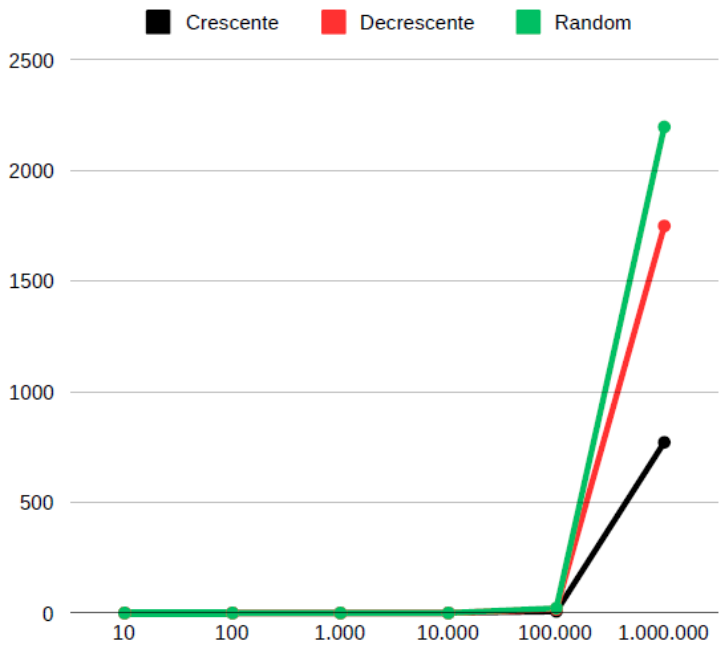

Figura 18: Gráfico de tempo por segundo do algoritmo Bubble Sort [Feito pelo Autor]

Como se é notado por meio da analise da tabela e do gráfico, o algoritmo **Bubble Sort** é um dos algoritmos mais simples de ordenação, entretanto, é ineficiente para listas de tamanho grande, pois necessita de múltiplas passagens. Mesmo que a lista esteja parcialmente ordenada, o algoritmo ainda precisa realizar diversas comparações e trocas para garantir que

tudo esteja no lugar. [2] Assim, o Bubble Sort é mais utilizado para fins didáticos do que práticos.

5.5 Merge Sort

A seguir, serão apresentados o gráfico [19] e a tabela [5] que ilustram o desempenho do **Merge Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.001s	0.001s	0.021s	0.245s
Decrescente	0.000s	0.000s	0.001s	0.002s	0.019s	0.221s
Random	0.000s	0.000s	0.000s	0.002s	0.028s	0.276s

Tabela 5: Tabela de tempo por segundo do algoritmo Merge Sort [Feito pelo Autor]

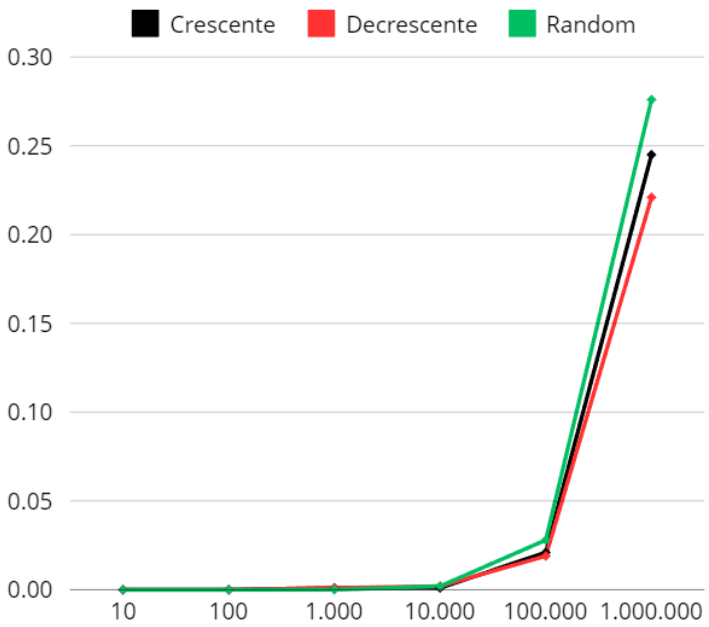

Figura 19: Gráfico de tempo por segundo do algoritmo Merge Sort [Feito pelo Autor]

Como se é notado por meio da analise da tabela e do gráfico, o algoritmo **Merge Sort** é um dos mais eficientes para ordenação, devido à sua abordagem de divisão e conquista. Ele realiza uma divisão sistemática da lista e, posteriormente, uma mesclagem ordenada, garantindo uma complexidade $O(n \log n)$ mesmo nos piores casos. [2] No entanto, o Merge

Sort exige espaço adicional proporcional ao tamanho da lista, o que pode ser uma limitação em cenários com restrições de memória. Apesar disso, sua eficiência e estabilidade tornam o algoritmo amplamente utilizado em contextos práticos e acadêmicos.

5.6 Quick Sort

A seguir, serão apresentados o gráfico [20] e a tabela [6] que ilustram o desempenho do **Quick Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.000s	0.000s	0.004s	0.048s
Decrescente	0.000s	0.000s	0.000s	0.001s	0.009s	0.110s
Random	0.000s	0.000s	0.000s	0.001s	0.010s	0.125s

Tabela 6: Tabela de tempo por segundo do algoritmo Quick Sort [Feito pelo Autor]

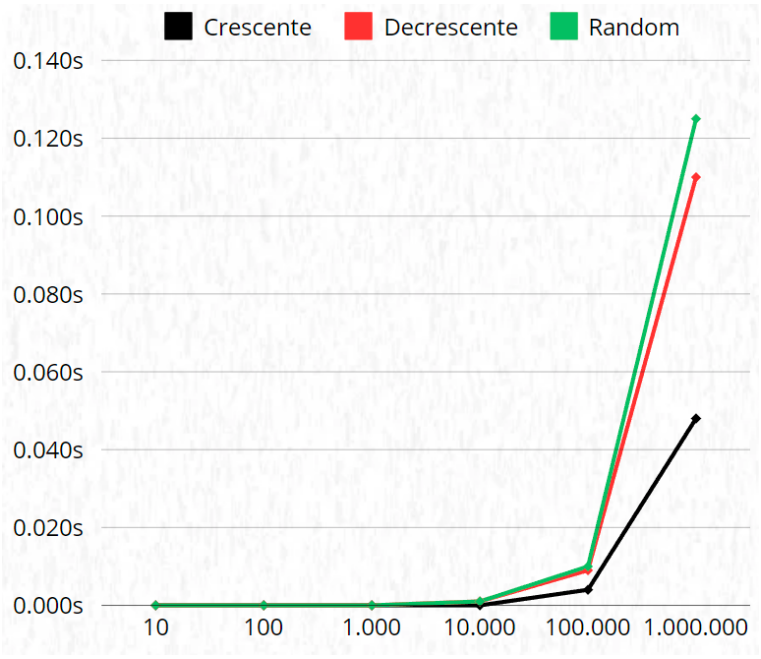

Figura 20: Gráfico de tempo por segundo do algoritmo Quick Sort [Feito pelo Autor]

Como se observa por meio da análise da tabela e do gráfico, o algoritmo **Quick Sort** destaca-se por sua eficiência em muitos casos práticos, sendo amplamente utilizado devido à sua complexidade média de O(nlogn). Esse desempenho é alcançado por meio de sua estratégia de divisão e conquista, que organiza os elementos em torno de um pivô, dividindo

o problema em subproblemas menores. No entanto, em casos específicos — como quando os dados estão quase ordenados e o pivô escolhido é desfavorável — o algoritmo pode atingir sua complexidade O(n2), tornando-se menos eficiente. [2]. Uma vantagem importante do Quick Sort é seu baixo consumo de memória, pois é implementado de forma in-place, o que o torna adequado para sistemas com recursos limitados.

5.7 Heap Sort

A seguir, serão apresentados o gráfico [21] e tabela [7] que ilustram o desempenho do **Heap Sort**. Os dados incluem o tempo de execução em funçao do tamanho do vetor gerado para diferentes tamanhos de vetores (crescente, decrescente e aleatória). A análise visual proporcionada por essas representações, ajudará a compreender a eficiência relativa de cada algoritmo.

//TEMPO//	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000s	0.000s	0.000s	0.001s	0.015s	0.169s
Decrescente	0.000s	0.000s	0.000s	0.002s	0.013s	0.151s
Random	0.000s	0.000s	0.000s	0.002s	0.017s	0.307s

Tabela 7: Tabela de tempo por segundo do algoritmo Heap Sort [Feito pelo Autor]

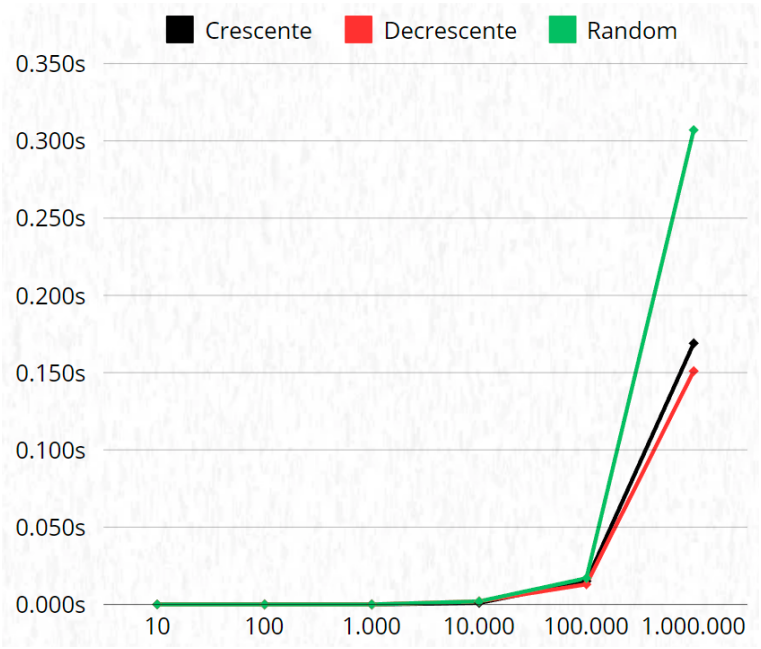

Figura 21: Gráfico de tempo por segundo do algoritmo Heap Sort [Feito pelo Autor]

Como se observa por meio da análise da tabela e do gráfico, o algoritmo **Heap Sort** destaca-se por sua eficiência em garantir uma complexidade de tempo O(nlogn) em todos os casos, devido à estrutura de dados subjacente: a heap binária. Esse desempenho uniforme é alcançado por meio da construção de uma heap máxima (ou mínima), seguida de sucessivas

remoções do maior (ou menor) elemento e reorganização da heap. Embora o Heap Sort seja estável em sua eficiência, ele apresenta um consumo de memória ligeiramente maior em comparação com o Quick Sort, devido à necessidade de manipular a estrutura de heap. Apesar disso, ele é amplamente utilizado em contextos onde a consistência de desempenho é crucial. [2]. Outra vantagem importante do Heap Sort é que ele não sofre degradação no desempenho com base na ordem inicial dos dados, tornando-o confiável em aplicações práticas.

5.8 Comparação dos Algoritmos

A seguir, será apresentado o gráfico, [22] que compara o desempenho de diferentes algoritmos de ordenação, incluindo: **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **Shell Sort**, **Bubble Sort**, **Merge Sort** e **Quick Sort**. Os dados incluem o tempo de execução em função do tamanho do vetor gerado. A análise visual proporcionada por essa representação, ajudará a compreender a eficiência relativa de cada algoritmo.

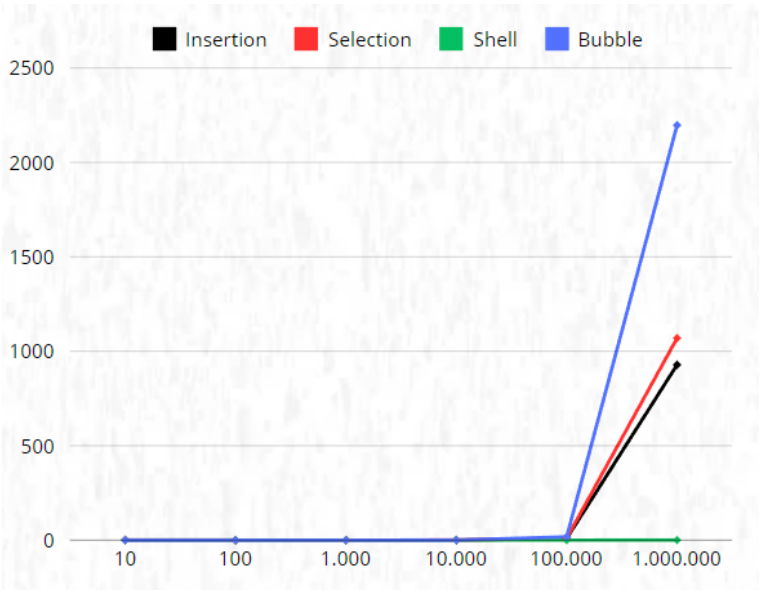

Figura 22: Grafico de comparação algoritmos [Feito Pelo Autor]

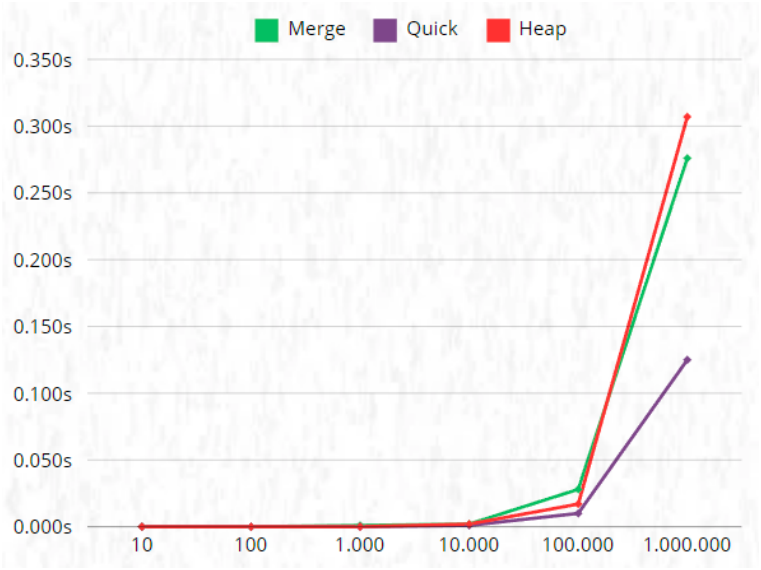

Figura 23: Grafico de comparação algoritmos método Divisão e Conquista [Feito Pelo Autor]

6 CONCLUSÃO

O **Insertion Sort** é um algoritmo de ordenação simples e eficiente para casos específicos. Ele funciona de maneira semelhante à organização manual de cartas, onde cada elemento é inserido na posição correta em relação aos elementos anteriores, ele não é ideal para grandes volumes de dados, pois tende a ser mais lento nesses casos. No entanto, o Insertion Sort se destaca por ser rápido e de fácil implementação em listas pequenas ou quase ordenadas. No caso crescente, o algoritmo atinge seu melhor caso; no caso decrescente, o pior caso; e na ordenação aleatória, ele atinge o caso médio.

O **Selection Sort** é um algoritmo de ordenação básico que funciona selecionando o menor elemento e colocando-o na posição correta a cada passagem pela lista. Ele é fácil de entender e implementar, mas não é muito eficiente para listas grandes, já que sempre precisa percorrer toda a lista para garantir a ordem. Esse algoritmo funciona bem em listas pequenas, mas fica lento em listas maiores. Seu melhor desempenho ocorre quando a lista já está em ordem crescente, e o pior caso ocorre quando está em ordem decrescente. Com uma lista aleatória, ele apresenta uma performance média.

O **Shell Sort** é uma versão mais avançada do Insertion Sort, onde os elementos são comparados em intervalos maiores que vão diminuindo até que toda a lista esteja ordenada. Ele é bem mais rápido que o Insertion Sort em listas grandes, pois reduz a quantidade de trocas necessárias para colocar os elementos na posição certa. No melhor caso, o Shell Sort é muito rápido, especialmente em listas que já estão parcialmente organizadas. No pior caso, sua eficiência pode se aproximar da do Selection Sort se os intervalos não forem bem escolhidos. Em listas aleatórias, ele tem um bom desempenho na média, especialmente para listas grandes.

O **Bubble Sort** é um algoritmo de ordenação simples, que compara elementos vizinhos e os troca se estiverem fora de ordem, repetindo o processo até a lista estar totalmente organizada. Apesar de ser fácil de implementar, ele é lento para listas grandes porque precisa de várias passagens para garantir que todos os elementos estejam ordenados. O melhor caso acontece se a lista já está ordenada, onde ele faz poucas trocas. No pior caso, ele precisa fazer muitas trocas e comparações, especialmente se a lista estiver em ordem decrescente.

Em uma lista aleatória, seu desempenho médio ainda é bem lento, sendo o mais demorado de todos os algoritmos estudados.

O **Merge Sort** é um algoritmo de ordenação eficiente, que segue a abordagem de divisão e conquista. Ele divide a lista em sublistas menores, ordena cada uma delas e, em seguida, mescla as sublistas de forma ordenada para compor a lista final. No entanto, uma de suas limitações é o consumo adicional de memória durante o processo de mesclagem, o que pode ser um problema em sistemas com restrições de recursos. Apesar disso, o Merge Sort se destaca por seu desempenho consistente e por ser ideal para lidar com grandes volumes de dados, sendo um dos mais rápidos dos algoritmos estudados.

O **Quick Sort** é um algoritmo eficiente que utiliza a estratégia de divisão e conquista para ordenar listas. Ele organiza os elementos menores que o pivô à esquerda e os maiores à direita, repetindo o processo recursivamente. Adaptações como o uso do pivô mediano melhoram sua eficiência. No entanto, no pior caso geralmente quando a lista está ordenada de forma crescente ou decrescente e um pivô desfavorável é escolhido. Por ser *in-place*, o Quick Sort é uma escolha prática para ordenar grandes volumes de dados.

O **Heap Sort** é um algoritmo de ordenação baseado na estrutura de dados heap binário, que garante a ordenação ao transformar a lista em um **max-heap** ou **min-heap** e reorganizá-la iterativamente. Ele apresenta complexidade de tempo consistente sendo eficiente e previsível no uso de memória. Embora não seja um algoritmo *in-place* puro, é uma escolha confiável para cenários onde a estabilidade de desempenho e o controle de memória são fundamentais.

Abaixo segue a Complexidade de cada um:

Insertion Sort	
Nome	**Insertion Sort**
Análise de Complexidade	
Melhor Caso	$O(n)$
Médio Caso	$O(n^2)$
Pior Caso	$O(n^2)$

Tabela 8: Insertion Sort

Bubble Sort	
Nome	**Bubble Sort**
Análise de Complexidade	
Melhor Caso	$O(n^2)$
Médio Caso	$O(n^2)$
Pior Caso	$O(n^2)$

Tabela 9: Bubble Sort

Shell Sort	
Nome	**Shell Sort**
Análise de Complexidade	
Melhor Caso	$O(n \log_2 n)$
Médio Caso	Sequência do *gap*
Pior Caso	$O(n \log_2 n)$

Tabela 10: Shell Sort

Selection Sort	
Nome	**Selection Sort**
Análise de Complexidade	
Melhor Caso	$O(n^2)$
Médio Caso	$O(n^2)$
Pior Caso	$O(n^2)$

Tabela 11: Selection Sort

Merge Sort	
Nome	**Merge Sort**
Análise de Complexidade	
Melhor Caso	$O(n \log_2 n)$
Médio Caso	$O(n \log_2 n)$
Pior Caso	$O(n \log_2 n)$

Tabela 12: Merge Sort

Quick Sort	
Nome	**Quick Sort**
Análise de Complexidade	
Melhor Caso	$O(n \log_2 n)$
Médio Caso	$O(n \log_2 n)$
Pior Caso	$O(n^2)$

Tabela 13: Quick Sort

Heap Sort	
Nome	**Heap Sort**
Análise de Complexidade	
Melhor Caso	$O(n \log n)$
Médio Caso	$O(n \log n)$
Pior Caso	$O(n \log n)$

Tabela 14: Heap Sort

7 REFERÊNCIAS BIBLIOGRÁFICAS

Referências

[1] Riddhima Agarwal. Quick sort in c++ - algorithm, code example, and explanation, 2024. Accessed: December 6, 2024.

[2] Cormen. *Introduction to algorithms*. MIT press, 2022.

[3] FavTutor. Bubble sort in python: A step-by-step tutorial, 2023. Acesso em: 13 nov. 2024.

[4] Stoimen Popov. Computer algorithms: Shell sort, 2012. Acesso em: 13 nov. 2024.

[5] W3Resource. Php searching and sorting algorithm: Selection sort, 2024. Acesso em: 13 nov. 2024.

[6] Nivio Ziviani. *Projeto de algoritmos com implementações em Pascal e C*. Pioneira, 4 edition, 1999.

8 FUNÇÃO PARA CALCULAR O TEMPO

void operacoes(int tipo, int tamanho)

$clock_t start_t, end_t$; //Variavel para guardar o tempo

double tempoGasto;

int * vetor = gerarSequencia(tipo, tamanho); //Gera sequencia de numeros

//Salva a entrada de numeros

salvarEntrada(tipo, tamanho, vetor);

$start_t = clock()$; //Calcula o tempo atual

insertionSort(vetor, tamanho); //Ordena o Vetor

$end_t = clock()$; //Calcula o tempo apos ordenação

tempoGasto = (($end_t - start_t$) / (double)$CLOCKS_PER_SEC$); //Calcula diferença de tempo

salvarTempo(tipo, tamanho, tempoGasto); //Salva o tempo gasto

//Salva a saida do programa

salvarSaida(tipo, tamanho, vetor);

//Libera a memoria

free(vetor);