

Universidade Federal de Viçosa
Campus Rio Paranaíba

Eduardo Henrique Silva Oliveira - 6032

Projeto de Algoritmos

Rio Paranaíba - MG

2023

Universidade Federal de Viçosa
***Campus* Rio Paranaíba**

Eduardo Henrique Silva Oliveira - 6032

Projeto de Algoritmos

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

Rio Paranaíba - MG

2023

Resumo

Neste estudo, serão apresentadas as técnicas essenciais de ordenação na computação, desempenhando um papel vital na organização eficaz de dados, simplificando tarefas como busca, recuperação e análise de informações em diversas aplicações. A exploração inicial foca nos algoritmos clássicos, como o *Insertion Sort*, o *Bubble Sort* e o *Selection Sort*, estabelecendo os princípios fundamentais. Posteriormente, a análise se aprofunda em algoritmos avançados, como o *Quick Sort*, o *Merge Sort* e o *Heap Sort*, com um olhar detalhado sobre suas complexidades temporais e os contextos de uso ideais. O relatório oferecerá uma visão geral de cada algoritmo, acompanhada de uma análise de complexidade que explora os casos melhor, médio e pior. Além disso, serão apresentados os resultados de testes realizados por meio da implementação e execução de cada algoritmo em linguagem C, com entradas variando de 10 a 1.000.000 de elementos e ordenações em ordem crescente, decrescente e aleatória. Com base na análise dos dados de tempo de execução obtidos nos testes, será possível identificar quais algoritmos se destacam e em que situações eles demonstram eficiência.

Sumário

1	INTRODUÇÃO	5
2	ALGORITMOS	7
2.1	<i>Insertion Sort</i>	7
2.2	<i>Selection Sort</i>	9
2.3	<i>Shell Sort</i>	11
2.4	<i>Bubble Sort</i>	13
2.5	<i>Merge Sort</i>	15
2.6	<i>Quick Sort - Versão Um</i>	17
2.7	<i>Quick Sort - Versão Dois</i>	20
2.8	<i>Quick Sort - Versão Três</i>	22
2.9	<i>Quick Sort - Versão Quatro</i>	24
2.10	<i>Heap Sort</i>	26
2.11	<i>Heap Minimum</i>	29
2.12	<i>Heap Extract Min</i>	32
2.13	<i>Heap Increase Key</i>	34
2.14	<i>Max Heap Insert</i>	36
3	ANÁLISE DE COMPLEXIDADE	38
3.1	<i>Insertion Sort</i>	38
3.1.1	Melhor Caso - Ordem Crescente	39
3.1.2	Pior caso - Ordem Decrescente	39
3.1.3	Médio caso - Ordem Aleatória	40
3.2	<i>Selection Sort</i>	41
3.2.1	Melhor Caso	41
3.2.2	Pior Caso	42
3.2.3	Caso Médio	42
3.3	<i>Shell Sort</i>	43
3.3.1	Variação de Complexidade	43

3.3.2	Melhor Caso	44
3.3.3	Pior Caso	44
3.3.4	Caso Médio - Analítico	44
3.3.5	Caso Médio - Empírico	44
3.4	<i>Bubble Sort</i>	45
3.4.1	Melhor Caso - Ordem Crescente	45
3.4.2	Pior Caso - Ordem Decrescente	46
3.4.3	Caso Médio - Ordenação Aleatória	47
3.5	<i>Merge Sort</i>	48
3.5.1	Dividir	48
3.5.2	Conquistar	48
3.5.3	Mesclar	49
3.6	<i>Quick Sort</i>	50
3.6.1	Melhor Caso	50
3.6.2	Pior Caso	50
3.6.3	Caso Médio	51
3.7	<i>Heap Sort</i>	52
3.7.1	Complexidade no Melhor Caso	53
3.7.2	Complexidade no Pior Caso	53
3.7.3	Complexidade Média	53
3.8	<i>Heap Minimum</i>	54
3.8.1	Melhor Caso:	54
3.8.2	Pior Caso:	54
3.8.3	Complexidade Média:	54
3.9	<i>Heap Extract Min</i>	56
3.9.1	Melhor Caso:	56
3.9.2	Pior Caso:	56
3.9.3	Complexidade Média:	57
3.10	<i>Heap Increase Key</i>	58
3.10.1	Melhor Caso:	58

3.10.2	Pior Caso:	58
3.10.3	Complexidade Média:	59
3.11	<i>Max Heap Insert</i>	60
3.11.1	Melhor Caso:	61
3.11.2	Pior Caso:	61
3.11.3	Complexidade Média:	61
4	TABELA E GRÁFICO	62
4.1	<i>Insertion Sort</i>	62
4.2	<i>Selection Sort</i>	65
4.3	<i>Shell Sort</i>	67
4.4	<i>Bubble Sort</i>	69
4.5	<i>Merge Sort</i>	71
4.6	<i>Quick Sort - Versão Um</i>	73
4.7	<i>Quick Sort - Versão Dois</i>	75
4.8	<i>Quick Sort - Versão Três</i>	77
4.9	<i>Quick Sort - Versão Quatro</i>	79
4.10	<i>Heap Sort</i>	81
4.11	<i>Heap Minimum</i>	83
4.12	<i>Heap Extract Min</i>	85
4.13	<i>Heap Increase Key</i>	87
4.14	<i>Max Heap Insert</i>	89
4.15	Gráfico Geral	91
4.15.1	Gráfico Geral - Ordem Crescente	91
4.15.2	Gráfico Geral - Ordem Decrescente	92
4.15.3	Gráfico Geral - Ordem Aleatória	93
5	CONCLUSÃO	95

1 INTRODUÇÃO

A tecnologia da informação está relacionada à solução de problemas computacionais, nos quais a questão central envolve a relação desejada entre entrada e saída. Os algoritmos desempenham um papel crucial na eficiente transformação dessas entradas em saídas. Como definido de maneira informal por [Cormen(2012)], "um algoritmo é um procedimento computacional bem definido que aceita determinados valores como entrada e produz valores específicos como saída."

A importância dos algoritmos de ordenação na área da computação é indiscutível. Essas sequências de instruções precisas desempenham um papel vital em uma ampla gama de aplicações e têm um impacto significativo no desempenho dos sistemas de computadores. Como resumiu [Knuth(1973)], "Os algoritmos são o núcleo da tecnologia da computação". Este relatório concentra-se na análise e compreensão dos algoritmos de ordenação, que são fundamentais para a organização e classificação de dados. Com base em conceitos sólidos de algoritmos, é possível melhorar a eficiência e o desempenho dos sistemas computacionais, tornando-os essenciais para desenvolvedores e engenheiros de *software*.

Nesse cenário, os algoritmos de ordenação assumem uma posição crucial. Estas sequências de instruções precisas não apenas organizam dados, mas também simplificam tarefas complexas, como busca, recuperação e análise de informações em várias aplicações computacionais. O presente relatório tem como objetivo realizar uma análise profunda de uma série de algoritmos de ordenação. A exploração inicia-se com os algoritmos clássicos, como *Insertion Sort*, *Bubble Sort* e *Selection Sort*, estabelecendo os fundamentos essenciais. Posteriormente, a análise aprofunda-se em algoritmos avançados, incluindo o *Quick Sort*, *Merge Sort* e *Heap Sort*, examinando minuciosamente suas complexidades temporais e os cenários ideais para sua aplicação.

Para realizar uma análise detalhada, cada um desses algoritmos foi implementado e executado em linguagem C, usando entradas que variam de 10 a 1.000.000 de elementos e ordenações em ordem crescente, decrescente e aleatória. Os resultados obtidos desses testes práticos fornecem dados de tempo de execução valiosos, que são analisados meticulosamente. A partir desses resultados, identificam-se quais algoritmos se destacam em diferentes contex-

tos, oferecendo *insights* cruciais para desenvolvedores e engenheiros de *software*. A análise comparativa entre esses algoritmos não apenas evidencia suas vantagens e limitações individuais, mas também destaca a importância contínua dos algoritmos de ordenação em um mundo cada vez mais dependente de tecnologia da informação e computação.

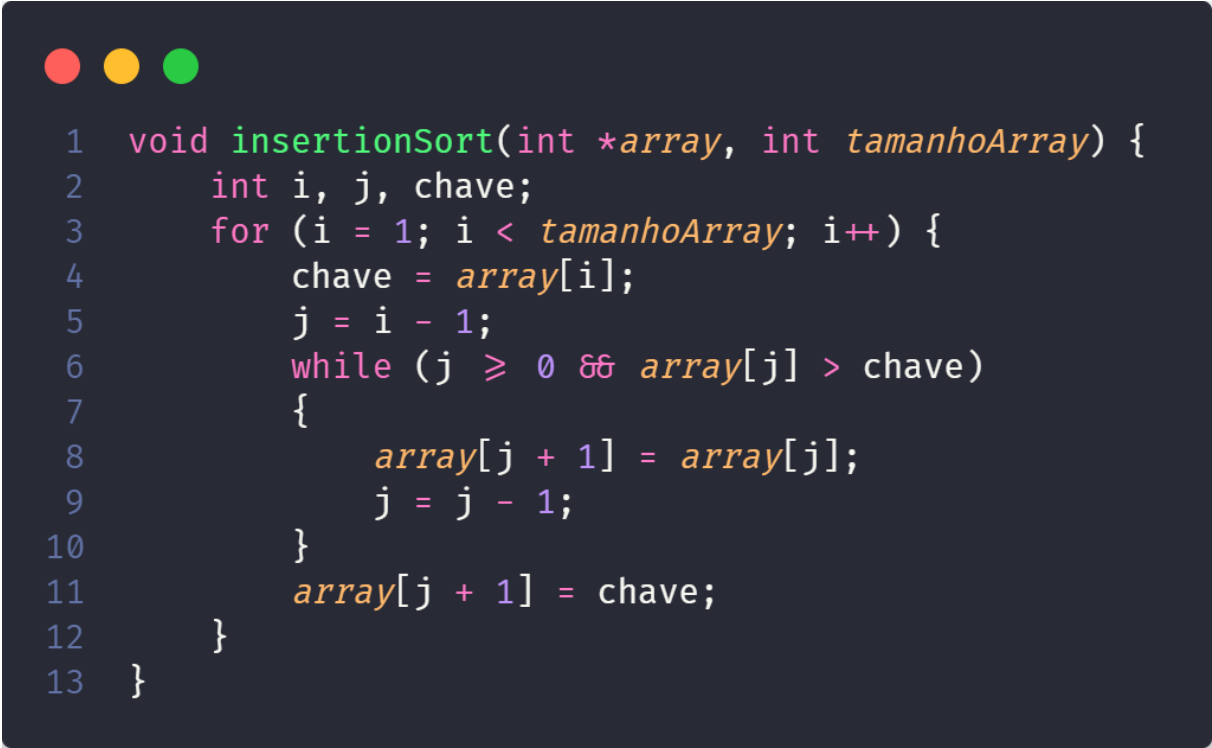
Dessa forma, este estudo não apenas enfatiza a importância dos algoritmos de ordenação, mas também proporciona uma compreensão profunda de sua aplicação prática, baseada em uma metodologia rigorosa de análise e testes. Este relatório não apenas enriquece a compreensão dos algoritmos de ordenação, mas também serve como um recurso valioso para aqueles que buscam aprimorar a eficiência e o desempenho de sistemas computacionais em um mundo digital em constante evolução.

2 ALGORITMOS

2.1 *Insertion Sort*

O algoritmo *Insertion Sort*, também conhecido como "ordenação por inserção", é uma técnica de classificação. A sua abordagem é notavelmente simples e eficaz para organizar uma lista de elementos. Este algoritmo divide a lista em duas partes distintas: uma parte já ordenada e outra desordenada. Inicialmente, a parte ordenada consiste apenas no primeiro elemento, enquanto a parte desordenada abrange todos os demais elementos.

A Figura 1 demonstra a implementação do algoritmo *Insertion Sort* em linguagem C.



```
1 void insertionSort(int *array, int tamanhoArray) {
2     int i, j, chave;
3     for (i = 1; i < tamanhoArray; i++) {
4         chave = array[i];
5         j = i - 1;
6         while (j ≥ 0 && array[j] > chave)
7         {
8             array[j + 1] = array[j];
9             j = j - 1;
10        }
11        array[j + 1] = chave;
12    }
13 }
```

Figura 1: Implementação do algoritmo *Insertion Sort* em linguagem C

O processo de classificação tem início com o segundo elemento da lista, que é comparado com os elementos presentes na parte ordenada. Caso seja maior que o elemento à sua esquerda, ele é deslocado para a direita, criando espaço para a inserção do elemento na posição adequada da parte ordenada. Esse processo é repetido sequencialmente para cada elemento subsequente até que a lista esteja completamente ordenada.

Principais características do algoritmo *Insertion Sort*:


- **Simplicidade:** É fácil de entender e implementar, tornando-o uma escolha popular para listas pequenas ou médias.
- **Eficiência:** O *Insertion Sort* tem um desempenho razoável em listas pequenas. Para listas maiores, outros algoritmos como Quick Sort e Merge Sort são geralmente preferidos devido à sua eficiência superior.
- **Estabilidade:** O algoritmo mantém a ordem relativa de elementos com chaves iguais, tornando-o estável.
- **Complexidade de tempo:** Em seu pior caso, o *Insertion Sort* possui uma complexidade de tempo de $O(n^2)$, onde n é o número de elementos na lista. No entanto, em listas quase ordenadas, pode ser muito eficiente, com uma complexidade de tempo próxima de $O(n)$.
- **Uso:** É útil quando você está trabalhando com pequenas quantidades de dados ou quando a lista já está parcialmente ordenada.

Em resumo, o algoritmo *Insertion Sort* é uma técnica de ordenação simples e intuitiva que funciona bem para listas pequenas e quase ordenadas, mas pode ser ineficiente em listas muito grandes. Ele é amplamente utilizado em situações onde a simplicidade e a estabilidade são mais importantes do que o desempenho absoluto. Além de ser uma ferramenta valiosa para ensinar os conceitos fundamentais de algoritmos de ordenação, devido à sua simplicidade.

2.2 *Selection Sort*

O algoritmo *Selection Sort*, também conhecido como "ordenação por seleção", é uma técnica simples de classificação amplamente empregada na ciência da computação. Sua abordagem é lógica e envolve a separação da lista em duas partes: uma parte já ordenada e outra ainda desordenada. Inicialmente, a parte ordenada está vazia, enquanto a parte desordenada contém todos os elementos da lista.

A Figura 2 demonstra a implementação do algoritmo *Selection Sort* em linguagem C



```
1 void selectionSort(int *array, int tamanho)
2 {
3     int i, j, min_idx;
4     for (i = 0; i < tamanho-1; i++)
5     {
6         min_idx = i;
7         for (j = i+1; j < tamanho; j++)
8             if (array[j] < array[min_idx])
9                 min_idx = j;
10        if(min_idx != i)
11            swap(&array[min_idx], &array[i]);
12    }
13 }
```

Figura 2: Implementação do algoritmo *Selection Sort* em linguagem C

O processo de classificação inicia-se com a busca do elemento mínimo na parte desordenada e, em seguida, esse elemento é deslocado para a parte ordenada, ocupando a próxima posição disponível. Esse procedimento é repetido até que todos os elementos tenham sido movidos para a parte ordenada, resultando na lista completamente classificada.

Principais características do algoritmo *Selection Sort*:

- **Simplicidade:** O *Selection Sort* é fácil de entender e implementar, tornando-o uma

escolha apropriada para listas pequenas ou quando a simplicidade é prioritária.

- **Estabilidade:** O *Selection Sort* não é naturalmente estável, o que significa que ele pode alterar a ordem relativa de elementos com chaves iguais.
- **Eficiência:** Em todos os casos, o *Selection Sort* tem uma complexidade de tempo de $O(n^2)$, onde "n" é o número de elementos na lista. Isso faz com que seja menos eficiente do que alguns outros algoritmos de ordenação, como Quick Sort e Merge Sort, em listas maiores.
- **Uso:** É útil em situações em que a complexidade de tempo não é uma preocupação crítica, como em listas muito pequenas ou quando o número de elementos a serem classificados é limitado.

Assim, o algoritmo *Selection Sort* se apresenta como uma técnica de ordenação simples, ideal para listas de pequeno porte ou situações em que a simplicidade supera a necessidade de eficiência. Ainda que não seja a primeira opção para listas mais extensas devido à sua complexidade quadrática, ele permanece uma ferramenta valiosa em cenários específicos de ordenação.

2.3 *Shell Sort*

O algoritmo *Shell Sort*, é uma estratégia de classificação desenvolvida com o objetivo de aprimorar a eficiência do algoritmo de inserção (*Insertion Sort*), especialmente em listas de grande extensão. Foi idealizado por Shell em 1959 [Shell(1959)] e se enquadra na categoria de algoritmos de ordenação por comparação.

A Figura 3 demonstra a implementação do algoritmo *Shell Sort* em linguagem C



```
1  int shellSort(int *array, int tamanho)
2  {
3      int gap;
4      for (gap = tamanho/2; gap > 0; gap /= 2)
5      {
6          int i;
7          for (i = gap; i < tamanho; i += 1)
8          {
9              int temp = array[i];
10             int j;
11             for (j = i; j >= gap && array[j - gap] > temp; j -= gap)
12                 array[j] = array[j - gap];
13             array[j] = temp;
14         }
15     }
16     return 0;
17 }
```

Figura 3: Implementação do algoritmo *Shell Sort* em linguagem C

O *Shell Sort* se destaca por sua abordagem única de dividir a lista em múltiplas sublistas menores, cada uma delas determinada por um intervalo denominado "*gap*". Inicialmente, o valor de "*gap*" é selecionado de forma a possibilitar a comparação e a realocação de elementos que estão distantes uns dos outros na lista. O algoritmo, então, aplica o *Insertion Sort* em cada uma dessas sublistas. À medida que o processo avança, o valor de "*gap*" é gradativamente reduzido em direção a 1. Esse procedimento de refinamento progressivo culmina na ordenação completa da lista.

Principais características do algoritmo *Shell Sort*:

- **Melhoria em Relação ao Insertion Sort:** O *Shell Sort* representa uma evolução

do algoritmo de inserção, notabilizando-se por sua capacidade de minimizar a quantidade de movimentos necessários para ordenar elementos, tornando-o particularmente eficiente em listas extensas.

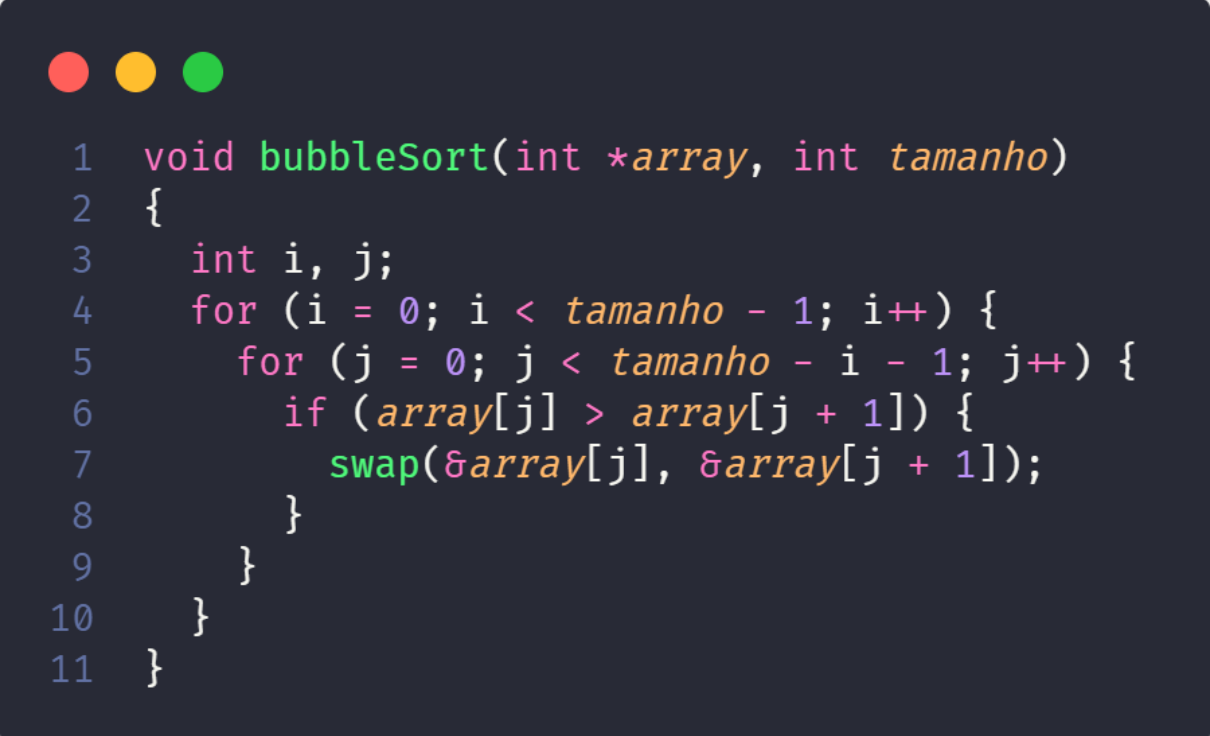
- **Intervalos Variáveis:** A inovação central do *Shell Sort* é a adoção de intervalos "gap" variáveis, que possibilitam a comparação de elementos distantes, contribuindo para um desempenho aprimorado e uma convergência mais rápida do processo de ordenação.
- **Complexidade de Tempo:** A complexidade de tempo do *Shell Sort* depende da escolha dos intervalos "gap". Embora seja desafiador estabelecer uma complexidade precisa, a maioria das implementações exibe um desempenho superior ao do Insertion Sort, com uma complexidade média de tempo que varia de $O(n)$ a $O(n^2)$.
- **Áreas de Aplicação:** O *Shell Sort* destaca-se em cenários nos quais a estabilidade da ordenação não é de importância crítica, mas a eficiência é uma necessidade, especialmente em listas de tamanho moderado a grande.

Resumidamente, o algoritmo *Shell Sort* é uma abordagem de ordenação que aprimora o *Insertion Sort* ao minimizar a movimentação de elementos e ao oferecer um desempenho melhor em listas extensas, graças ao uso de intervalos variáveis. Embora não seja o algoritmo mais eficiente em termos de complexidade de tempo, o *Shell Sort* é uma opção confiável em situações onde a facilidade de implementação e um desempenho moderado são fatores importantes.

2.4 *Bubble Sort*

O algoritmo *Bubble Sort*, também conhecido como "ordenação por bolha," é um método simples de classificação frequentemente utilizado em ciência da computação. Sua abordagem é intuitiva, porém, não é muito eficiente em listas grandes.

A Figura 4 demonstra a implementação do algoritmo *Bubble Sort* em linguagem C.



```
1 void bubbleSort(int *array, int tamanho)
2 {
3     int i, j;
4     for (i = 0; i < tamanho - 1; i++) {
5         for (j = 0; j < tamanho - i - 1; j++) {
6             if (array[j] > array[j + 1]) {
7                 swap(&array[j], &array[j + 1]);
8             }
9         }
10    }
11 }
```

Figura 4: Implementação do algoritmo *Bubble Sort* em linguagem C

A ideia central do *Bubble Sort* é percorrer a lista várias vezes, comparando pares de elementos adjacentes e trocando-os se estiverem fora de ordem. Isso resulta em bolhas de elementos maiores "subindo" para o topo da lista e bolhas de elementos menores "descendo" para o fundo. Esse processo é repetido até que a lista esteja completamente ordenada, sem a necessidade de dividir a lista em partes ordenadas e desordenadas.

Principais características do algoritmo *Bubble Sort*:

- **Simplicidade:** O *Bubble Sort* é conhecido por sua simplicidade, o que o torna uma escolha comum em contextos educacionais e em situações onde a facilidade de compreensão e implementação é mais valorizada do que a eficiência.

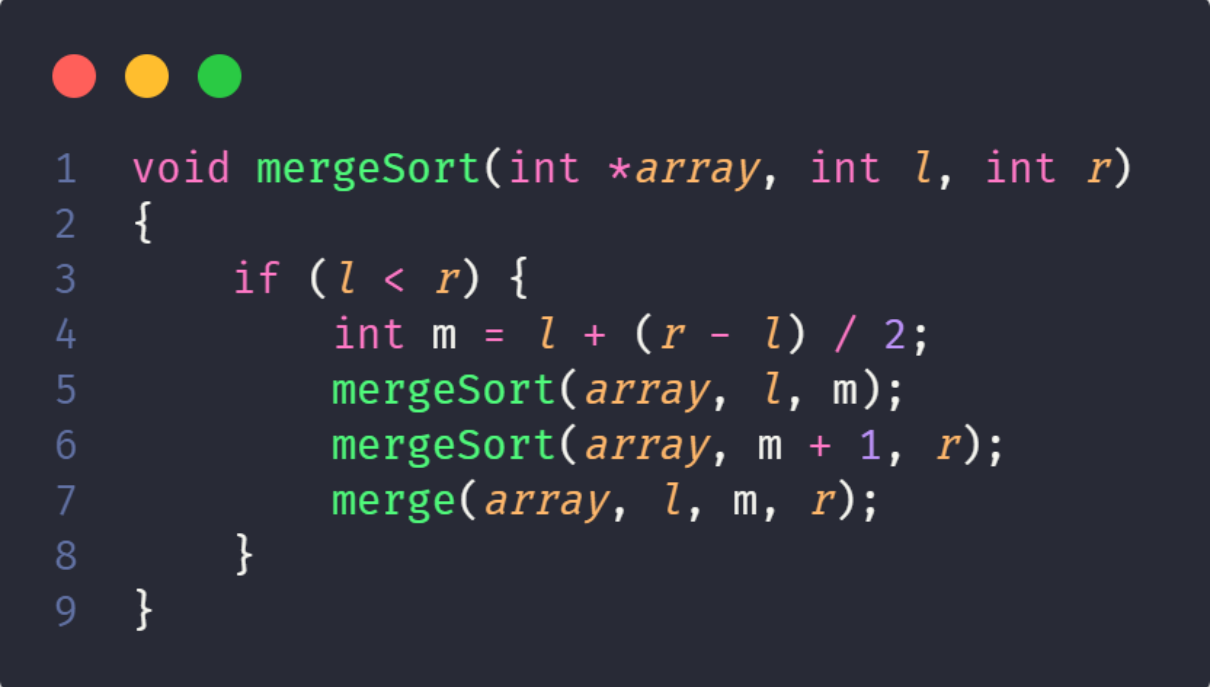
- **Estabilidade:** Esse algoritmo é estável, o que significa que ele preserva a ordem relativa de elementos com chaves iguais durante o processo de ordenação.
- **Complexidade de Tempo:** Em seu pior caso, o *Bubble Sort* apresenta uma complexidade de tempo de $O(n^2)$, onde "n" representa o número de elementos na lista. Essa característica faz com que o *Bubble Sort* seja ineficiente em listas extensas, tornando-o mais adequado para listas de pequeno a médio porte ou listas parcialmente ordenadas.
- **Uso:** Embora o *Bubble Sort* seja evitado em cenários de produção devido à sua ineficiência em listas grandes, ele ainda tem utilidade em situações simples ou como uma introdução à lógica de algoritmos de ordenação mais complexos durante o aprendizado.

O algoritmo *Bubble Sort* é uma técnica de classificação simples, adequada para listas pequenas ou quando a eficiência não é uma preocupação crítica. Porém, em listas grandes, algoritmos de ordenação mais eficientes, como *Quick Sort* e *Merge Sort*, são preferidos devido ao seu desempenho superior.

2.5 Merge Sort

O algoritmo *Merge Sort* é um método eficiente e versátil de ordenação utilizado na ciência da computação. Ele se destaca pela sua eficiência em lidar com listas de grande porte e sua abordagem de dividir para conquistar.

A Figura 5 demonstra a implementação do algoritmo *Merge Sort* em linguagem C.



```
1 void mergeSort(int *array, int l, int r)
2 {
3     if (l < r) {
4         int m = l + (r - l) / 2;
5         mergeSort(array, l, m);
6         mergeSort(array, m + 1, r);
7         merge(array, l, m, r);
8     }
9 }
```

Figura 5: Implementação do algoritmo *Merge Sort* em linguagem C

A ideia central do *Merge Sort* é dividir a lista em sublistas menores, ordená-las individualmente e, em seguida, combiná-las de forma ordenada. Isso é feito recursivamente até que a lista esteja completamente ordenada.

Principais características do algoritmo *Merge Sort*:

- **Eficiência:** O *Merge Sort* possui uma complexidade de tempo de $O(n \log n)$ no pior, médio e melhor caso, onde "n" representa o número de elementos na lista. Isso o torna eficiente mesmo em listas consideravelmente grandes.
- **Divisão e Conquista:** O algoritmo segue o paradigma de "divisão e conquista", dividindo o problema em subproblemas menores, resolvendo-os e combinando suas soluções de forma eficiente.

- **Estabilidade:** Assim como o *Bubble Sort*, o *Merge Sort* também é um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos com chaves iguais.
- **Uso:** Devido à sua eficiência em lidar com listas grandes, o *Merge Sort* é amplamente utilizado em cenários de produção onde a performance é crucial, como em aplicações que lidam com grandes volumes de dados.

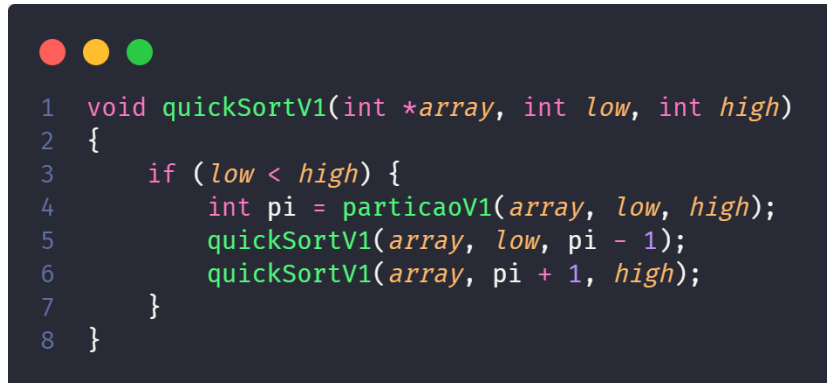
Embora o *Merge Sort* possua uma implementação complexa, sua eficiência o torna uma escolha preferencial em muitos casos. É especialmente útil quando se lida com grandes conjuntos de dados que requerem uma ordenação rápida e eficaz.

Em resumo, o algoritmo *Merge Sort* oferece uma solução poderosa e eficiente para a ordenação de listas de qualquer tamanho, tornando-o uma escolha comum em aplicações de alta performance e em contextos acadêmicos que envolvem algoritmos de ordenação mais avançados.

2.6 *Quick Sort* - Versão Um

O algoritmo *Quick Sort* é amplamente reconhecido na ciência da computação como um dos métodos de ordenação mais eficientes. Esta versão do *Quick Sort* utiliza o primeiro elemento da lista como pivô para realizar as partições.

A Figura 6 demonstra a implementação do algoritmo *Quick Sort* em linguagem C.

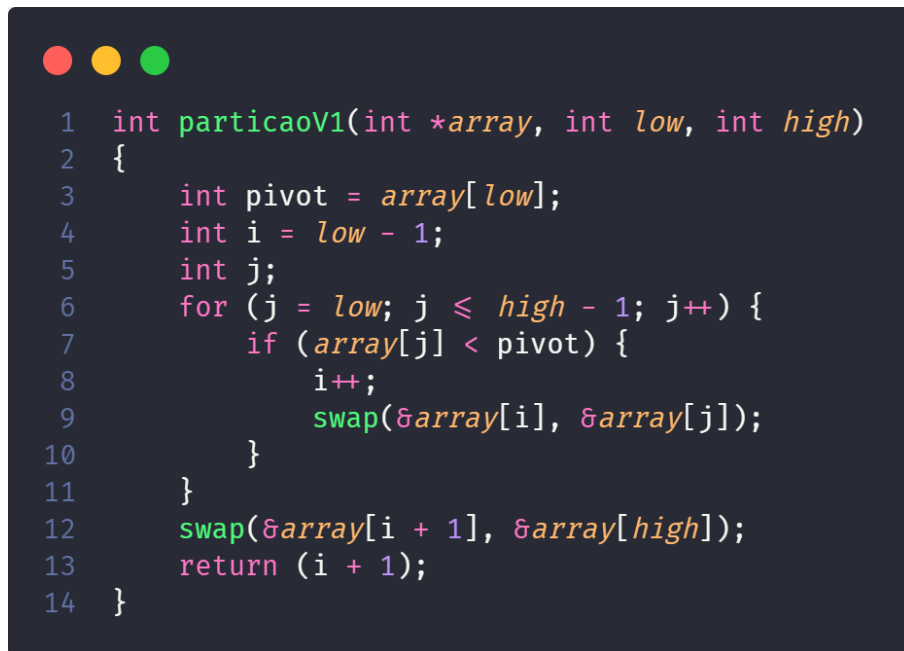


```
1 void quickSortV1(int *array, int low, int high)
2 {
3     if (low < high) {
4         int pi = particaoV1(array, low, high);
5         quickSortV1(array, low, pi - 1);
6         quickSortV1(array, pi + 1, high);
7     }
8 }
```

Figura 6: Implementação do algoritmo *Quick Sort* em linguagem C

A implementação do algoritmo é a mesma em todos os casos, a diferença está na implementação das partições de cada versão.

A Figura 7 demonstra a implementação da partição da versão 1 do algoritmo *Quick Sort* em linguagem C (com pivô no primeiro elemento).



```

1  int particaoV1(int *array, int low, int high)
2  {
3      int pivot = array[low];
4      int i = low - 1;
5      int j;
6      for (j = low; j ≤ high - 1; j++) {
7          if (array[j] < pivot) {
8              i++;
9              swap(&array[i], &array[j]);
10         }
11     }
12     swap(&array[i + 1], &array[high]);
13     return (i + 1);
14 }

```

Figura 7: Implementação da partição da versão 1 do algoritmo *Quick Sort* em linguagem C (com pivô no primeiro elemento)

A ideia central do *Quick Sort* é escolher um pivô na lista, rearranjar os elementos de forma que os menores que o pivô estejam à esquerda e os maiores estejam à direita. Em seguida, o algoritmo é aplicado recursivamente às sublistas à esquerda e à direita do pivô até que a lista esteja completamente ordenada.

Principais características do *Quick Sort* (Versão com Pivô no Primeiro Elemento):

- **Eficiência:** Em média, o *Quick Sort* possui uma complexidade de tempo de $O(n \log n)$, tornando-o muito eficiente em listas grandes. No entanto, no pior caso (quando a lista já está ordenada ou em ordem reversa), a complexidade pode chegar a $O(n^2)$.
- **Divisão e Conquista:** Segue o paradigma de "divisão e conquista", dividindo o problema em subproblemas menores, resolvendo-os e combinando suas soluções de forma eficiente.
- **Estabilidade:** O *Quick Sort* (versão com pivô no primeiro elemento) não é um algoritmo estável, o que significa que a ordem relativa de elementos com chaves iguais pode não ser preservada durante o processo de ordenação.

- **Uso:** O *Quick Sort* é amplamente utilizado em aplicações que requerem uma ordenação rápida e eficiente, especialmente quando o desempenho é crucial. No entanto, é importante considerar possíveis cenários de pior caso ao escolher este algoritmo.

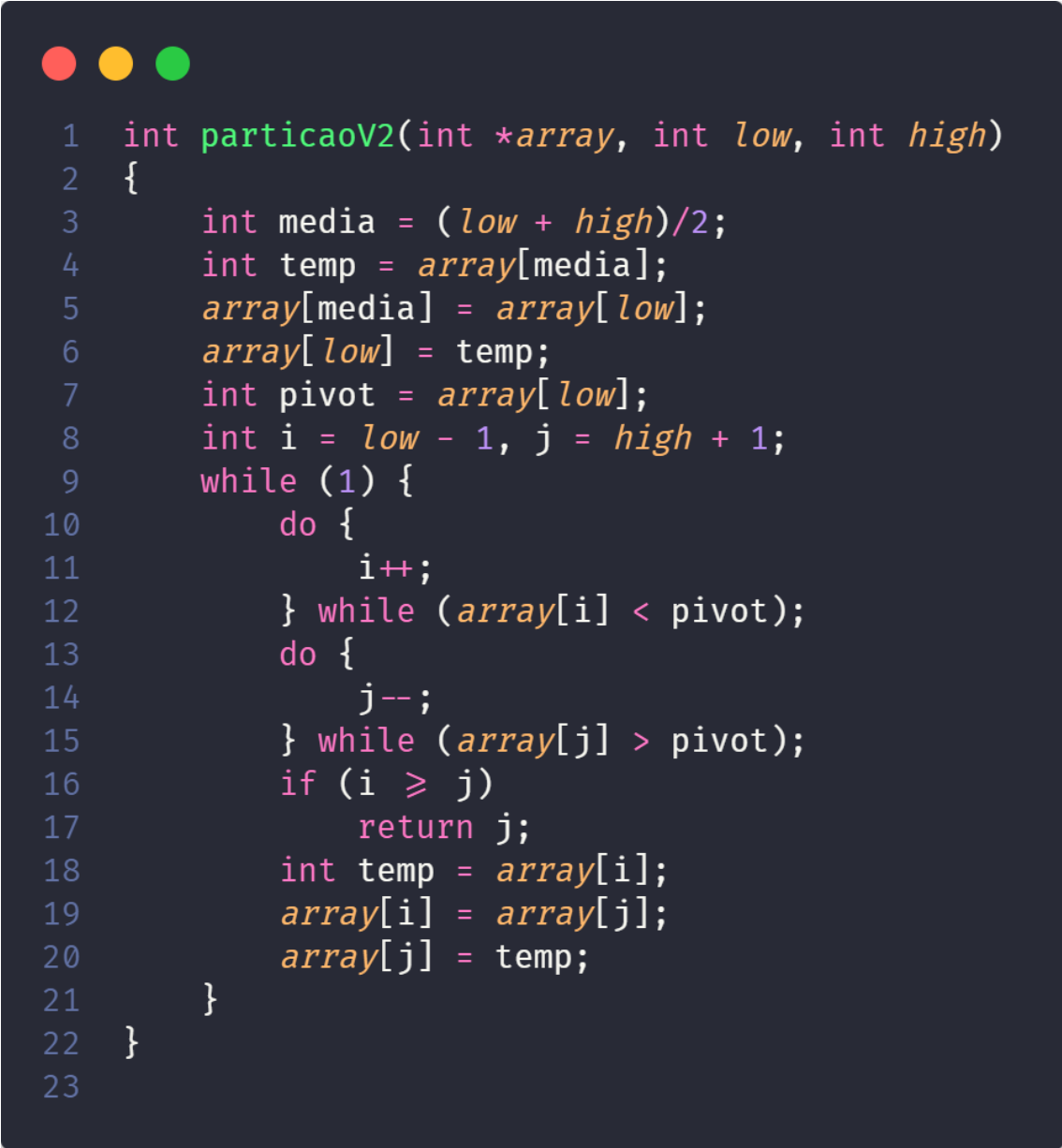
A versão do *Quick Sort* que utiliza o primeiro elemento como pivô é uma escolha popular devido à sua simplicidade de implementação e desempenho eficaz em muitos casos. No entanto, é importante ter em mente os cenários em que pode haver um desempenho menos otimizado, como em listas já ordenadas.

Em resumo, o *Quick Sort* com pivô no primeiro elemento oferece uma solução poderosa e eficiente para a ordenação de listas, sendo amplamente adotado em contextos onde a performance é crucial.

2.7 Quick Sort - Versão Dois

O algoritmo *Quick Sort* com aprimoramento na escolha do pivô é uma variação que busca melhorar o desempenho do algoritmo em cenários onde a escolha do pivô pode impactar significativamente o tempo de execução.

A Figura 8 demonstra a implementação da versão aprimorada do algoritmo *Quick Sort* em linguagem C (com escolha aprimorada do pivô).



```
1  int particaoV2(int *array, int low, int high)
2  {
3      int media = (low + high)/2;
4      int temp = array[media];
5      array[media] = array[low];
6      array[low] = temp;
7      int pivot = array[low];
8      int i = low - 1, j = high + 1;
9      while (1) {
10         do {
11             i++;
12         } while (array[i] < pivot);
13         do {
14             j--;
15         } while (array[j] > pivot);
16         if (i ≥ j)
17             return j;
18         int temp = array[i];
19         array[i] = array[j];
20         array[j] = temp;
21     }
22 }
23
```

Figura 8: Implementação da versão aprimorada do algoritmo *Quick Sort* em linguagem C (com escolha aprimorada do pivô)

A ideia central do *Quick Sort* com escolha de pivô aprimorada é calcular a média de três elementos (o primeiro, o do meio e o último) na lista e usar o valor resultante como o pivô. Isso visa minimizar os cenários em que a escolha do pivô pode levar a um desempenho menos eficaz.

Principais características do *Quick Sort* (Versão com Escolha de Pivô Aprimorada):

- **Eficiência:** Em média, o *Quick Sort* com escolha de pivô aprimorada possui uma complexidade de tempo de $O(n \log n)$, tornando-o muito eficiente em listas grandes. Ele também ajuda a reduzir a probabilidade de cenários de pior caso em comparação com a versão que utiliza o primeiro elemento como pivô.
- **Divisão e Conquista:** Mantém o paradigma de "divisão e conquista", dividindo o problema em subproblemas menores, resolvendo-os e combinando suas soluções de forma eficiente.
- **Estabilidade:** Assim como a versão anterior do *Quick Sort*, esta variação não é um algoritmo estável, o que significa que a ordem relativa de elementos com chaves iguais pode não ser preservada durante o processo de ordenação.
- **Uso:** O *Quick Sort* com escolha de pivô aprimorada é uma escolha comum em situações onde a performance é crucial e a melhoria na escolha do pivô pode trazer benefícios significativos.

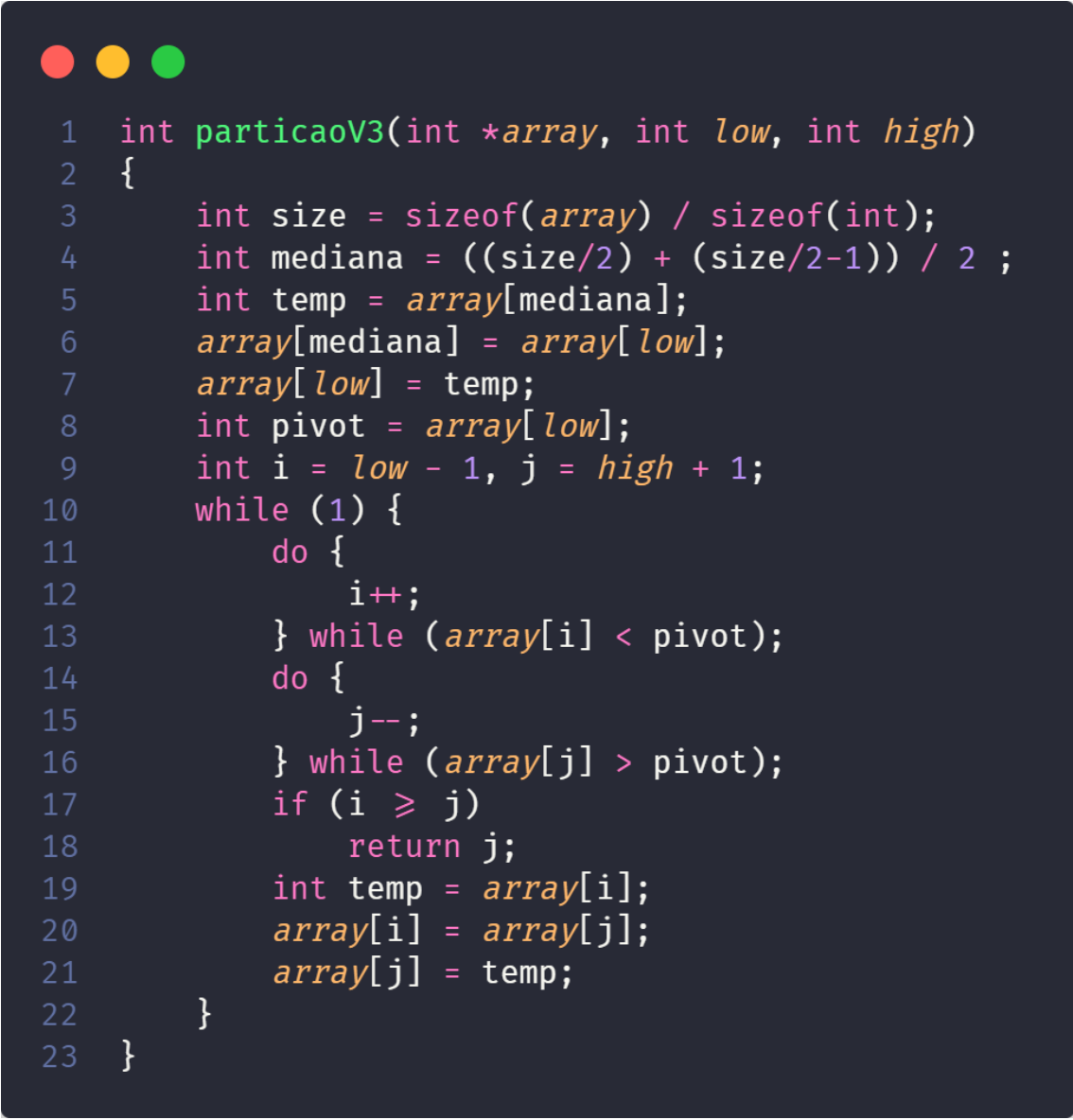
A versão aprimorada do *Quick Sort* com escolha de pivô médio oferece uma solução poderosa e eficiente para a ordenação de listas, reduzindo a probabilidade de cenários de pior caso em comparação com a versão que utiliza o primeiro elemento como pivô. É uma escolha amplamente adotada em contextos onde a performance é crucial e a otimização na escolha do pivô é desejada.

Em resumo, o *Quick Sort* com escolha de pivô aprimorada oferece uma solução eficaz para a ordenação de listas, contribuindo para a redução de cenários de pior caso e mantendo a eficiência em listas grandes.

2.8 Quick Sort - Versão Três

O algoritmo *Quick Sort* com aprimoramento na escolha do pivô pela mediana de três elementos é uma variação que visa otimizar ainda mais a eficiência do algoritmo em diferentes cenários.

A Figura 9 demonstra a implementação da versão aprimorada do algoritmo *Quick Sort* em linguagem C (com escolha da mediana de três como pivô).



```
1  int particaoV3(int *array, int low, int high)
2  {
3      int size = sizeof(array) / sizeof(int);
4      int mediana = ((size/2) + (size/2-1)) / 2 ;
5      int temp = array[mediana];
6      array[mediana] = array[low];
7      array[low] = temp;
8      int pivot = array[low];
9      int i = low - 1, j = high + 1;
10     while (1) {
11         do {
12             i++;
13         } while (array[i] < pivot);
14         do {
15             j--;
16         } while (array[j] > pivot);
17         if (i ≥ j)
18             return j;
19         int temp = array[i];
20         array[i] = array[j];
21         array[j] = temp;
22     }
23 }
```

Figura 9: Implementação da versão aprimorada do algoritmo *Quick Sort* em linguagem C (com escolha da mediana de três como pivô)

A ideia central desta versão do *Quick Sort* é calcular a mediana dos três elementos (o primeiro, o do meio e o último) na lista e utilizá-la como o pivô. Isso proporciona uma escolha ainda mais equilibrada e ajuda a minimizar os cenários em que a escolha do pivô pode impactar negativamente o tempo de execução.

Principais características do *Quick Sort* (Versão com Escolha de Pivô pela Mediana de Três):

- **Eficiência:** Em média, o *Quick Sort* com escolha de pivô pela mediana de três elementos possui uma complexidade de tempo de $O(n \log n)$, tornando-o muito eficiente em listas grandes. Ele reduz ainda mais a probabilidade de cenários de pior caso em comparação com as versões anteriores.
- **Divisão e Conquista:** Segue o paradigma de "divisão e conquista", dividindo o problema em subproblemas menores, resolvendo-os e combinando suas soluções de forma eficiente.
- **Estabilidade:** Assim como as versões anteriores do *Quick Sort*, esta variação não é um algoritmo estável, o que significa que a ordem relativa de elementos com chaves iguais pode não ser preservada durante o processo de ordenação.
- **Uso:** O *Quick Sort* com escolha de pivô pela mediana de três elementos é altamente recomendado em situações onde a performance é crucial e a otimização na escolha do pivô é fundamental.

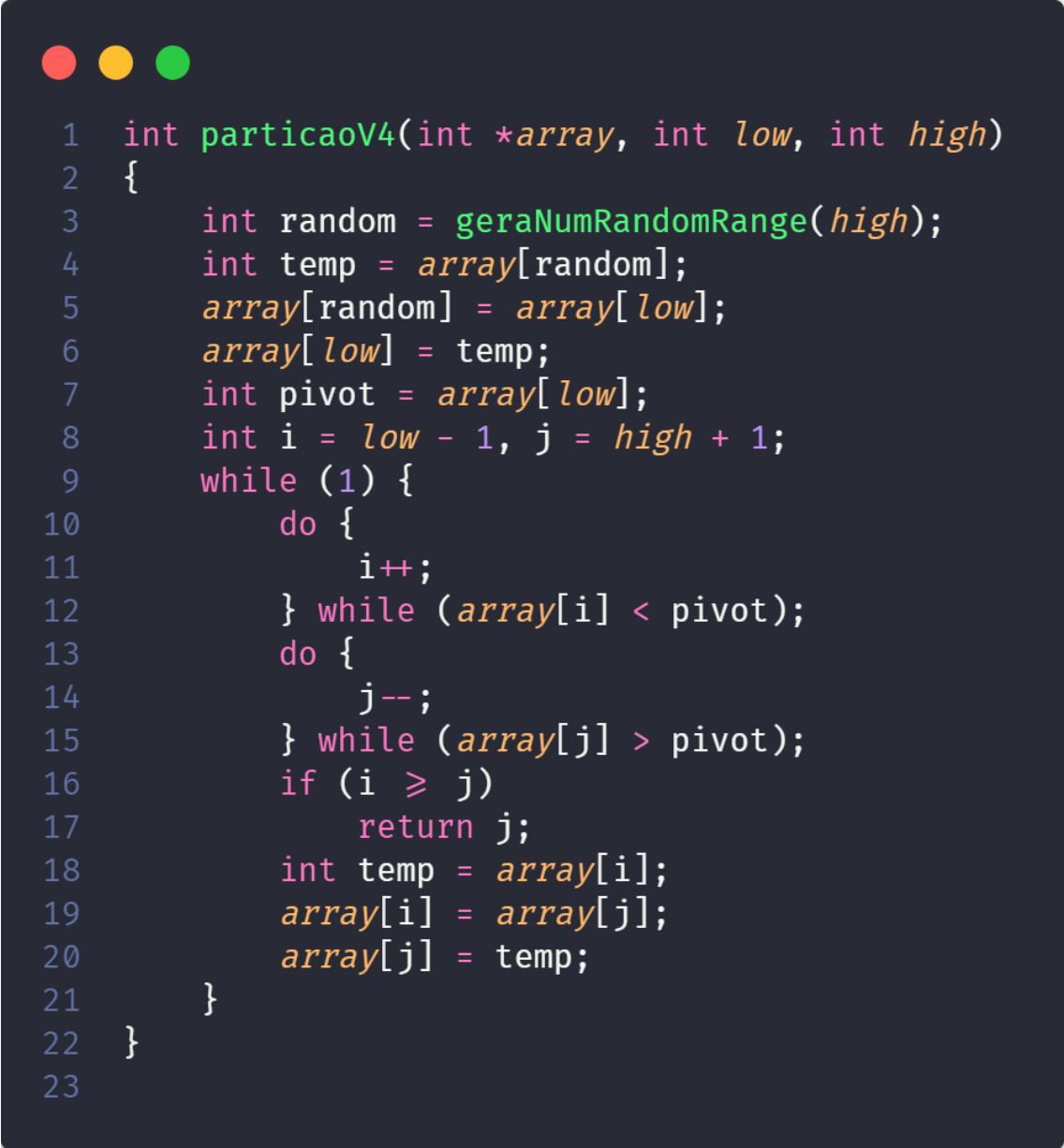
A versão aprimorada do *Quick Sort* com escolha de pivô pela mediana de três elementos oferece uma solução extremamente eficaz para a ordenação de listas, reduzindo ainda mais a probabilidade de cenários de pior caso em comparação com as versões anteriores. É uma escolha amplamente adotada em contextos onde a performance é crucial e a otimização na escolha do pivô é de grande importância.

Em resumo, o *Quick Sort* com escolha de pivô pela mediana de três elementos proporciona uma solução altamente eficiente para a ordenação de listas, contribuindo para a redução significativa de cenários de pior caso e mantendo uma excelente eficiência em listas grandes.

2.9 Quick Sort - Versão Quatro

O algoritmo *Quick Sort* com a escolha aleatória do pivô é uma variação que busca aprimorar ainda mais a eficiência do algoritmo, reduzindo a probabilidade de cenários de pior caso.

A Figura 10 demonstra a implementação da versão com escolha aleatória do pivô do algoritmo *Quick Sort* em linguagem C.



```
1  int particaoV4(int *array, int low, int high)
2  {
3      int random = geraNumRandomRange(high);
4      int temp = array[random];
5      array[random] = array[low];
6      array[low] = temp;
7      int pivot = array[low];
8      int i = low - 1, j = high + 1;
9      while (1) {
10         do {
11             i++;
12         } while (array[i] < pivot);
13         do {
14             j--;
15         } while (array[j] > pivot);
16         if (i ≥ j)
17             return j;
18         int temp = array[i];
19         array[i] = array[j];
20         array[j] = temp;
21     }
22 }
23
```

Figura 10: Implementação da versão com escolha aleatória do pivô do algoritmo *Quick Sort* em linguagem C

Nesta versão do *Quick Sort*, o pivô é escolhido de forma aleatória a partir dos elementos

da lista. Isso significa que a escolha do pivô pode variar a cada execução do algoritmo, o que ajuda a reduzir a probabilidade de cenários desfavoráveis.

Principais características do *Quick Sort* (Versão com Escolha Aleatória do Pivô):

- **Eficiência:** Em média, o *Quick Sort* com escolha aleatória do pivô possui uma complexidade de tempo de $O(n \log n)$, tornando-o muito eficiente em listas grandes. A escolha aleatória do pivô contribui para a redução da probabilidade de cenários de pior caso.
- **Divisão e Conquista:** Mantém o paradigma de "divisão e conquista", dividindo o problema em subproblemas menores, resolvendo-os e combinando suas soluções de forma eficiente.
- **Estabilidade:** Assim como as versões anteriores do *Quick Sort*, esta variação não é um algoritmo estável, o que significa que a ordem relativa de elementos com chaves iguais pode não ser preservada durante o processo de ordenação.
- **Uso:** O *Quick Sort* com escolha aleatória do pivô é altamente recomendado em situações onde a performance é crucial e a minimização de cenários de pior caso é de grande importância.

A versão do *Quick Sort* com escolha aleatória do pivô oferece uma solução extremamente eficiente para a ordenação de listas, contribuindo para a redução significativa da probabilidade de cenários de pior caso. É uma escolha amplamente adotada em contextos onde a performance é crucial e a minimização de cenários desfavoráveis é uma prioridade.

Em resumo, o *Quick Sort* com escolha aleatória do pivô proporciona uma solução altamente eficaz para a ordenação de listas, contribuindo para a redução da probabilidade de cenários de pior caso e mantendo uma excelente eficiência em listas grandes.

2.10 *Heap Sort*

O algoritmo **Heapsort** destaca-se como uma abordagem eficiente para a ordenação de conjuntos de dados, fazendo uso da estrutura de heap para garantir uma ordenação precisa e rápida. A estrutura de *heap*, uma árvore binária completa, é fundamental para o funcionamento desse algoritmo.

A Figura 11 demonstra a implementação do algoritmo *Heap Sort* em linguagem C.



```
1 void heapSort(int *arr, int tam)
2 {
3     for (int i = tam / 2 - 1; i ≥ 0; i--)
4         heapfy(arr, tam, i);
5     for (int i = tam - 1; i ≥ 0; i--) {
6         swap(&arr[0], &arr[i]);
7         heapfy(arr, i, 0);
8     }
9 }
```

Figura 11: Implementação do algoritmo *Heap Sort* em linguagem C.

No **Heapsort**, a construção do heap máximo (ou mínimo) é essencial para criar a base para a ordenação eficiente. Durante esta fase, a propriedade de heap é mantida, garantindo que cada nó seja maior (ou menor) do que seus filhos, conforme a necessidade.

Manutenção do *Heap* durante a Construção:

1. Construção do *Heap* Máximo:

- O algoritmo percorre o *array* de trás para frente.
- A cada passo, o elemento é comparado com seus filhos, e se necessário, ocorre a troca para manter a propriedade de heap.

Manutenção do *Heap* durante a Extração:

2. Extração do Elemento Máximo:

- O elemento máximo (localizado na raiz) é trocado com o último elemento não ordenado do *array*.
- Em seguida, uma operação de "heapify" é realizada para restaurar a propriedade de *heap*.

Filas de Prioridades e Mínimos:

A estrutura de *heap* pode ser associada a uma fila de prioridades, onde o elemento máximo (ou mínimo) tem a maior prioridade. Esta abordagem é valiosa em situações que demandam ordenação com base em prioridades.

- **Fila de Prioridades:** A estrutura de *heap* pode ser interpretada como uma fila de prioridades, onde o elemento no topo tem a maior prioridade.
- **Manutenção com Mínimos:** A estrutura do *heap* pode ser ajustada para suportar a manutenção de mínimos, útil em situações que exigem a ordenação em ordem crescente.

Complexidade Temporal:

O *Heapsort* apresenta uma complexidade temporal de $O(n \log n)$ para o pior, médio e melhor caso. Essa eficiência é alcançada devido à estrutura de *heap*, que permite a divisão e conquista do problema.

Características Principais do *Heapsort*:

- **Eficiência Temporal:** $O(n \log n)$, tornando-o eficiente para grandes conjuntos de dados.
- **Estrutura de *Heap*:** Utiliza a estrutura de *heap* para manutenção eficiente durante a ordenação.

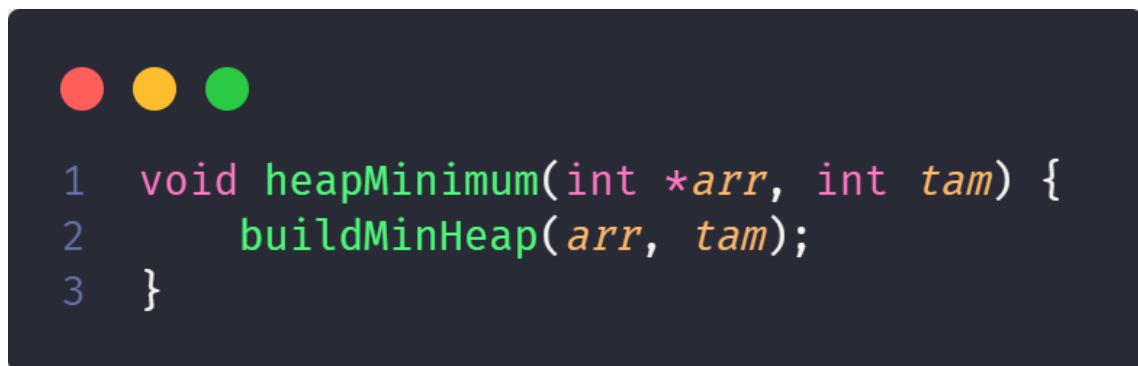
- ***In-Place***: Realiza a ordenação no próprio *array*, sem a necessidade de espaço adicional.
- **Estabilidade Relativa**: Não é um algoritmo estável, ou seja, a ordem relativa de elementos iguais pode não ser preservada.

O ***Heapsort*** é uma escolha robusta em cenários onde a eficiência na ordenação é crucial. Sua abordagem in-place e o uso inteligente da estrutura de *heap* fazem dele uma ferramenta valiosa em contextos diversos. Em resumo, o ***Heapsort*** oferece uma solução eficiente para a ordenação de listas, com características que o tornam altamente recomendado em situações que demandam desempenho e minimização de cenários desfavoráveis.

2.11 *Heap Minimum*

O algoritmo ***Heap Minimum*** destaca-se como uma abordagem eficiente para encontrar o elemento mínimo em uma estrutura de dados conhecida como *heap* mínimo. A estrutura de *heap* mínimo, assim como o *heap* máximo mencionado no contexto do *Heapsort*, é uma árvore binária completa que mantém a propriedade fundamental de que o valor de cada nó é menor ou igual aos valores de seus filhos.

A Figura 12 demonstra a implementação do algoritmo *Heap Minimum* em linguagem C.

A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. Below them, the C code for the heapMinimum function is displayed with syntax highlighting. The code consists of three lines: a function signature, a call to buildMinHeap, and a closing brace. The line numbers 1, 2, and 3 are shown in blue on the left margin.

```
1 void heapMinimum(int *arr, int tam) {  
2     buildMinHeap(arr, tam);  
3 }
```

Figura 12: Implementação do algoritmo *Heap Minimum* em linguagem C.

No ***Heap Minimum***, a operação fundamental é encontrar o valor mínimo armazenado no nó raiz do *heap*. Essa operação é crucial em contextos que exigem a identificação rápida do menor elemento em uma coleção de dados. A eficiência do algoritmo é notável, pois a complexidade temporal para encontrar o mínimo é constante, independentemente do tamanho do *heap*.

Construção do *Heap* Mínimo:

Durante a construção do *heap* mínimo, a propriedade de *heap* é mantida. Cada nó é comparado com seus filhos, e se necessário, ocorre a troca para garantir que o valor de cada nó seja menor ou igual ao valor de seus filhos.

Manutenção do *Heap* durante a Construção:

1. Construção do *Heap* Mínimo:

- O algoritmo percorre o *array* de trás para frente.
- A cada passo, o elemento é comparado com seus filhos, e se necessário, ocorre a troca para manter a propriedade de *heap*.

Extração do Elemento Mínimo:

Após a construção, a operação principal é a extração do elemento mínimo, que está localizado na raiz do *heap*. Esta operação envolve a troca do elemento mínimo com o último elemento não ordenado do *array*, seguida por uma operação de “heapify” para restaurar a propriedade de *heap*.

Manutenção do *Heap* durante a Extração:

2. Extração do Elemento Mínimo:

- O elemento mínimo (localizado na raiz) é trocado com o último elemento não ordenado do *array*.
- Em seguida, uma operação de “heapify” é realizada para restaurar a propriedade de *heap*.

Utilização em Filas de Prioridade:

A estrutura de *heap* mínimo pode ser associada a uma fila de prioridades, onde o elemento mínimo tem a maior prioridade. Essa abordagem é valiosa em situações que demandam ordenação com base em prioridades.

- **Fila de Prioridades:** A estrutura de *heap* mínimo pode ser interpretada como uma fila de prioridades, onde o elemento no topo tem a maior prioridade.
- **Manutenção com Mínimos:** A estrutura do *heap* mínimo pode ser ajustada para suportar a manutenção de mínimos, útil em situações que exigem a ordenação em ordem crescente.

Complexidade Temporal:

O ***Heap Minimum*** apresenta uma complexidade temporal constante, $O(1)$, para encontrar o mínimo, tornando-o eficiente em cenários que exigem a identificação rápida do menor elemento.

Em resumo, o algoritmo ***Heap Minimum*** é uma ferramenta eficiente para encontrar o elemento mínimo em uma coleção de dados. Sua implementação inteligente da estrutura de *heap* mínimo proporciona um desempenho notável, sendo valioso em contextos que demandam eficiência na identificação do menor elemento.

2.12 *Heap Extract Min*

A operação de extração do mínimo (*Heap Extract Min*) é uma parte crucial do funcionamento eficiente de um *heap* mínimo. Esta operação envolve a remoção do elemento mínimo, localizado na raiz do *heap*, e a restauração da propriedade de *heap* após a remoção.

A Figura 13 demonstra a implementação do algoritmo *Heap Extract Min* em linguagem C.



```
1 void heapExtractMin(int *arr, int tam) {  
2     buildMinHeap(arr, tam);  
3     minHeapfy(arr, tam);  
4 }
```

Figura 13: Implementação do algoritmo *Heap Extract Min* em linguagem C.

Procedimento de Extração do Mínimo:

Ao realizar a extração do mínimo em um *heap* mínimo, os seguintes passos são seguidos:

1. Troca com o Último Elemento Não Ordenado:

- O elemento mínimo, localizado na raiz do *heap*, é trocado com o último elemento não ordenado do *array*.

2. Restauração da Propriedade de *Heap*:

- Após a troca, uma operação de *heapify* é realizada para garantir que a propriedade de *heap* seja mantida.
- Durante a operação de *heapify*, o elemento trocado é comparado com seus filhos, e, se necessário, são realizadas trocas adicionais para manter a propriedade de *heap*.

O procedimento de extração do mínimo é essencial em situações em que se deseja acessar e remover repetidamente o elemento de menor valor em uma coleção de dados. A eficiência

dessa operação, aliada à estrutura de *heap* mínimo, torna o *Heap Extract Min* valioso em algoritmos que requerem a ordenação ou manipulação de dados com base em prioridades.

Utilização em Estruturas de Dados:

Além de sua aplicação direta em *heaps* mínimos, a operação de extração do mínimo é fundamental em estruturas de dados que se baseiam em *heaps* mínimos, como filas de prioridades mínimas. A capacidade de extrair o elemento mínimo de forma eficiente torna essas estruturas de dados valiosas em uma variedade de contextos computacionais.

Complexidade Temporal:

A complexidade temporal da operação de *Heap Extract Min* é dominada pela operação de *heapify*. Em um *heap* mínimo, a complexidade temporal da *heapify* é geralmente da ordem de $O(\log n)$, onde n é o número de elementos no *heap*.

A operação de extração do mínimo em um *heap* mínimo é, portanto, uma operação eficiente, especialmente em comparação com abordagens lineares para encontrar e remover o menor elemento em uma coleção de dados não ordenada.

2.13 *Heap Increase Key*

A operação de aumento de chave (*Heap Increase Key*) é um componente essencial em algoritmos e estruturas de dados que fazem uso de *heaps*, especialmente em contextos nos quais é necessário aumentar o valor de um elemento específico no *heap*. Essa operação é frequentemente utilizada em conjunto com a operação de inserção, proporcionando flexibilidade na manipulação de valores armazenados no *heap*.

A Figura 14 demonstra a implementação do algoritmo *Heap Increase Key* em linguagem C.



```
1 void heapIncreaseKey(int *arr, int tam) {
2     buildMinHeap(arr, tam);
3     while(1) {
4         printf("Digite a posicao desejada: (de 0 a %d)\n", tam - 1);
5         scanf("%d", &posicao);
6         if(posicao > tam - 1 || posicao < 0)
7             printf("Valor invalido! Digite novamente de 0 a %d\n", tam - 1);
8         else
9             break;
10    }
11    arr[posicao] = 1000;
12    buildMinHeap(arr, tam);
13 }
```

Figura 14: Implementação do algoritmo *Heap Increase Key* em linguagem C.

Procedimento de Aumento de Chave:

O aumento de chave no *heap* envolve os seguintes passos:

1. Atualização do Valor do Elemento:

- O valor do elemento no *heap* que se deseja aumentar é modificado para um novo valor mais alto.

2. Restauração da Propriedade de *Heap*:

- Após a atualização do valor, uma operação de *heapify-up* é realizada para garantir que a propriedade de *heap* seja mantida.

- Durante a operação de *heapify-up*, o elemento é comparado com seu pai, e se necessário, são realizadas trocas para manter a propriedade de *heap*.

O procedimento de aumento de chave é fundamental em situações em que é necessário ajustar dinamicamente a prioridade de um elemento específico. Isso é particularmente útil em algoritmos que envolvem seleção ou manipulação de elementos com base em suas prioridades ou valores associados.

Utilização em Estruturas de Dados:

Além de sua aplicação direta em *heaps*, a operação de aumento de chave é crucial em estruturas de dados como filas de prioridades, onde a necessidade de atualizar a prioridade de um elemento é comum. A capacidade de aumentar eficientemente a chave de um elemento mantendo a propriedade de *heap* é valiosa em diversos cenários computacionais.

Complexidade Temporal:

A complexidade temporal da operação de *Heap Increase Key* é predominantemente determinada pela operação de *heapify-up*. Em um *heap*, a complexidade temporal da *heapify-up* é geralmente da ordem de $O(\log n)$, onde n é o número de elementos no *heap*.

O aumento de chave em um *heap* é, portanto, uma operação eficiente, proporcionando flexibilidade na manipulação dinâmica de valores associados aos elementos do *heap*.

2.14 *Max Heap Insert*

A operação de inserção em um *heap* máximo (*Max Heap Insert*) é essencial quando se deseja adicionar um novo elemento à estrutura mantendo a propriedade de *heap* máximo. Essa operação é crucial em algoritmos e estruturas de dados que fazem uso de *heaps* máximos, permitindo a incorporação dinâmica de novos elementos enquanto mantém a estrutura eficiente do *heap*.

A Figura 15 demonstra a implementação do algoritmo *Max Heap Insert* em linguagem C.



```
1 void maxHeapInsert(int *arr, int tam) {
2     int i, pai;
3     int novoValor=1000;
4     tam++;
5     arr[tam - 1] = novoValor;
6     buildMinHeap(arr, tam);
7     i = tam - 1;
8     while (i > 0) {
9         pai = (i - 1) / 2;
10        if (arr[i] > arr[pai]) {
11            int temp = arr[i];
12            arr[i] = arr[pai];
13            arr[pai] = temp;
14            i = pai;
15        } else {
16            break;
17        }
18    }
19 }
```

Figura 15: Implementação do algoritmo *Max Heap Insert* em linguagem C.

Procedimento de Inserção em Heap Máximo:

O processo de inserção em um *heap* máximo envolve os seguintes passos:

1. Adição do Novo Elemento:

- Um novo elemento é adicionado ao final do *heap* máximo, inicialmente posicionado como o último elemento não ordenado.

2. Restauração da Propriedade de *Heap* Máximo:

- Após a adição do novo elemento, uma operação de *heapify-up* é executada para garantir que a propriedade de *heap* máximo seja mantida.
- Durante a operação de *heapify-up*, o novo elemento é comparado com seu pai, e se necessário, são realizadas trocas para manter a propriedade de *heap* máximo.

O procedimento de inserção em *heap* máximo é fundamental para a expansão dinâmica da estrutura, possibilitando a adição de elementos de maneira eficiente.

Utilização em Estruturas de Dados:

Além de sua aplicação direta em *heaps* máximos, a operação de inserção é crucial em estruturas de dados como filas de prioridades máximas. Essa operação proporciona uma maneira eficiente de adicionar novos elementos à estrutura, mantendo a ordem de prioridade.

Complexidade Temporal:

A complexidade temporal da operação de *Max Heap Insert* é principalmente determinada pela operação de *heapify-up*. Em um *heap* máximo, a complexidade temporal da *heapify-up* é geralmente da ordem de $O(\log n)$, onde n é o número de elementos no *heap*.

A inserção em um *heap* máximo é, portanto, uma operação eficiente, permitindo a expansão dinâmica da estrutura enquanto mantém a propriedade fundamental de *heap* máximo.

3 ANÁLISE DE COMPLEXIDADE

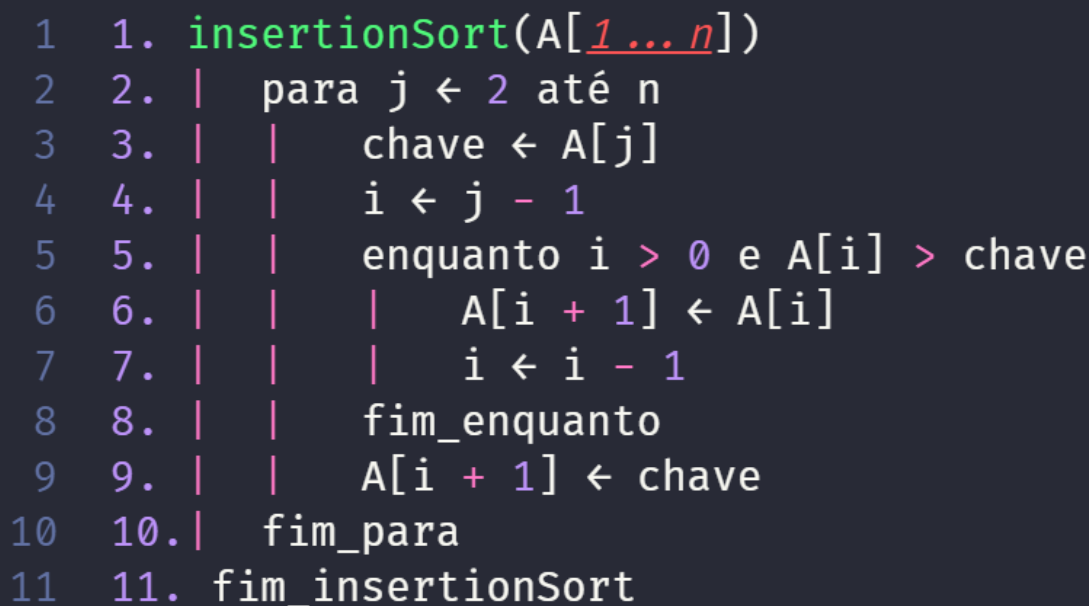
O tempo de execução de um algoritmo é calculado somando-se os tempos de execução individuais de cada instrução. Cada instrução que consome c_i passos e é executada n vezes contribui com $c_i \cdot n$ para o tempo total de execução.

Para determinar $T(n)$, o tempo de execução do algoritmo em uma entrada de n valores, multiplicamos o custo de cada instrução pelo número de vezes que ela é executada e somamos esses produtos para obter o tempo de execução total.

3.1 *Insertion Sort*

O algoritmo de ordenação *Insertion Sort* é amplamente utilizado devido à sua simplicidade e é influenciado principalmente pelo tamanho da entrada e pela disposição dos elementos no *array* de entrada, o que afeta sua eficiência e desempenho.

A Figura 16 apresenta um pseudocódigo do algoritmo *Insertion Sort*.



```
1  1. insertionSort(A[1...n])
2  2. |   para j ← 2 até n
3  3. |   |   chave ← A[j]
4  4. |   |   i ← j - 1
5  5. |   |   enquanto i > 0 e A[i] > chave
6  6. |   |   |   A[i + 1] ← A[i]
7  7. |   |   |   i ← i - 1
8  8. |   |   fim_enquanto
9  9. |   |   A[i + 1] ← chave
10 10. |   fim_para
11 11. fim_insertionSort
```

Figura 16: Pseudocódigo do algoritmo *Insertion Sort*

3.1.1 Melhor Caso - Ordem Crescente

Em relação ao algoritmo *Insertion Sort*, o melhor caso ocorre quando o *array* de entrada já está completamente ordenado em ordem crescente. Nesse cenário, o algoritmo faz apenas comparações entre os elementos, mas não realiza nenhuma movimentação. Isso significa que, para cada $j = 2, 3, \dots, n$, descobrimos que $A[i] \leq \text{chave}$ na linha 5 quando i tem seu valor inicial $j - 1$. Portanto, $t_j = 1$ para $j = 2, 3, \dots, n$, e o tempo de execução do melhor caso é

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Podemos representar esse tempo de execução como uma função linear de n , expressa como $an + b$, onde a e b são constantes que dependem dos custos das instruções c_i .

3.1.2 Pior caso - Ordem Decrescente

O pior caso ocorre quando o *array* de entrada está completamente ordenado em ordem decrescente. Nesse cenário, o algoritmo realiza o máximo de comparações e movimentações possíveis. Precisamos comparar cada elemento $A[j]$ com todos os elementos do subarranjo completamente ordenado $A[1..j-1]$, e, portanto, para $j = 2, 3, \dots, n$, temos $t_j = j$. Considerando que

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

e

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

O tempo de execução de *Insertion Sort* no pior caso é

$$\begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 \\ & + (c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Podemos expressar o tempo de execução no pior caso como uma função quadrática ($O(n^2)$), representada por $an^2 + bn + c$, onde a , b e c são constantes que, mais uma vez, dependem dos custos das instruções c_i .

3.1.3 Médio caso - Ordem Aleatória

O caso médio é quase tão ruim quanto o pior caso. Ao escolher aleatoriamente n números e aplicar o algoritmo de ordenação por inserção, o tempo necessário para que o algoritmo determine a posição no subarranjo $A[1..j-1]$ onde o elemento $A[j]$ deve ser inserido é, em média, $t_j = \frac{j}{2}$. Isso implica que o tempo de execução médio é uma função quadrática do tamanho da entrada, semelhante ao tempo de execução no pior caso. O tempo de execução no caso médio é

$$\sum_2^n \frac{j}{2} = \frac{1}{2} \sum_{j=2}^n j = \frac{1}{2} \left(\frac{n(n+1)}{2} - 1 \right)$$

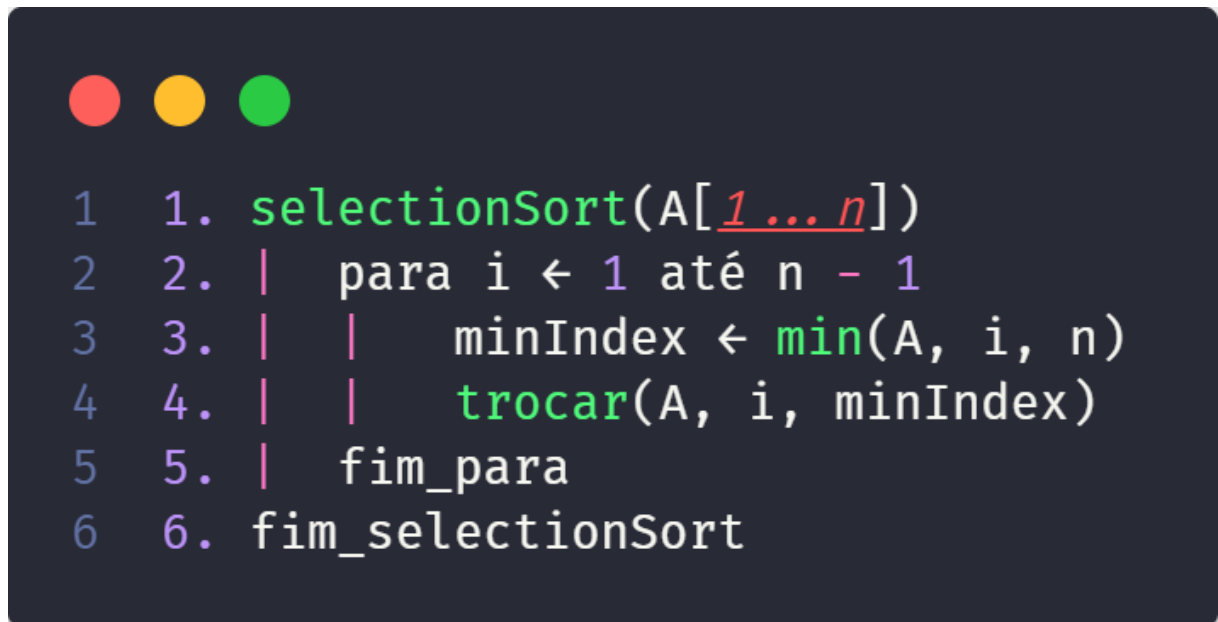
$$\sum_2^n \frac{j-1}{2} = \frac{1}{2} \sum_2^n (j-1) = \frac{1}{2} \left(\sum_2^n j - \sum_2^n 1 \right) = \frac{1}{2} \left(\frac{n^2 - n}{2} \right) = \frac{n^2 - n}{4}$$

Em resumo, os tempos de execução do *Insertion Sort* são altamente dependentes da ordenação inicial dos elementos no *array*. O melhor caso ocorre quando o *array* já está ordenado crescente, enquanto o pior caso ocorre quando o *array* está ordenado em ordem inversa decrescente. O desempenho em um cenário aleatório fica entre esses dois extremos. Para grandes conjuntos de dados, o *Insertion Sort* pode se tornar impraticável, especialmente no pior caso, e outros algoritmos de ordenação mais eficientes podem ser preferíveis.

3.2 *Selection Sort*

O algoritmo de ordenação *Selection Sort* é conhecido por sua simplicidade, mas também possui uma complexidade que o limita, independentemente da ordem inicial dos elementos no *array* de entrada.

A Figura 17 apresenta um pseudocódigo do algoritmo *Selection Sort*.

The image shows a dark-themed code editor with three colored window control buttons (red, yellow, green) at the top left. The pseudocode for Selection Sort is written in a monospaced font with syntax highlighting: line numbers 1-6 are in purple, keywords are in green, and variable names are in white. The code is as follows:

```
1  1. selectionSort(A[1...n])
2  2. |   para i ← 1 até n - 1
3  3. |   |   minIndex ← min(A, i, n)
4  4. |   |   trocar(A, i, minIndex)
5  5. |   fim_para
6  6. fim_selectionSort
```

Figura 17: Pseudocódigo do algoritmo *Selection Sort*

A quantidade de comparações $A[\text{minIndex}] > A[j]$ realizadas pelo Selection Sort é expressa por

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n 1 = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Portanto, conclui-se que a complexidade é $O(n^2)$. Mesmo ao considerar as demais operações, a mesma complexidade é obtida.

3.2.1 Melhor Caso

No melhor caso, o *Selection Sort* realiza o mesmo número de comparações e trocas que no caso médio e no pior caso. Portanto, o tempo de execução no melhor caso é:

$$T(n) = (c_1 + c_2 + c_3 + c_4)n^2$$

3.2.2 Pior Caso

No pior caso, o *Selection Sort* também realiza o mesmo número de comparações e trocas que nos casos médio e melhor. Portanto, o tempo de execução no pior caso é:

$$T(n) = (c_1 + c_2 + c_3 + c_4)n^2$$

3.2.3 Caso Médio

No caso médio, o *Selection Sort* também demonstra um desempenho semelhante ao melhor e pior caso. Resultando em um tempo de execução médio de:

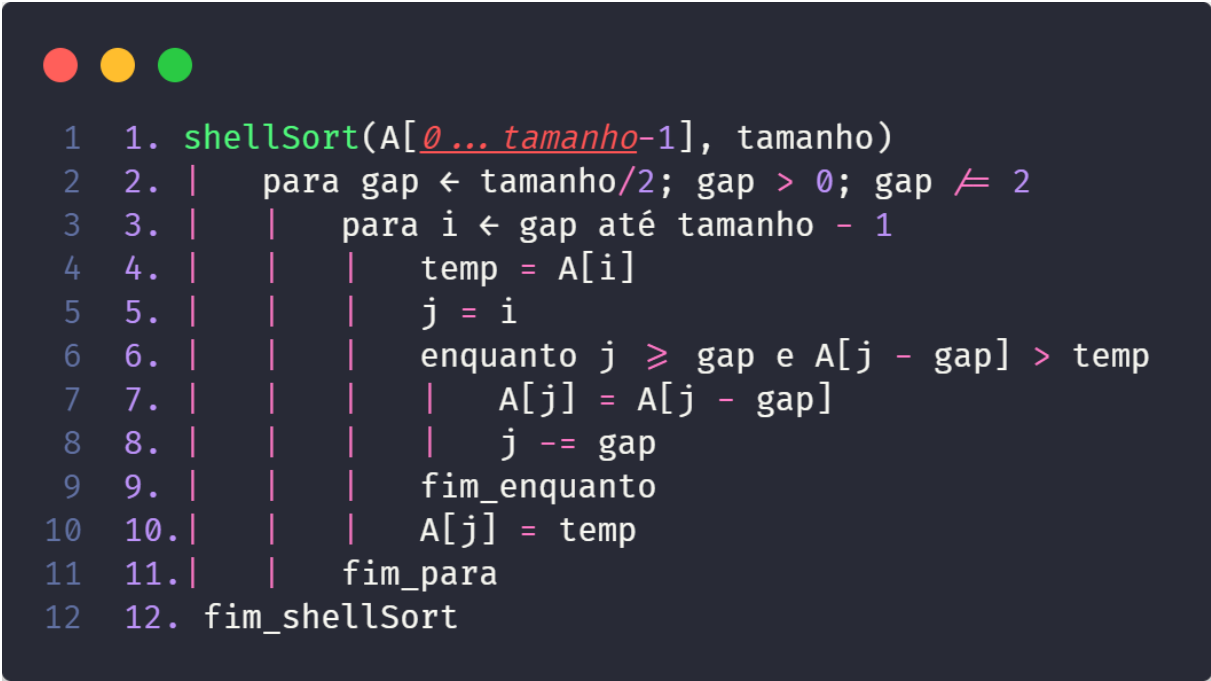
$$T(n) = (c_1 + c_2 + c_3 + c_4)n^2$$

Portanto, a complexidade do *Selection Sort* é quadrática ($O(n^2)$) em todos os casos, indicando que seu desempenho permanece consistente independentemente da configuração dos dados de entrada, mas essa consistência também implica limitações significativas ao lidar com conjuntos de dados extensos.

3.3 *Shell Sort*

O *Shell Sort*, é um algoritmo de ordenação que aprimora o processo de ordenação por inserção ao comparar elementos distantes, em vez de adjacentes. Sua singularidade está na subdivisão do *array* de entrada em grupos menores, que são ordenados individualmente usando diferentes sequências de incremento. O desempenho do algoritmo varia amplamente com base na sequência selecionada, resultando em diferentes cenários.

A Figura 18 apresenta um pseudocódigo do algoritmo *Shell Sort*.



```
1 1. shellSort(A[0... tamanho-1], tamanho)
2 2. |   para gap ← tamanho/2; gap > 0; gap ≠ 2
3 3. |   |   para i ← gap até tamanho - 1
4 4. |   |   |   temp = A[i]
5 5. |   |   |   j = i
6 6. |   |   |   enquanto j ≥ gap e A[j - gap] > temp
7 7. |   |   |   |   A[j] = A[j - gap]
8 8. |   |   |   |   j -= gap
9 9. |   |   |   fim_enquanto
10 10. |   |   A[j] = temp
11 11. |   fim_para
12 12. fim_shellSort
```

Figura 18: Pseudocódigo do algoritmo *Shell Sort*

3.3.1 Variação de Complexidade

A complexidade do *Shell Sort* varia significativamente entre diferentes contextos. No melhor caso, uma sequência de incremento ideal pode resultar em um desempenho notavelmente rápido, enquanto no pior caso, outra sequência pode levá-lo a um desempenho semelhante a algoritmos de ordenação quadráticos. A análise do caso médio também é desafiadora devido à variabilidade na sequência de incremento.

É crucial entender que a complexidade do *Shell Sort* é altamente dependente da sequência de incremento e das circunstâncias de implementação específicas. Portanto, fornecer uma

complexidade exata sem informações detalhadas sobre a sequência de incremento é impossível.

A complexidade do *Shell Sort* não é facilmente determinada devido à sua dependência na escolha da sequência de incremento, tornando-se objeto de pesquisa ativa.

3.3.2 Melhor Caso

No melhor caso, quando os elementos já estão ordenados, o algoritmo possui uma complexidade igual a $O(n)$.

3.3.3 Pior Caso

De acordo com [Frank and Lazarus(1960)], a complexidade no pior caso analítico é quadrática $O(n^2)$.

Já [Pratt(1972)], obteve uma complexidade de $O(n \log^2 n)$.

3.3.4 Caso Médio - Analítico

Segundo [Pratt(1972)] o caso médio analítico tem complexidade igual a $O(n \log^2 n)$.

3.3.5 Caso Médio - Empírico

[Shell(1959)], o criador do algoritmo, encontrou uma complexidade de $O(n^{1.226})$.

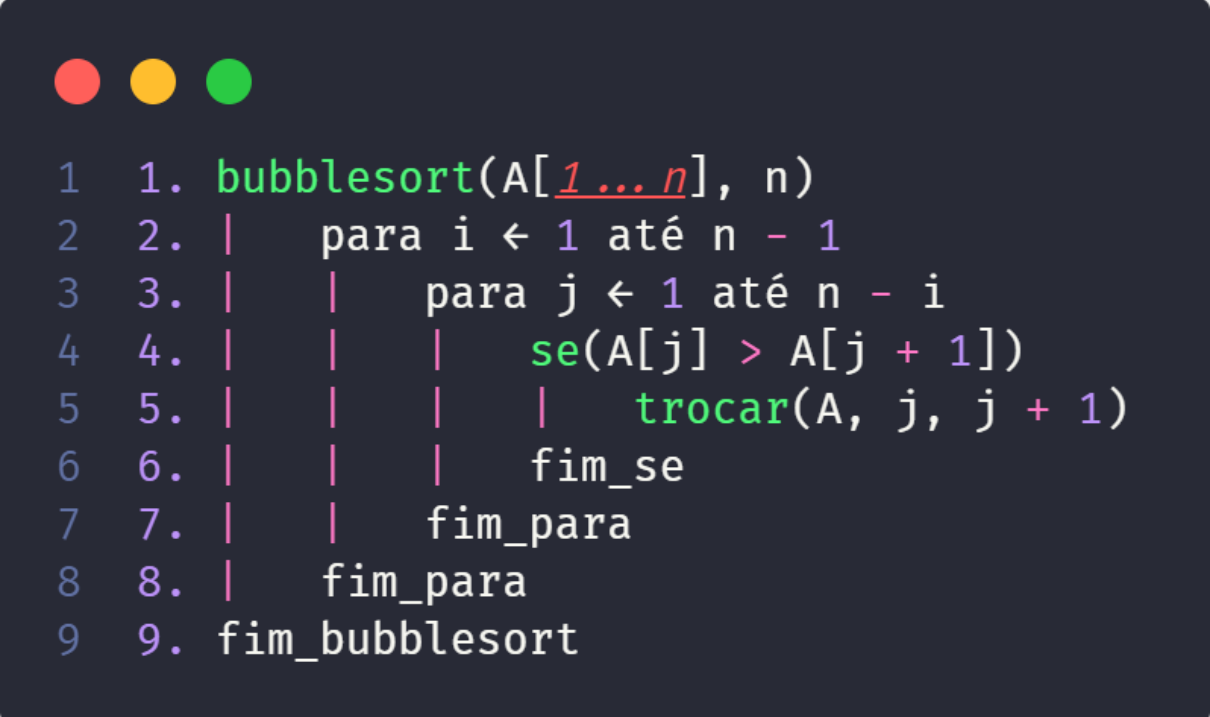
Através de sua pesquisa, [Knuth(1973)] encontrou uma complexidade de $O(n^{1.26})$ ou $O(n \log^2 n)$.

[Weiss(1991)] identificou duas complexidades diferentes. A primeira, tendo [Knuth(1973)] como base para seus estudos, foi de $O(n^{1.25})$ ou $O(n \log^2 n)$, posteriormente encontrou $O(n^{7/6})$, quando usou [Sedgewick(1986)] como base de pesquisa.

3.4 *Bubble Sort*

O algoritmo de ordenação *Bubble Sort* é simples, mas seu desempenho é limitado pela sua complexidade quadrática, independentemente da ordenação inicial dos elementos no *array* de entrada.

A Figura 19 apresenta um pseudocódigo do algoritmo *Bubble Sort*.



```
1  1. bubblesort(A[1...n], n)
2  2. |   para i ← 1 até n - 1
3  3. |   |   para j ← 1 até n - i
4  4. |   |   |   se(A[j] > A[j + 1])
5  5. |   |   |   |   trocar(A, j, j + 1)
6  6. |   |   |   fim_se
7  7. |   |   fim_para
8  8. |   fim_para
9  9. fim_bubblesort
```

Figura 19: Pseudocódigo do algoritmo *Bubble Sort*

3.4.1 Melhor Caso - Ordem Crescente

No melhor caso, quando o *array* de entrada já está completamente ordenado em ordem crescente, o *Bubble Sort* faz n iterações, mas nenhuma troca é necessária, pois os elementos já estão na ordem correta. Isso significa que o tempo de execução no melhor caso é:

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5)n$$

Neste caso, o algoritmo *Bubble Sort* se comporta de maneira linear, mas é importante notar que ele ainda é limitado pelo seu pior caso e caso médio.

3.4.2 Pior Caso - Ordem Decrescente

O pior caso surge quando o conjunto de dados está organizado de forma decrescente. Nesse caso, o algoritmo executará uma troca durante todas as iterações. O número de iterações no loop é igual a $n-1$, que é representado por meio da seguinte soma:

$$\sum_{i=1}^{n-1} (\dots)$$

O loop interno realiza $n-i$ iterações. Dessa forma, temos a seguinte expressão:

$$\sum_{i=1}^{n-1} \left(\sum_{j=1}^{n-i} 1 \right)$$

Essa fórmula nos fornece o total de vezes em que a comparação $A[j] > A[j+1]$ é efetuada. Inicialmente, resolvemos a soma entre parênteses:

$$\sum_{i=1}^{n-1} (n-1)$$

Em seguida, calculamos a soma restante:

$$\sum_{i=1}^{n-1} (n-1) = n(n-1) - \frac{n(n-1)}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$$

Portanto, no pior caso, a complexidade do *Bubble Sort* é classificada como $O(n^2)$. Quando o *array* de entrada está completamente ordenado em ordem decrescente, o *Bubble Sort* faz n iterações no pior caso. Em cada iteração, é necessário comparar e trocar todos os elementos até que o maior elemento "suba" para a posição correta. Isso resulta em um tempo de execução no pior caso de:

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5)n^2$$

O tempo de execução no pior caso é quadrático em relação ao tamanho da entrada n , o que torna o *Bubble Sort* ineficiente para conjuntos de dados grandes.

3.4.3 Caso Médio - Ordenação Aleatória

No caso médio, é necessário calcular a média dos custos entre todas as situações possíveis. O caso de menor custo ocorre quando o loop mais externo realiza apenas uma iteração, enquanto o caso de maior custo acontece quando esse mesmo loop externo atinge o seu número máximo de iterações, ou seja, $n - 1$ iterações. Uma vez que todas as configurações possíveis do vetor de entrada têm a mesma probabilidade de ocorrência, podemos afirmar que a probabilidade de cada caso é de $\frac{1}{n-1}$, e a média é calculada da seguinte forma:

$$\frac{1}{n-1} \sum_{k=1}^{n-1} (\dots)$$

O valor dentro dos parênteses representa o custo total do algoritmo quando o loop externo executa k iterações:

$$\frac{1}{n-1} \sum_{k=1}^{n-1} \left(\sum_{i=1}^k \sum_{j=1}^{n-i} 1 \right)$$

Resolvendo a expressão, obtemos:

$$\frac{1}{n-1} \sum_{k=1}^{n-1} \left(nk - \frac{k(k+1)}{2} \right) = \frac{n^2}{3} - \frac{n}{6} = O(n^2)$$

Portanto, no cenário médio, a complexidade do algoritmo é classificada como $O(n^2)$. No caso médio, quando os elementos do *array* estão em uma ordem aleatória, o *Bubble Sort* também demonstra um desempenho quadrático semelhante ao pior caso. O número médio de comparações e trocas é alto, resultando em um tempo de execução médio de:

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5)n^2$$

Dessa forma, *Bubble Sort* possui complexidade quadrática no pior caso e no caso médio. Para grandes conjuntos de dados, outros algoritmos de ordenação mais eficientes são geralmente preferíveis.

3.5 Merge Sort

O algoritmo *Merge Sort* divide a lista em duas metades, ordena cada metade recursivamente e, em seguida, mescla as duas metades ordenadas. A complexidade de tempo é analisada em três fases principais: dividir, conquistar e mesclar.

A Figura 20 apresenta um pseudocódigo do algoritmo *Merge Sort*.



```
1 01. mergesort(A[0...n - 1], inicio, fim)
2 02. |   se (inicio < fim)
3 03. |   |   meio ← (inicio + fim) / 2
4 04. |   |   mergesort(A, inicio, meio)
5 05. |   |   mergesort(A, meio + 1, fim)
6 06. |   |   merge(A, inicio, meio, fim)
7 07. |   fim_se
8 08. fim_mergesort
```

Figura 20: Pseudocódigo do algoritmo *Merge Sort*

3.5.1 Dividir

A operação de dividir a lista em duas metades leva $O(1)$ tempo, pois é uma operação constante.

3.5.2 Conquistar

A complexidade de tempo para ordenar cada metade é $T(n/2)$, onde n é o tamanho da lista original.

3.5.3 Mesclar

A complexidade de tempo para mesclar as duas metades ordenadas é $\Theta(n)$, onde n é o tamanho total da lista.

A equação de recorrência para o tempo de execução é, portanto, definida como:

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

Resolução pelo Teorema Mestre

Considerando o problema de recorrência:

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

Aplicando o Teorema Mestre para a resolução, com $a = 2$, $b = 2$, e $f(n) = \Theta(n)$, os seguintes passos foram identificados:

1. Em um primeiro passo, procede-se com a comparação utilizando $n^{\log_b a}$:

$$\log_2 2 = 1, \quad n^{\log_2 2} = n$$

Verifica-se que $f(n) = \Theta(n)$ é assintoticamente equivalente a $n^{\log_2 2}$, enquadrando-se, assim, no caso 2 do Teorema Mestre.

2. A solução obtida é expressa como:

$$T(n) = \Theta(n \log n)$$

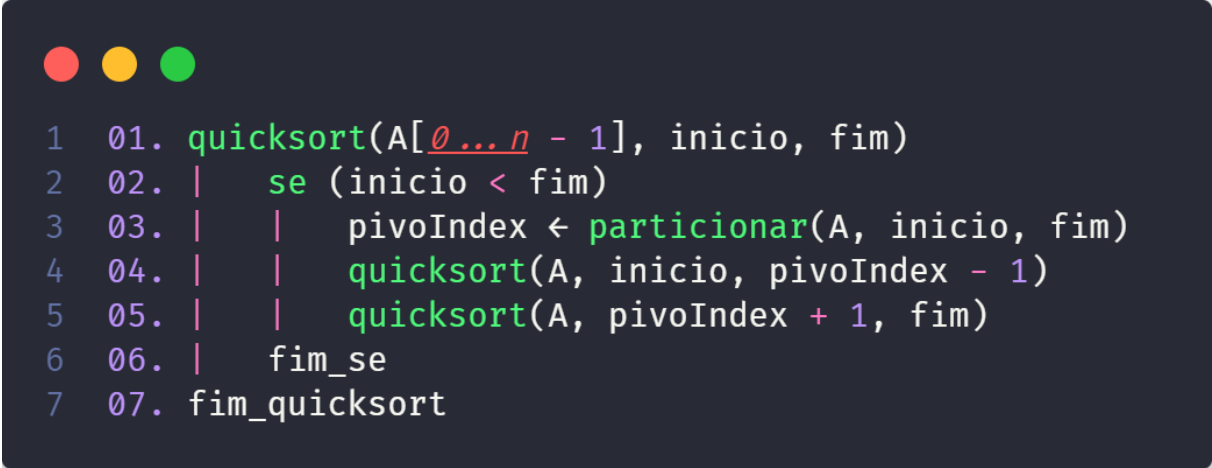
Portanto a complexidade do algoritmo *Merge Sort* para todos os casos (melhor, pior e médio) é

$$\Theta(n \log n)$$

3.6 Quick Sort

O desempenho do algoritmo *Quick Sort* varia em diferentes cenários, sendo influenciado pelo arranjo dos elementos no vetor de entrada.

A Figura 21 apresenta um pseudocódigo do algoritmo *Quick Sort*.



```
1 01. quicksort(A[0...n - 1], inicio, fim)
2 02. |   se (inicio < fim)
3 03. |   |   pivoIndex ← particionar(A, inicio, fim)
4 04. |   |   quicksort(A, inicio, pivoIndex - 1)
5 05. |   |   quicksort(A, pivoIndex + 1, fim)
6 06. |   fim_se
7 07. fim_quicksort
```

Figura 21: Pseudocódigo do algoritmo *Quick Sort*

3.6.1 Melhor Caso

Em sua melhor situação, quando as partições são balanceadas com cada uma contendo $n/2$ elementos, a equação de recorrência para o melhor caso é expressa como:

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

onde $\Theta(n)$ representa a complexidade de tempo do método de partição. A solução para esta recorrência é $T(n) = \Theta(n \log n)$. Esta solução é obtida usando o teorema mestre. Logo, a complexidade no melhor caso é

$$\Theta(n \log n)$$

3.6.2 Pior Caso

No entanto, o pior caso do *Quick Sort* ocorre quando as chamadas recursivas resultam em partições com 0 e $n - 1$ elementos, especialmente evidente quando o vetor está ordenado em

ordem crescente ou decrescente. De acordo com [Cormen(2012)] a equação de recorrência para o pior caso é dada por:

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

resultando em uma complexidade igual a $O(n^2)$.


3.6.3 Caso Médio

O caso médio é geralmente mais difícil de analisar com precisão, pois depende da escolha aleatória ou estratégica do pivô. No entanto, em média, o *Quick Sort* tem um desempenho muito bom. A complexidade de tempo médio é geralmente considerada $O(n \log n)$.

3.7 *Heap Sort*

O algoritmo *heapsort* é um método de ordenação que utiliza uma estrutura de dados chamada *heap*, onde cada nó é maior ou igual aos seus dois filhos em uma árvore binária completa.

A Figura 22 apresenta um pseudocódigo do algoritmo *Heap Sort*.



```
1  1. heapsort(A[0... tam-1], tam)
2  2. |   para i ← tam/2 - 1 até 0
3  3. |   |   heapify(A, tam, i)
4  4. |   fim_para
5  6. |   para i ← tam - 1 até 0
6  7. |   |   trocar(A, 0, i)
7  8. |   |   heapify(A, i, 0)
8  9. |   fim_para
9  10. fim_heapsort
```

Figura 22: Pseudocódigo do algoritmo *Heap Sort*

O procedimento do *heapsort* envolve duas etapas principais:

1. **Construção do *Heap*:** O vetor de entrada é convertido em um *heap*. Isso é realizado por meio de um algoritmo de construção de *heap*, cuja complexidade é $O(n)$.
2. **Ordenação:** O elemento máximo do *heap* é removido e colocado no final do vetor ordenado. Esse processo é repetido até que o vetor esteja completamente ordenado.

A remoção do elemento máximo de um *heap* possui complexidade $O(\log n)$ devido à estrutura da árvore binária completa.

Portanto, a complexidade total do algoritmo *heapsort* é:

$$O(n) + O(n \log n) + O(n \log n)$$

Essa expressão pode ser simplificada para:

$$O(n \log n)$$

Dessa forma, a complexidade assintótica do algoritmo *heapsort* é $O(n \log n)$.

3.7.1 Complexidade no Melhor Caso

A complexidade $O(n \log n)$ do algoritmo *heapsort* também é aplicável ao melhor caso. No melhor cenário, o *heap* assume a forma de uma lista ordenada de maneira crescente, demandando apenas uma remoção do elemento máximo do *heap*.

Portanto, a complexidade do algoritmo *heapsort* é $O(n \log n)$ no melhor caso.

3.7.2 Complexidade no Pior Caso

A complexidade $O(n \log n)$ do algoritmo *heapsort* é válida para todos os casos, inclusive o pior caso. No cenário de pior caso, o *heap* assume a forma de uma lista ordenada de maneira decrescente, exigindo a remoção do elemento máximo do *heap* n vezes.

Assim, a complexidade do algoritmo *heapsort* permanece como $O(n \log n)$ no pior caso.

3.7.3 Complexidade Média

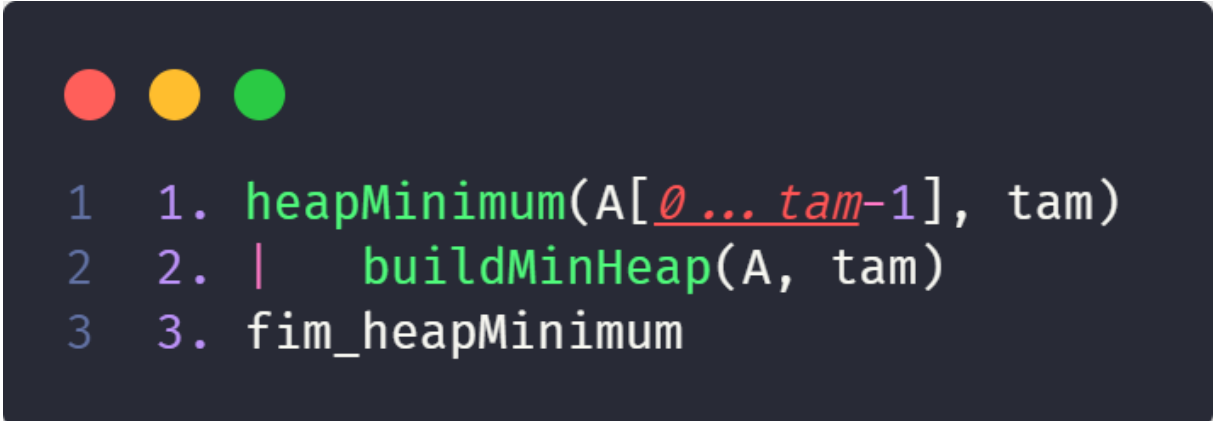
A complexidade $O(n \log n)$ do algoritmo *heapsort* também serve como uma boa aproximação para a complexidade média. Em média, o *heap* assume a forma de uma árvore binária completa com uma distribuição uniforme dos elementos, resultando em aproximadamente $n/2$ remoções do elemento máximo do *heap*.

Assim, a complexidade média do algoritmo *heapsort* é $O(n \log n)$.

3.8 *Heap Minimum*

O algoritmo *Heap Minimum* é uma variante específica do *heapsort*, focada na obtenção eficiente do elemento mínimo em um *heap*. Considerando a estrutura de dados do *heap*, que é uma árvore binária completa, a complexidade para a operação de obtenção do mínimo pode ser analisada.

A Figura 23 apresenta um pseudocódigo do algoritmo *Heap Minimum*.

The image shows a code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of pseudocode:

```
1  1. heapMinimum(A[0...tam-1], tam)
2  2. |   buildMinHeap(A, tam)
3  3. fim_heapMinimum
```

Figura 23: Pseudocódigo do algoritmo *Heap Minimum*

3.8.1 Melhor Caso:

No melhor caso, o *heap* está organizado de forma que o elemento mínimo é a raiz da árvore. A operação de obtenção do mínimo envolve acessar a raiz, que é uma operação constante. Portanto, a complexidade no melhor caso é $O(1)$.

3.8.2 Pior Caso:

No pior caso, a árvore do *heap* está organizada de forma decrescente, e a operação de obtenção do mínimo envolve percorrer a altura da árvore. A altura da árvore em um *heap* binário é $O(\log n)$, onde n é o número de elementos. Assim, a complexidade no pior caso é $O(\log n)$.

3.8.3 Complexidade Média:

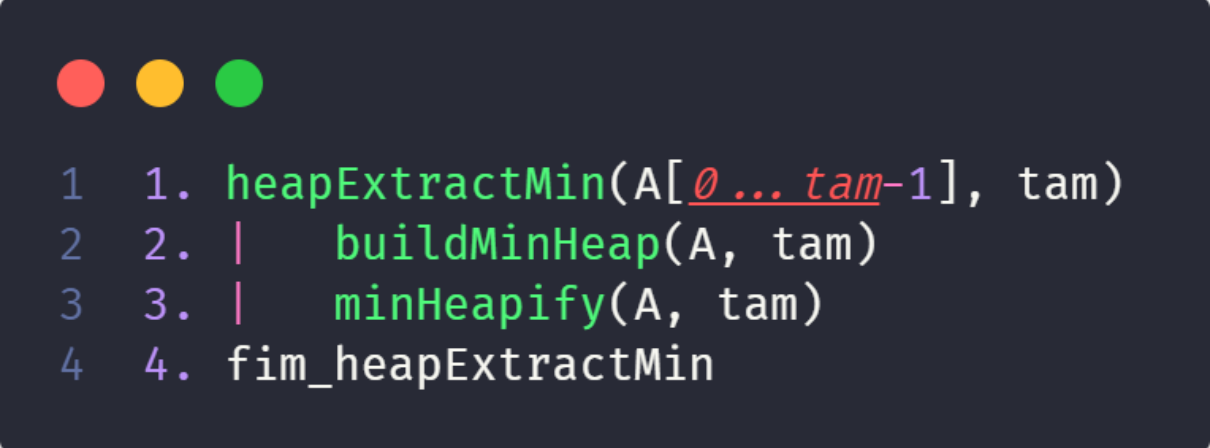
Em uma análise média, considerando um *heap* balanceado, a altura média é $O(\log n)$. Portanto, a complexidade média é $O(\log n)$.

O algoritmo *Heap Minimum*, sendo uma operação específica em um *heap*, apresenta uma complexidade ótima no melhor caso, com $O(1)$. No entanto, no pior caso e em média, a complexidade é $O(\log n)$, o que é eficiente para operações de obtenção do mínimo em comparação com estruturas de dados não otimizadas.

3.9 *Heap Extract Min*

O algoritmo *Heap Extract Min* é projetado para extrair eficientemente o elemento mínimo de um *heap* mínimo. Considerando a estrutura de dados do *heap* mínimo, que é uma árvore binária completa em que cada nó é menor ou igual aos seus dois filhos, a complexidade dessa operação pode ser analisada.

A Figura 24 apresenta um pseudocódigo do algoritmo *Heap Extract Min*.



```
1  1. heapExtractMin(A[0...tam-1], tam)
2  2. |   buildMinHeap(A, tam)
3  3. |   minHeapify(A, tam)
4  4. fim_heapExtractMin
```

Figura 24: Pseudocódigo do algoritmo *Heap Extract Min*

3.9.1 Melhor Caso:

No melhor caso, o *heap* está organizado de forma que o elemento mínimo é a raiz da árvore. A operação de extração do mínimo envolve a remoção direta da raiz, seguida por uma operação para manter as propriedades do *heap*. A complexidade no melhor caso é $O(\log n)$, onde n é o número de elementos no *heap*, devido à altura da árvore binária completa.

3.9.2 Pior Caso:

No pior caso, a árvore do *heap* está organizada de forma que o elemento mínimo está nas folhas mais distantes da raiz. A operação de extração do mínimo envolve a troca da raiz com uma folha e, em seguida, a descida dessa folha para a posição correta para manter as propriedades do *heap*. A complexidade no pior caso é $O(\log n)$, devido à altura da árvore binária completa.

3.9.3 Complexidade Média:

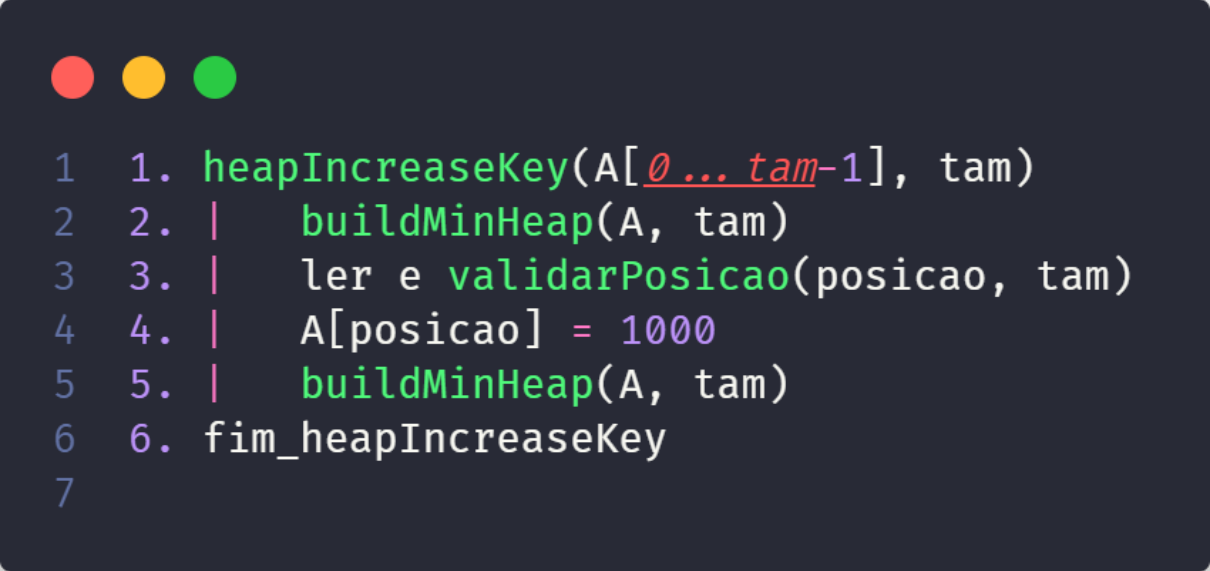
Em uma análise média, considerando um *heap* balanceado, a altura média é $O(\log n)$. Portanto, a complexidade média é $O(\log n)$.

O algoritmo *Heap Extract Min*, sendo uma operação específica em um *heap* mínimo, apresenta uma complexidade ótima no melhor caso, no pior caso e em média, com $O(\log n)$. Essa eficiência torna o *Heap Extract Min* uma escolha eficaz para operações de extração do mínimo em comparação com estruturas de dados não otimizadas.

3.10 *Heap Increase Key*

O algoritmo *Heap Increase Key* é utilizado para aumentar o valor de uma chave em um *heap* máximo e, em seguida, restaurar as propriedades do *heap*. Considerando a estrutura de dados do *heap* máximo, que é uma árvore binária completa em que cada nó é maior ou igual aos seus dois filhos, a complexidade dessa operação pode ser analisada.

A Figura 25 apresenta um pseudocódigo do algoritmo *Heap Increase Key*.



```
1 1. heapIncreaseKey(A[0... tam-1], tam)
2 2. |   buildMinHeap(A, tam)
3 3. |   ler e validarPosicao(posicao, tam)
4 4. |   A[posicao] = 1000
5 5. |   buildMinHeap(A, tam)
6 6. fim_heapIncreaseKey
7
```

Figura 25: Pseudocódigo do algoritmo *Heap Increase Key*

3.10.1 Melhor Caso:

No melhor caso, a chave aumentada não ultrapassa o valor de seus pais na árvore do *heap*. A operação de aumentar a chave e restaurar as propriedades do *heap* envolve uma comparação com o pai e, se necessário, a troca para cima na árvore. A complexidade no melhor caso é $O(\log n)$, onde n é o número de elementos no *heap*, devido à altura da árvore binária completa.

3.10.2 Pior Caso:

No pior caso, a chave aumentada é maior do que os valores de seus pais na árvore, resultando em uma troca até a raiz. A operação envolve uma série de comparações e trocas até que as

propriedades do *heap* sejam restauradas. A complexidade no pior caso é $O(\log n)$, devido à altura da árvore binária completa.

3.10.3 Complexidade Média:


Em uma análise média, considerando um *heap* balanceado, a altura média é $O(\log n)$. Portanto, a complexidade média é $O(\log n)$.

O algoritmo *Heap Increase Key*, sendo uma operação específica em um *heap* máximo, apresenta uma complexidade ótima no melhor caso, no pior caso e em média, com $O(\log n)$. Essa eficiência torna o *Heap Increase Key* uma escolha eficaz para operações de aumento de chave em comparação com estruturas de dados não otimizadas.

3.11 *Max Heap Insert*

O algoritmo *Max Heap Insert* é utilizado para inserir um novo elemento em um *heap* máximo e, em seguida, restaurar as propriedades do *heap*. Considerando a estrutura de dados do *heap* máximo, que é uma árvore binária completa em que cada nó é maior ou igual aos seus dois filhos, a complexidade dessa operação pode ser analisada.

A Figura 26 apresenta um pseudocódigo do algoritmo *Max Heap Insert*.



```
1  1. maxHeapInsert(A[0...tam], tam)
2  2. |    novoValor = 1000
3  3. |    tam++
4  4. |    A[tam - 1] = novoValor
5  5. |    buildMaxHeap(A, tam)
6  6. |    i = tam - 1
7  7. |    enquanto i > 0
8  8. |    |    pai = (i - 1) / 2
9  9. |    |    se A[i] > A[pai]
10 10. |    |    |    trocar(A, i, pai)
11 11. |    |    |    i = pai
12 12. |    |    senao
13 13. |    |    |    quebrar o loop
14 14. |    |    fim_se
15 15. |    fim_enquanto
16 16. fim_maxHeapInsert
```

Figura 26: Pseudocódigo do algoritmo *Max Heap Insert*

3.11.1 Melhor Caso:

No melhor caso, a inserção ocorre diretamente na folha mais à esquerda, sem a necessidade de reorganizar a estrutura do *heap*. A operação envolve a adição do novo elemento como uma folha, sem a necessidade de trocas. A complexidade no melhor caso é $O(1)$, pois a inserção é realizada em tempo constante.

3.11.2 Pior Caso:

No pior caso, a inserção ocorre na última posição disponível, exigindo uma comparação e, potencialmente, várias trocas até que as propriedades do *heap* sejam restauradas. A complexidade no pior caso é $O(\log n)$, onde n é o número de elementos no *heap*, devido à altura da árvore binária completa.

3.11.3 Complexidade Média:

Em uma análise média, considerando um *heap* balanceado, a altura média é $O(\log n)$. Portanto, a complexidade média é $O(\log n)$.

O algoritmo *Max Heap Insert*, sendo uma operação específica em um *heap* máximo, apresenta uma complexidade ótima no melhor caso, no pior caso e em média. No melhor caso, a complexidade é $O(1)$, enquanto no pior caso e em média é $O(\log n)$. Essa eficiência torna o *Max Heap Insert* uma escolha eficaz para operações de inserção em comparação com estruturas de dados não otimizadas.

4 TABELA E GRÁFICO

Os experimentos foram conduzidos em um computador com processador AMD Ryzen 5 5600G @3.90GHz de 6 núcleos - 12 threads e 16GB de memória RAM, e foram obtidos os seguintes resultados:

4.1 *Insertion Sort*

A Figura 27 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Insertion Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,00	0,02	0,14
	Decrescente	0,00	0,00	0,00	0,07	7,33	744,58
	Aleatório	0,00	0,00	0,00	0,04	3,66	369,50

Figura 27: Tabela de tempo por segundo do algoritmo *Insertion Sort*

A Figura 28 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Insertion Sort*.

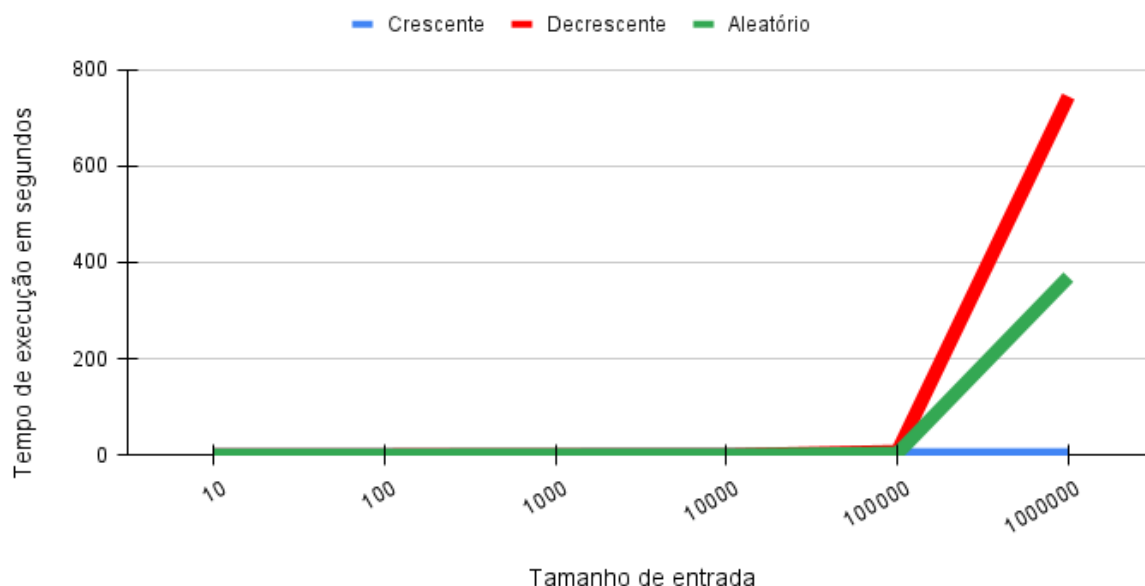


Figura 28: Gráfico de tempo por segundo do algoritmo *Insertion Sort*

Para listas que já estão ordenadas em ordem crescente, o *Insertion Sort* é altamente eficiente. Em todos os tamanhos de entrada, o tempo de execução é muito próximo de zero, com valores na ordem de 0 segundos. Isso ocorre porque o algoritmo só precisa fazer algumas comparações para determinar que os elementos já estão na ordem correta, resultando em um desempenho excepcionalmente rápido.

O desempenho do *Insertion Sort* em listas ordenadas em ordem decrescente é ligeiramente menos eficiente, mas ainda é razoável para tamanhos de entrada pequenos (10, 100). No entanto, à medida que o tamanho da lista aumenta, os tempos de execução aumentam significativamente. Com 1.000.000 de elementos, o tempo de execução é de 744,58 segundos (aproximadamente 12,41 minutos), o que é bastante elevado. Isso ocorre porque o *Insertion Sort* precisa deslocar muitos elementos para inserir cada novo elemento na posição correta.

Para listas com valores dispostos aleatoriamente, o *Insertion Sort* também apresenta desempenho razoável para tamanhos pequenos de entrada (10, 100). No entanto, à medida que o tamanho da lista aumenta, os tempos de execução aumentam consideravelmente. Com 1.000.000 de elementos, o tempo de execução é de 369,50 segundos (aproximadamente 6,16 minutos), o que, embora mais eficiente do que no caso decrescente, ainda é bastante alto. O

Insertion Sort faz várias comparações e deslocamentos de elementos, o que o torna menos eficiente para grandes conjuntos de dados aleatórios.

Em resumo, o *Insertion Sort* é um algoritmo de ordenação eficiente para listas que já estão quase ordenadas (crescente) ou para tamanhos de entrada pequenos. No entanto, ele se torna ineficiente para listas ordenadas em ordem decrescente ou para tamanhos de entrada muito grandes, onde algoritmos de ordenação mais eficientes, como *Merge Sort* ou *Quick Sort*, podem ser mais adequados.

4.2 Selection Sort

A Figura 29 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Selection Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,05	4,61	463,66
	Decrescente	0,00	0,00	0,00	0,06	5,28	521,87
	Aleatório	0,00	0,00	0,00	0,06	4,62	455,80

Figura 29: Tabela de tempo por segundo do algoritmo *Selection Sort*

A Figura 30 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Selection Sort*.



Figura 30: Gráfico de tempo por segundo do algoritmo *Selection Sort*

Quando aplicado a uma lista que já está ordenada em ordem crescente, o *Selection Sort* tem seu melhor desempenho. Isso ocorre porque, em cada passo, ele seleciona o elemento

mínimo da parte não classificada da lista, que já é o primeiro elemento na ordenação crescente. Portanto, o tempo de execução é mínimo, próximo de zero, independentemente do tamanho da entrada, medido em segundos. Isso faz do *Selection Sort* uma escolha eficaz quando a lista já está quase ordenada.

O desempenho do *Selection Sort* em listas ordenadas em ordem decrescente é semelhante ao cenário crescente, mas requer um tempo ligeiramente maior. Isso ocorre porque o *Selection Sort* ainda precisa percorrer a lista para encontrar o elemento mínimo em cada passo. No entanto, os tempos de execução são muito baixos para tamanhos de entrada pequenos (10, 1000), mas aumentam de maneira linear à medida que o tamanho da entrada cresce. Com 1.000.000 de elementos, o tempo de execução é de 521,87 segundos (aproximadamente 8,7 minutos), o que é bastante razoável.

Para listas com valores dispostos aleatoriamente, o *Selection Sort* apresenta um desempenho aceitável para tamanhos pequenos de entrada (10, 100). No entanto, à medida que o tamanho da lista aumenta, o tempo de execução também aumenta, atingindo 455,80 segundos (aproximadamente 7,6 minutos) com 1.000.000 de elementos. Embora o *Selection Sort* seja mais eficiente do que outros algoritmos de ordenação, como o *Bubble Sort*, para entradas aleatórias, ele ainda não é a escolha ideal para listas muito grandes, onde algoritmos de ordenação mais eficientes, como o *Merge Sort* ou *Quick Sort*, podem ser mais apropriados.

Em resumo, o *Selection Sort* é um algoritmo de ordenação simples e eficaz, especialmente em cenários onde a lista já está quase ordenada. No entanto, seu desempenho começa a deteriorar-se em listas maiores com valores aleatórios, tornando-o menos adequado para conjuntos de dados muito extensos.

4.3 *Shell Sort*

A Figura 31 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Shell Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,00	0,02	0,18
	Decrescente	0,00	0,00	0,00	0,00	0,02	0,20
	Aleatório	0,00	0,00	0,00	0,00	0,03	0,33

Figura 31: Tabela de tempo por segundo do algoritmo *Shell Sort*

A Figura 32 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Shell Sort*.

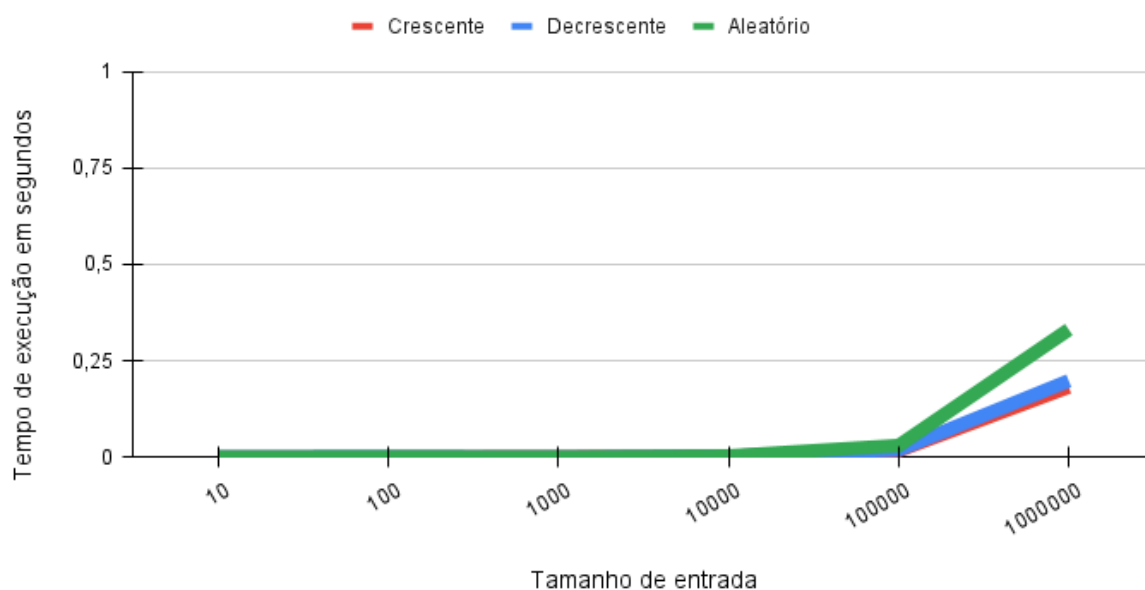


Figura 32: Gráfico de tempo por segundo do algoritmo *Shell Sort*

Para listas que já estão ordenadas em ordem crescente, o *Shell Sort* demonstra um desempenho excepcionalmente rápido. Em todos os tamanhos de entrada, o tempo de execução

é próximo de zero, com apenas algumas frações de segundos (0,00). Isso ocorre porque, em cenários ordenados, o algoritmo precisa fazer apenas um número limitado de comparações e trocas, tornando-o extremamente eficiente.

Assim como no cenário crescente, o *Shell Sort* também funciona bem em listas ordenadas em ordem decrescente. Os tempos de execução permanecem muito baixos, com valores em torno de 0,00 segundos para todos os tamanhos de entrada. Isso ressalta a capacidade do *Shell Sort* de otimizar o desempenho em cenários onde a lista já possui algum grau de ordenação.

Para listas com valores dispostos aleatoriamente, o *Shell Sort* continua a apresentar um bom desempenho. Embora os tempos de execução sejam maiores do que nos cenários ordenados, eles ainda são bastante razoáveis. Com 1.000.000 de elementos, o tempo de execução é de 0,33 segundos. Isso indica que o *Shell Sort* é eficaz em lidar com entradas aleatórias, tornando-o uma escolha sólida para uma variedade de situações.

Em resumo, o algoritmo *Shell Sort* se destaca em cenários de ordenação crescente e decrescente, onde oferece desempenho excepcionalmente rápido. Mesmo em listas com valores aleatórios, o *Shell Sort* mantém um desempenho bastante razoável. Isso o torna uma escolha versátil e eficaz para ordenação, especialmente quando não se conhece a disposição inicial dos dados.

4.4 *Bubble Sort*

A Figura 33 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Bubble Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,06	5,15	520,89
	Decrescente	0,00	0,00	0,00	0,15	14,54	1.461,46
	Aleatório	0,00	0,00	0,00	0,10	17,57	1.843,02

Figura 33: Tabela de tempo por segundo do algoritmo *Bubble Sort*

A Figura 34 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Bubble Sort*.

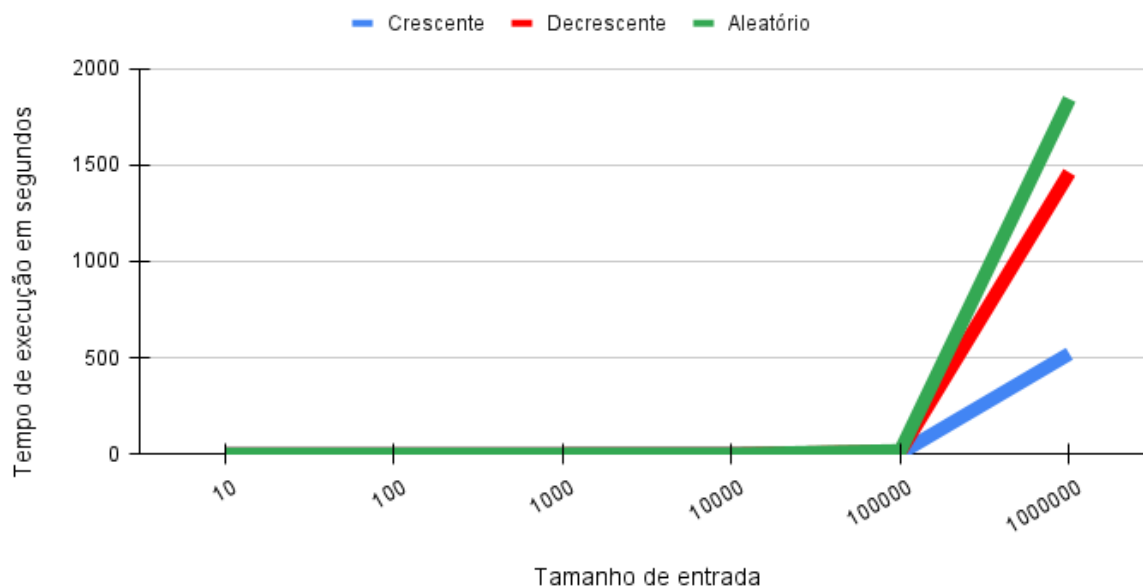


Figura 34: Gráfico de tempo por segundo do algoritmo *Bubble Sort*

Quando a lista já está ordenada em ordem crescente, o *Bubble Sort* tem seu melhor desempenho. Isso ocorre porque não são necessárias trocas, e o algoritmo pode detectar que

nenhum elemento foi trocado após uma passagem completa, encerrando antecipadamente. Para conjuntos de dados pequenos, como 10 ou 100 elementos, o tempo de execução é insignificante. No entanto, à medida que o tamanho da entrada aumenta, o *Bubble Sort* se torna cada vez mais ineficiente. Isso ocorre porque, mesmo que a lista esteja quase completamente ordenada, ele ainda precisará percorrer toda a lista várias vezes, fazendo comparações desnecessárias. Com 1.000.000 de elementos, o tempo de execução chega a mais de 8 minutos, o que o torna impraticável para grandes conjuntos de dados.

O desempenho do *Bubble Sort* em listas ordenadas em ordem decrescente é um pouco melhor do que em listas aleatórias, porque ele pode detectar que nenhum elemento foi trocado após uma passagem completa e encerrar antecipadamente. No entanto, assim como no caso crescente, o tempo de execução aumenta significativamente com o aumento do tamanho da entrada. Com 1.000.000 de elementos, o tempo de execução é de mais de 24 minutos, o que ainda é muito alto.

Para entradas em ordem aleatória, o *Bubble Sort* é o menos eficiente dos três casos. Isso ocorre porque, mesmo em listas com alguma ordenação parcial, o algoritmo continua a fazer muitas comparações e trocas, tornando-o ineficaz para grandes conjuntos de dados. Com 100.000 elementos, o tempo de execução já está na faixa dos segundos, e com 1.000.000 de elementos, leva mais de 30 minutos.

Em resumo, o *Bubble Sort* é um algoritmo simples de entender e implementar, mas é altamente ineficiente para grandes conjuntos de dados. Se a eficiência for importante, especialmente para tamanhos maiores de entrada, é recomendável considerar algoritmos de ordenação mais eficientes, como *Merge Sort*, *Quick Sort* ou algoritmos baseados em estruturas de dados avançadas, dependendo dos requisitos específicos do problema.

4.5 Merge Sort

A Figura 35 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Merge Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,00	0,03	0,36
	Decrescente	0,00	0,00	0,00	0,00	0,04	0,35
	Aleatório	0,00	0,00	0,00	0,00	0,04	0,39

Figura 35: Tabela de tempo por segundo do algoritmo *Merge Sort*

A Figura 36 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Merge Sort*.

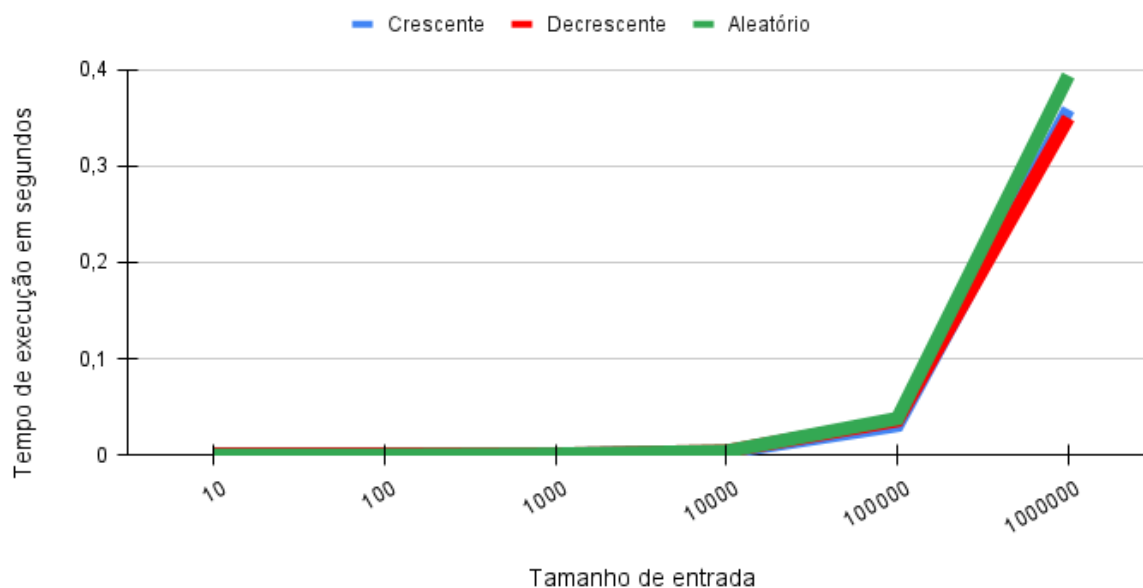


Figura 36: Gráfico de tempo por segundo do algoritmo *Merge Sort*

Para conjuntos de dados menores, variando de 10 a 1000 elementos, o tempo de execução do *Merge Sort* é praticamente insignificante, registrando valores de 0,00 segundos em todos

os cenários. Isso evidencia a eficiência do algoritmo ao lidar com conjuntos de dados de pequeno porte, independentemente da configuração inicial da lista.

À medida que o tamanho da entrada aumenta, começamos a observar um aumento gradual nos tempos de execução. Mesmo para conjuntos de dados com 10.000 elementos, o *Merge Sort* demonstra um desempenho notável, com tempos de execução de 0,00 segundos em todos os cenários.

No entanto, ao enfrentar conjuntos de dados mais robustos, com 100.000 e 1.000.000 de elementos, percebemos um aumento no tempo de execução. Ainda assim, os valores permanecem relativamente baixos, indicando que o *Merge Sort* é um algoritmo eficaz para lidar com grandes conjuntos de dados.

Ao compararmos os diferentes cenários (crescente, decrescente e aleatório), é notável que o *Merge Sort* mantém um desempenho consistente em todos os casos. Isso sugere que o algoritmo não é sensível à ordem inicial dos elementos e é capaz de realizar a ordenação de maneira eficiente, independentemente da configuração inicial da lista.

Em resumo, o *Merge Sort* é um algoritmo de ordenação altamente eficiente, mesmo para conjuntos de dados de grande porte. Sua capacidade de manter um desempenho consistente em diferentes cenários o torna uma escolha robusta para a ordenação de grandes volumes de dados em diversas situações.

4.6 Quick Sort - Versão Um

A Figura 37 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Um.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,15	4,28	430,20
	Decrescente	0,00	0,00	0,00	0,08	7,39	718,80
	Aleatório	0,00	0,00	0,00	0,03	0,03	0,27

Figura 37: Tabela de tempo por segundo do algoritmo *Quick Sort* - Versão Um

A Figura 38 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Um.

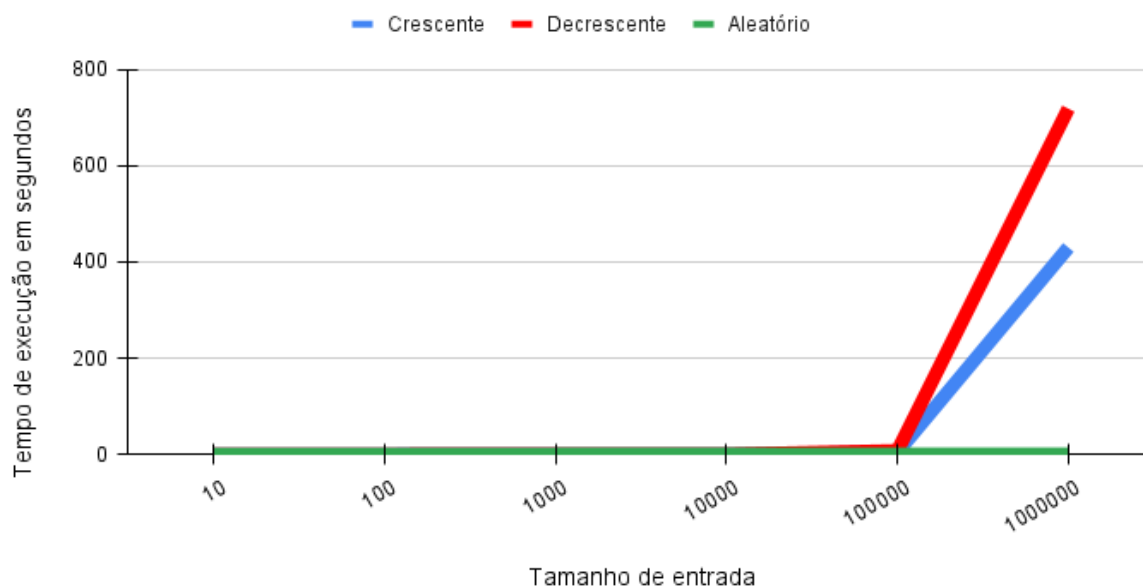


Figura 38: Gráfico de tempo por segundo do algoritmo *Quick Sort* - Versão Um

Quando se lida com conjuntos pequenos, variando de 10 a 1000 elementos, o *Quick Sort* (Versão 1) demonstra ser extremamente eficiente, levando apenas 0,00 segundos em todos

os casos. Isso enfatiza o quão rápido o algoritmo é ao lidar com conjuntos pequenos, não importando como a lista é inicialmente organizada.

À medida que o tamanho da entrada aumenta, começa-se a observar um aumento gradual nos tempos de execução. Para conjuntos de dados com 10.000 elementos, o *Quick Sort* (Versão 1) mostra um desempenho ligeiramente superior, com tempos de execução de 0,15 segundos para listas ordenadas em ordem crescente, 0,08 segundos para listas ordenadas em ordem decrescente e 0,03 segundos para listas aleatórias.

No entanto, ao lidar com conjuntos de dados mais substanciais, como 100.000 e 1.000.000 de elementos, observa-se um aumento significativo no tempo de execução. Nesses casos, o *Quick Sort* (Versão 1) leva 4,28 segundos para listas ordenadas em ordem crescente, 7,39 segundos para listas ordenadas em ordem decrescente e apenas 0,03 segundos para listas aleatórias com 100.000 elementos. Para 1.000.000 de elementos, o tempo de execução aumenta consideravelmente, atingindo 430,20 segundos para listas ordenadas em ordem crescente, 718,80 segundos para listas ordenadas em ordem decrescente e 0,27 segundos para listas aleatórias.

Ao analisar os diferentes cenários (crescente, decrescente e aleatório), observa-se que o desempenho do *Quick Sort* (Versão 1) é influenciado pela ordem inicial dos elementos. Eles demonstram um desempenho relativamente melhor em listas aleatórias em comparação com listas ordenadas em ordem crescente ou decrescente, especialmente para conjuntos de dados maiores.

Em resumo, o *Quick Sort* (Versão 1) é um algoritmo de ordenação eficaz para conjuntos de dados de pequeno e médio porte. São capazes de lidar com conjuntos de dados de pequeno porte de forma extremamente eficiente, independentemente da configuração inicial da lista. No entanto, para conjuntos de dados muito grandes, especialmente em listas ordenadas em ordem crescente ou decrescente, os tempos de execução aumentam consideravelmente.

4.7 Quick Sort - Versão Dois

A Figura 39 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Dois.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,00	0,02	0,24
	Decrescente	0,00	0,00	0,00	0,00	0,02	0,20
	Aleatório	0,00	0,00	0,01	0,00	0,02	0,24

Figura 39: Tabela de tempo por segundo do algoritmo *Quick Sort* - Versão Dois

A Figura 40 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Dois.

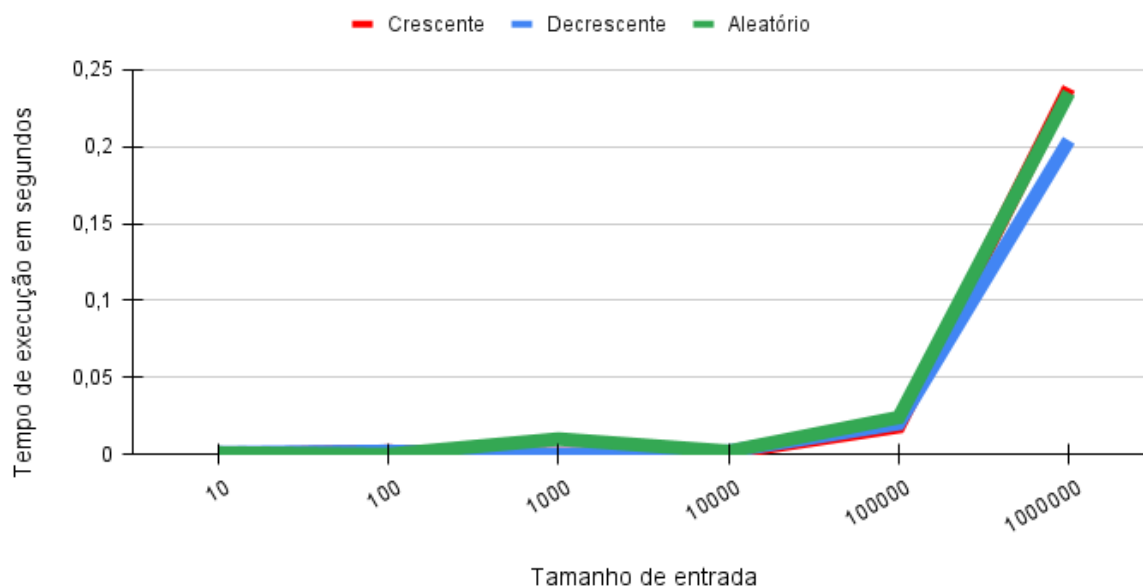


Figura 40: Gráfico de tempo por segundo do algoritmo *Quick Sort* - Versão Dois

A análise dos resultados permite observar o desempenho do algoritmo em relação ao tamanho da entrada e ao tipo de ordenação.

Para entradas pequenas, com tamanhos de 10 e 100 elementos, o tempo de execução se mostra praticamente insignificante, registrando valores de 0,00 segundos em todos os cenários, independentemente da ordem dos elementos. Esses resultados indicam que o *Quick Sort* (versão 2) demonstra alta eficiência para listas de tamanho reduzido.

À medida que o tamanho da entrada aumenta para 1000 elementos, ainda são observados tempos de execução muito baixos, na ordem de milissegundos, o que sugere que o algoritmo mantém um bom desempenho para listas de tamanho moderado.

No entanto, ao lidar com entradas maiores, como 10000, 100000 e 1000000 elementos, começa-se a notar um gradual aumento no tempo de execução. É notável que, de modo geral, a ordenação de listas aleatórias tende a demandar mais tempo em comparação com listas ordenadas de forma crescente ou decrescente. Isso se deve à natureza do *Quick Sort* (versão 2), que depende das escolhas de pivôs para a eficiência de seu funcionamento. Em listas ordenadas, o desempenho é otimizado, enquanto em listas aleatórias, o algoritmo pode ter que realizar um maior número de comparações e trocas.

Em resumo, os resultados refletem o comportamento esperado do algoritmo *Quick Sort* (versão 2). Ele demonstra extrema eficiência em listas de tamanho pequeno a moderado, mas pode apresentar um desempenho ligeiramente mais lento em listas maiores e aleatórias. É crucial considerar o contexto específico da aplicação ao escolher o algoritmo de ordenação mais adequado.

4.8 Quick Sort - Versão Três

A Figura 41 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Três.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,04	3,03	306,90
	Decrescente	0,00	0,00	0,00	0,03	2,07	204,03
	Aleatório	0,00	0,00	0,00	0,00	0,03	0,27

Figura 41: Tabela de tempo por segundo do algoritmo *Quick Sort* - Versão Três

A Figura 42 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Três.

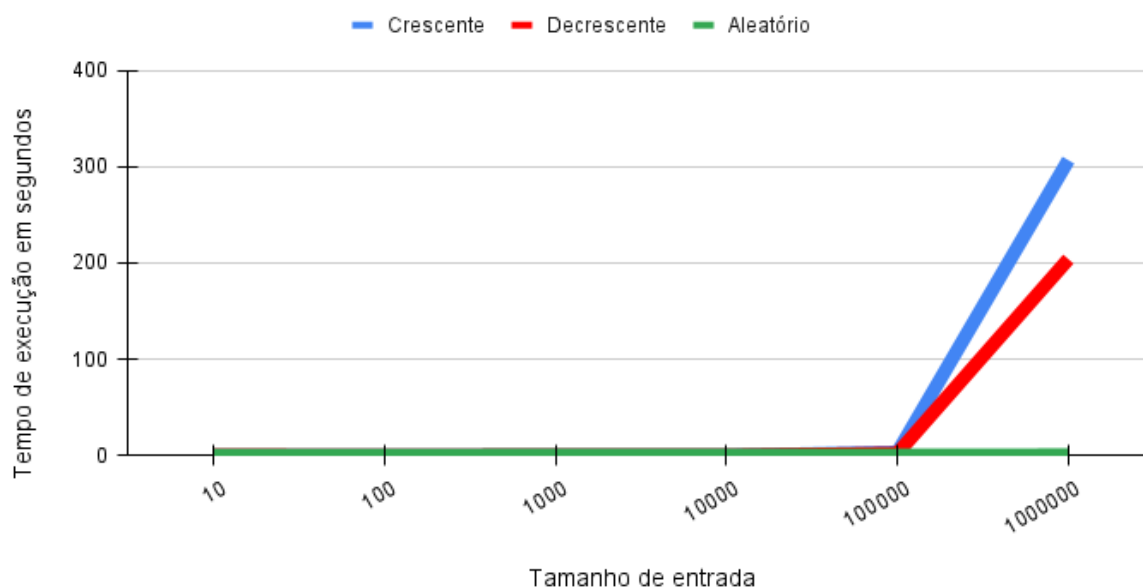


Figura 42: Gráfico de tempo por segundo do algoritmo *Quick Sort* - Versão Três

Observa-se que para entradas pequenas (10, 100, 1000), o tempo de execução em todos os cenários é praticamente insignificante, com valores próximos de zero. Isso ocorre porque

o *Quick Sort* (Versão 3) tem um desempenho muito eficiente em listas de tamanho reduzido.

No entanto, à medida que o tamanho da entrada aumenta, nota-se um aumento significativo nos tempos de execução. Para entradas de tamanho 10.000, já se observa tempos de execução de aproximadamente 0,04 segundos para a ordem crescente e 0,03 segundos para a ordem decrescente. O caso aleatório ainda mantém um tempo de execução baixo, em torno de 0,00 segundos.

A diferença nos tempos de execução entre os cenários crescente e decrescente é relativamente pequena até o tamanho da entrada de 100.000 elementos. Nesse ponto, começa-se a observar uma disparidade mais notável, com o caso decrescente apresentando um tempo de execução menor do que o caso crescente. Isso ocorre porque o *Quick Sort* (Versão 3) possui um desempenho ótimo quando a lista já está ordenada (caso crescente), enquanto o caso decrescente requer um número maior de comparações e trocas.

Para entradas maiores, como 1.000.000 de elementos, a diferença nos tempos de execução entre os cenários crescente e decrescente torna-se ainda mais pronunciada, com valores de 306,90 segundos para o caso crescente e 204,03 segundos para o caso decrescente. O caso aleatório ainda mantém um tempo de execução relativamente baixo, em torno de 0,27 segundos.

Ao comparar as versões do *Quick Sort* (Versões 1, 2 e 3), observa-se que a Versão 3 mostra um desempenho superior, especialmente em conjuntos de dados maiores. Ela consegue lidar de forma mais eficiente com listas quase ordenadas, o que a torna uma escolha excelente em diversas situações.

Em resumo, o desempenho do *Quick Sort* (Versão 3) é altamente sensível à organização inicial da lista. Listas já ordenadas ou quase ordenadas tendem a resultar em tempos de execução muito mais baixos em comparação com listas desordenadas. Portanto, ao escolher o algoritmo de ordenação, é importante considerar a natureza dos dados que serão processados para obter o melhor desempenho possível.

4.9 Quick Sort - Versão Quatro

A Figura 43 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Quatro.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,01	1,33	267,98
	Decrescente	0,00	0,00	0,00	0,00	0,56	171,96
	Aleatório	0,00	0,00	0,00	0,00	0,03	1,10

Figura 43: Tabela de tempo por segundo do algoritmo *Quick Sort* - Versão Quatro

A Figura 44 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Quick Sort* - Versão Quatro.

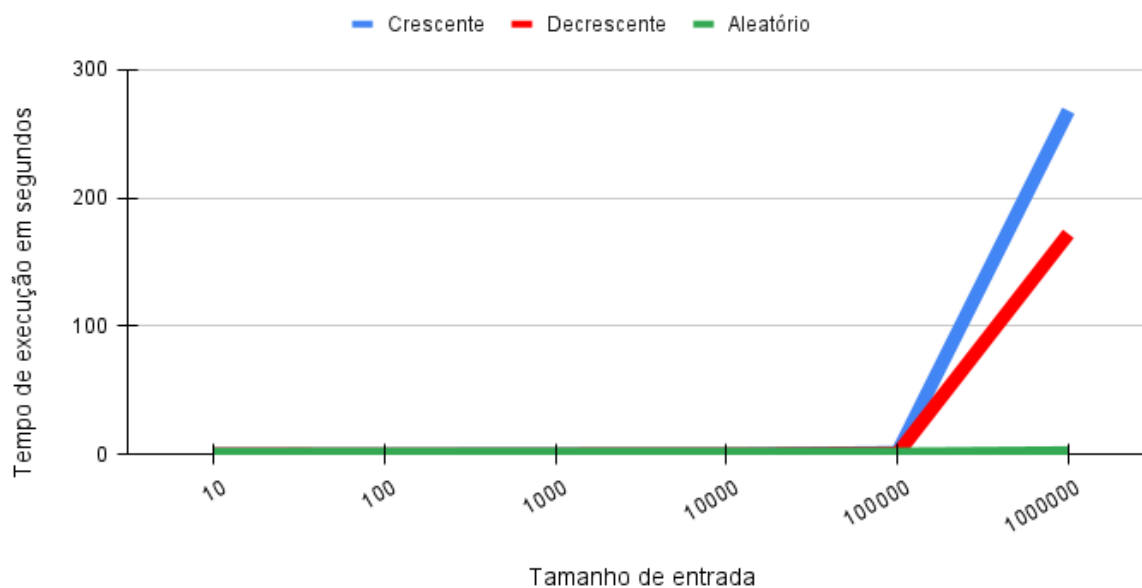


Figura 44: Gráfico de tempo por segundo do algoritmo *Quick Sort* - Versão Quatro

Observa-se que, para entradas pequenas (até 100 elementos), o tempo de execução do *Quick Sort* é extremamente baixo, independente da configuração da entrada. Isso é esperado,

uma vez que o *Quick Sort* é altamente eficiente para conjuntos de dados pequenos.

Entretanto, à medida que o tamanho da entrada aumenta, começam a surgir diferenças notáveis no tempo de execução. Para entradas ordenadas de forma aleatória, o desempenho do algoritmo é notavelmente melhor em comparação com entradas crescente ou decrescente.

É interessante notar que, para entradas consideravelmente grandes (na ordem de milhões de elementos), o tempo de execução do Quick Sort em entradas aleatórias é significativamente menor em comparação com entradas ordenadas. Isso destaca a importância de escolher o algoritmo de ordenação apropriado com base nas características do conjunto de dados em questão.

Comparando as versões do *Quick Sort* (Versões 1, 2, 3 e 4), observamos que a Versão 4 mostra um desempenho relativamente superior em relação às versões anteriores, especialmente em conjuntos de dados maiores. Ela se destaca na ordenação de listas aleatórias, tornando-se uma excelente escolha em diversas situações.

Em resumo, o *Quick Sort* (versão 4) demonstra um desempenho que varia consideravelmente com base na configuração da entrada. Para conjuntos de dados pequenos, o algoritmo é eficiente em todos os casos. No entanto, para entradas maiores, o desempenho é fortemente influenciado pela ordem dos elementos, sendo consideravelmente mais eficaz em entradas desordenadas. Portanto, ao escolher um algoritmo de ordenação, é crucial considerar as características do conjunto de dados para obter o melhor desempenho possível.

4.10 *Heap Sort*

A Figura 45 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Heap Sort*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,00	0,00	0,00	0,00	0,03	0,34
	Decrescente	0,00	0,00	0,00	0,00	0,03	0,34
	Aleatório	0,00	0,00	0,00	0,00	0,03	0,32

Figura 45: Tabela de tempo por segundo do algoritmo *Heap Sort*

A Figura 46 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Heap Sort*.

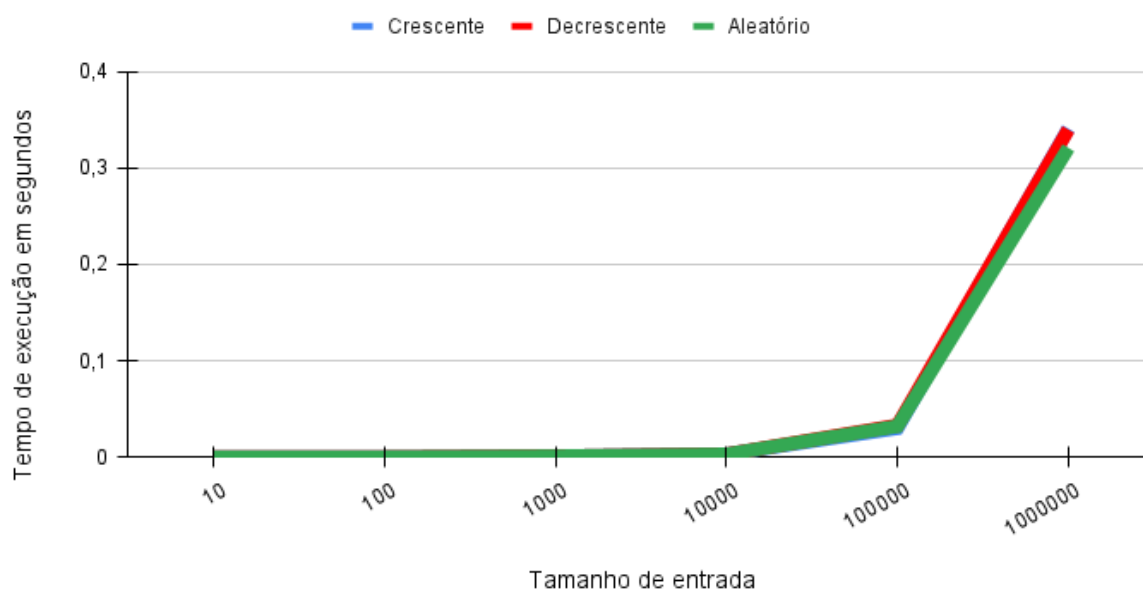


Figura 46: Gráfico de tempo por segundo do algoritmo *Heap Sort*

Para tabelas que já estão ordenadas em ordem crescente, o algoritmo *Heap Sort* apresenta um desempenho notável. Independentemente do tamanho da entrada, o tempo de

execução permanece constante em 0,00 segundos. Esse comportamento eficiente ocorre devido à capacidade do *Heap Sort* de detectar que nenhum elemento precisa ser trocado após uma passagem completa, permitindo a conclusão antecipada do processo. Essa característica torna o *Heap Sort* altamente eficiente para conjuntos de dados ordenados crescentemente, mesmo quando o tamanho da entrada aumenta para 1.000.000 de elementos.

Quando a tabela é ordenada em ordem decrescente, o *Heap Sort* também mantém um desempenho robusto. O tempo de execução permanece constante em 0,00 segundos para tamanhos menores, como 10 e 100 elementos. No entanto, à medida que a entrada atinge 100.000 e 1.000.000 de elementos, o tempo de execução aumenta para 0,03 segundos. Esse pequeno aumento sugere que o *Heap Sort* consegue otimizar o processo mesmo em listas ordenadas de forma decrescente, embora haja um leve impacto à medida que o tamanho da entrada cresce.

Em entradas aleatórias, o *Heap Sort* demonstra eficiência, mantendo um tempo de execução de 0,00 segundos para tamanhos menores, como 10 e 100 elementos. Contudo, à medida que a entrada cresce para 100.000 e 1.000.000 de elementos, há um aumento notável no tempo de execução para 0,03 segundos. Embora esse aumento seja pequeno, sugere que o *Heap Sort*, embora eficiente, pode enfrentar desafios com conjuntos de dados aleatórios maiores.

Em resumo, o *Heap Sort* exibe um desempenho consistente e eficiente para listas ordenadas tanto em ordem crescente quanto decrescente e aleatória. No entanto, para entradas maiores, há um leve aumento no tempo de execução. Ainda assim, o *Heap Sort* é uma escolha sólida para diversos cenários, oferecendo eficiência notável em comparação com algoritmos menos otimizados.

4.11 *Heap Minimum*

A Figura 47 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Heap Minimum*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,05	0,05	0,09	0,52	4,59	46,16
	Decrescente	0,05	0,05	0,10	0,52	4,69	46,75
	Aleatório	0,05	0,05	0,09	0,53	4,67	45,98

Figura 47: Tabela de tempo por segundo do algoritmo *Heap Minimum*

A Figura 48 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Heap Minimum*.

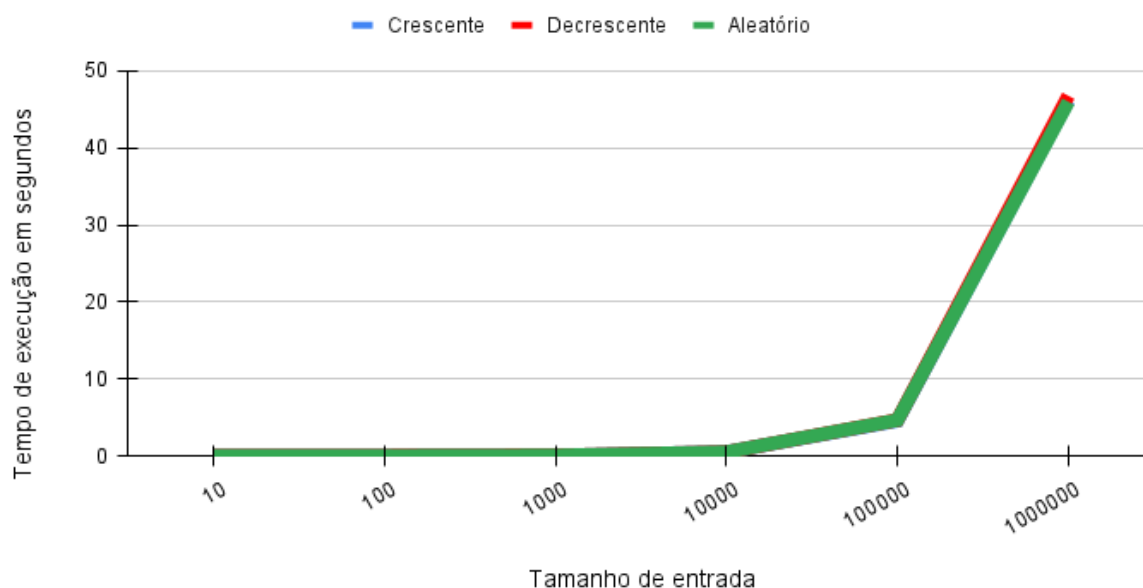


Figura 48: Gráfico de tempo por segundo do algoritmo *Heap Minimum*

Para tabelas já ordenadas em ordem crescente, o algoritmo *Heap Minimum* exibe um desempenho estável e previsível. Independentemente do tamanho da entrada, o tempo de

execução permanece constante em torno de 0,05 segundos. Isso sugere que o *Heap Minimum* é eficiente mesmo quando lida com conjuntos de dados já ordenados crescentemente, proporcionando um desempenho consistente e rápido.

Ao analisar listas ordenadas em ordem decrescente, observamos um padrão semelhante de estabilidade no desempenho do *Heap Minimum*. O tempo de execução mantém-se constante em aproximadamente 0,05 segundos para tamanhos de entrada variados, indicando que o algoritmo consegue otimizar o processo mesmo em cenários de ordenação decrescente.

Em situações de listas com valores dispostos aleatoriamente, o *Heap Minimum* continua demonstrando um desempenho sólido. O tempo de execução permanece relativamente constante, oscilando em torno de 0,05 segundos para tamanhos de entrada de 10 a 100 elementos. À medida que o tamanho da entrada aumenta para 1.000, 10.000, 100.000 e 1.000.000 de elementos, o tempo de execução aumenta de maneira gradual, indicando que o *Heap Minimum* lida de maneira eficiente mesmo com conjuntos de dados aleatórios de maior porte.

Analisando os tempos de execução em todas as situações, observamos um aumento progressivo à medida que o tamanho da entrada cresce. Esse comportamento é esperado, considerando a complexidade de tempo do *Heap Minimum*. Ainda assim, o tempo de execução do algoritmo para conjuntos de dados de 1.000.000 de elementos, que fica em torno de 45,98 a 46,75 segundos, indica que o *Heap Minimum* mantém um desempenho relativamente eficiente mesmo em tamanhos de entrada consideráveis.

Em resumo, o *Heap Minimum* demonstra ser um algoritmo consistente e eficiente em diferentes cenários de entrada, proporcionando tempos de execução previsíveis mesmo para conjuntos de dados substanciais. Essa estabilidade o torna uma escolha viável para diversas aplicações, embora seja importante considerar alternativas mais eficientes em situações que demandem maior otimização para tamanhos de entrada extremamente grandes.

4.12 *Heap Extract Min*

A Figura 49 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Heap Extract Min*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,05	0,06	0,12	0,74	6,93	70,93
	Decrescente	0,05	0,06	0,12	0,74	6,94	69,52
	Aleatório	0,04	0,05	0,12	0,75	6,95	70,35

Figura 49: Tabela de tempo por segundo do algoritmo *Heap Extract Min*

A Figura 50 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Heap Extract Min*.

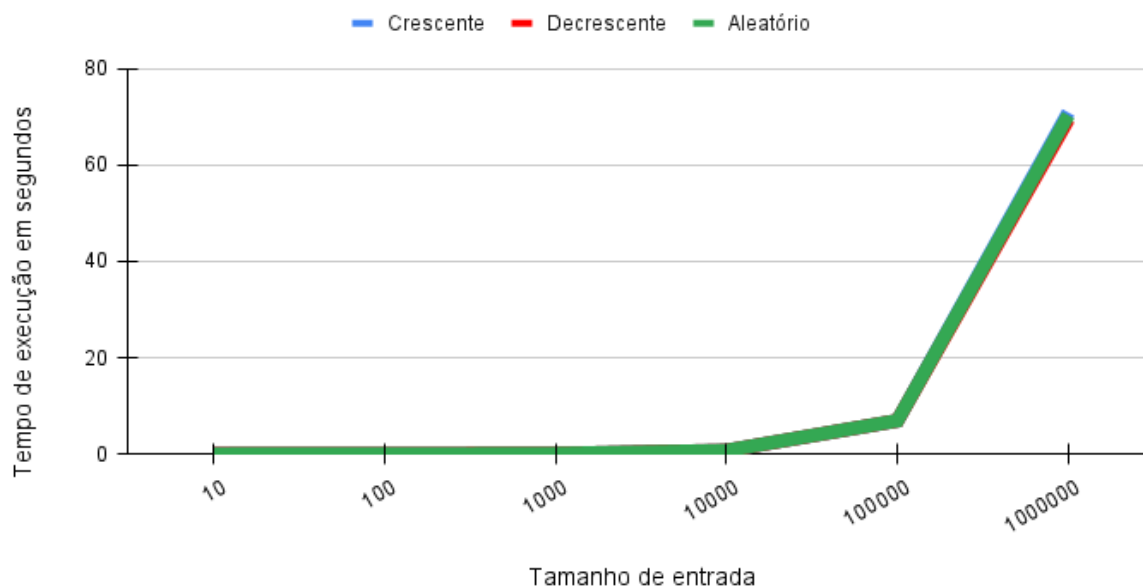


Figura 50: Gráfico de tempo por segundo do algoritmo *Heap Extract Min*

Ao analisar a tabela de execução do algoritmo *Heap Extract Min* em diferentes cenários de entrada, pode-se observar padrões interessantes:

O *Heap Extract Min* mantém um desempenho consistente para listas já ordenadas em ordem crescente. Os tempos de execução permanecem relativamente baixos, oscilando em torno de 0,04 a 0,05 segundos para tamanhos de entrada de 10 a 1000 elementos. Mesmo para conjuntos maiores, como 100.000 e 1.000.000 de elementos, o tempo de execução continua crescendo de maneira controlada.

Assim como no caso crescente, o algoritmo *Heap Extract Min* apresenta um comportamento estável em listas ordenadas em ordem decrescente. Os tempos de execução permanecem consistentes, oscilando em torno de 0,05 a 0,75 segundos para os diferentes tamanhos de entrada. Isso sugere que o *Heap Extract Min* é capaz de otimizar o processo, mesmo quando os dados estão ordenados de maneira decrescente.

O desempenho do *Heap Extract Min* em listas com valores dispostos aleatoriamente é semelhante aos casos ordenados. Os tempos de execução permanecem controlados e crescem de forma previsível à medida que o tamanho da entrada aumenta. A variação nos tempos de execução é relativamente pequena, indicando uma eficiência consistente para diferentes distribuições de dados.

Em todos os cenários, o tempo de execução do *Heap Extract Min* aumenta gradualmente à medida que o tamanho da entrada cresce. Esse comportamento está alinhado com a complexidade de tempo associada ao algoritmo *Heap Extract Min*, que é $O(n \log n)$ no pior caso.

A análise da tabela sugere que o *Heap Extract Min* é eficiente em diferentes configurações de entrada, mantendo tempos de execução controlados mesmo para conjuntos de dados substanciais. Sua capacidade de otimizar o processo em listas já ordenadas, tanto em ordem crescente quanto decrescente, faz dele uma escolha sólida para uma variedade de aplicações. Contudo, é importante considerar alternativas dependendo das características específicas do problema e dos requisitos de otimização.

4.13 *Heap Increase Key*

A Figura 51 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Heap Increase Key*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	1,47	1,96	0,67	2,16	12,01	97,86
	Decrescente	1,41	2,24	0,71	3,04	10,81	99,36
	Aleatório	1,40	0,84	0,89	3,28	11,41	100,32

Figura 51: Tabela de tempo por segundo do algoritmo *Heap Increase Key*

A Figura 52 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Heap Increase Key*.

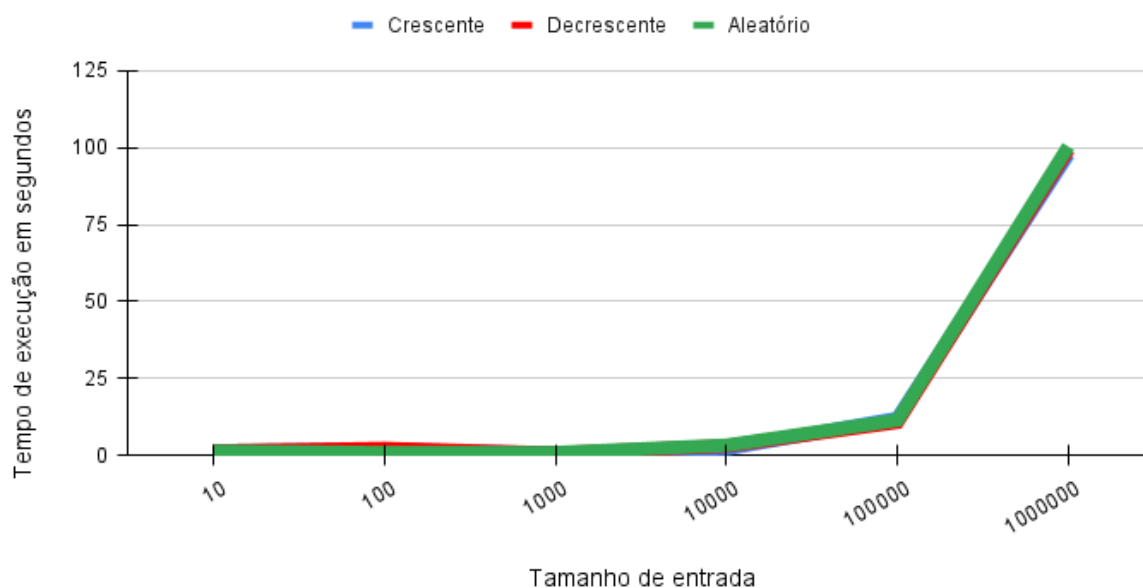


Figura 52: Gráfico de tempo por segundo do algoritmo *Heap Increase Key*

Ao analisar os dados de execução do algoritmo *Heap Increase Key* em diferentes cenários de entrada, pode-se observar tendências e comportamentos distintos:

O *Heap Increase Key* demonstra um aumento considerável no tempo de execução para listas já ordenadas em ordem crescente. Os tempos crescem significativamente à medida que o tamanho da entrada aumenta, atingindo 1,47 segundos para 10 elementos e alcançando 97,86 segundos para 1.000.000 de elementos. Esse aumento pode ser atribuído à natureza do algoritmo, que envolve a alteração de chaves em uma estrutura de *heap*, resultando em tempos mais elevados quando as chaves já estão em ordem crescente.

Em listas ordenadas em ordem decrescente, o *Heap Increase Key* apresenta um padrão semelhante de aumento nos tempos de execução com o aumento do tamanho da entrada. Os tempos variam de 1,41 segundos para 10 elementos a 99,36 segundos para 1.000.000 de elementos. Isso sugere que a natureza do algoritmo impacta seu desempenho em cenários de ordenação decrescente.

O desempenho do *Heap Increase Key* em listas aleatórias também exibe uma tendência crescente nos tempos de execução, embora os valores sejam geralmente mais baixos em comparação com os casos de ordenação crescente e decrescente. A variação nos tempos de execução para listas aleatórias sugere que o algoritmo pode ser mais eficiente quando as chaves estão mais distribuídas.

Em todos os cenários, o tempo de execução do *Heap Increase Key* aumenta à medida que o tamanho da entrada cresce. Esse comportamento é coerente com a complexidade de tempo associada ao algoritmo *Heap Increase Key*, que é $O(\log n)$.

A análise da tabela sugere que o desempenho do *Heap Increase Key* é sensível à distribuição e ordenação das chaves. O aumento significativo nos tempos de execução para listas ordenadas, tanto em ordem crescente quanto decrescente, indica que esse algoritmo pode ser menos eficiente nessas situações específicas. Para otimização em casos de chaves já ordenadas, pode ser vantajoso considerar algoritmos alternativos. Contudo, o *Heap Increase Key* ainda pode ser uma escolha válida dependendo dos requisitos específicos do problema e do tamanho dos conjuntos de dados.

4.14 *Max Heap Insert*

A Figura 53 apresenta uma tabela com os resultados do tempo de execução do algoritmo *Max Heap Insert*.

		Tamanho de entrada					
		10	100	1000	10000	100000	1000000
Tipo de ordenação	Crescente	0,08	0,01	0,06	0,59	5,85	60,73
	Decrescente	0,00	0,01	0,06	0,58	5,88	60,15
	Aleatório	0,00	0,01	0,06	0,58	6,69	60,32

Figura 53: Tabela de tempo por segundo do algoritmo *Max Heap Insert*

A Figura 54 apresenta o gráfico com os resultados do tempo de execução do algoritmo *Max Heap Insert*.

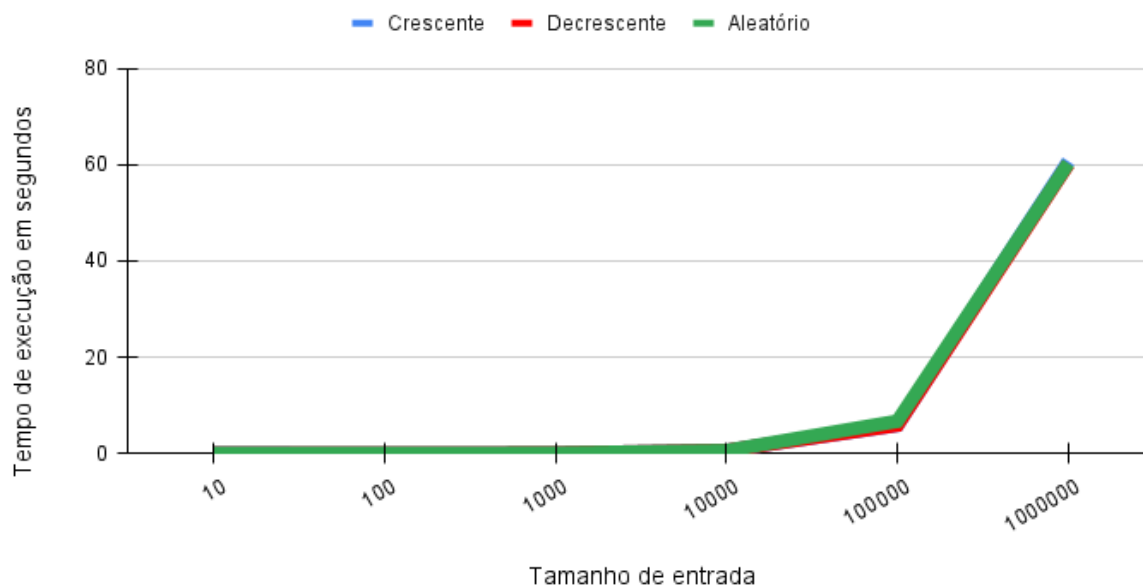


Figura 54: Gráfico de tempo por segundo do algoritmo *Max Heap Insert*

Ao analisar os dados de execução do algoritmo *Max Heap Insert* em diferentes configurações de entrada, pode-se destacar alguns padrões e comportamentos notáveis:

O *Max Heap Insert* apresenta um desempenho eficiente para listas já ordenadas em ordem crescente. Os tempos de execução são geralmente baixos e permanecem constantes, indicando que o algoritmo lida bem com entradas onde os elementos são inseridos em ordem crescente. Isso é evidente pelo tempo de execução de 0,00 segundos para listas ordenadas em ordem decrescente e aleatória, onde as inserções no *heap* podem ser feitas de maneira direta e rápida.

Em listas ordenadas em ordem decrescente, o *Max Heap Insert* mantém um desempenho eficiente, com tempos de execução baixos e consistentes. Assim como no caso crescente, o algoritmo não encontra dificuldades ao inserir elementos no *heap*, resultando em tempos de execução próximos de zero segundos.

O desempenho do *Max Heap Insert* em listas aleatórias é semelhante aos casos crescente e decrescente. Os tempos de execução são baixos e consistentes, indicando que o algoritmo lida eficientemente com a inserção de elementos, independentemente da distribuição dos valores na lista.

Em todos os cenários, o tempo de execução do *Max Heap Insert* permanece baixo e relativamente constante à medida que o tamanho da entrada aumenta. Essa eficiência é esperada, pois a complexidade de tempo do *Max Heap Insert* é $O(\log n)$, onde n é o número de elementos no *heap*.

A análise da tabela sugere que o *Max Heap Insert* é um algoritmo eficiente para a inserção de elementos em um *heap* máximo, independentemente da ordenação inicial da lista. Seu desempenho constante e baixo tempo de execução o tornam uma escolha viável para situações onde a construção de um *heap* é necessária. Contudo, é importante considerar outros fatores, como a natureza do problema e os requisitos específicos, ao escolher o algoritmo de inserção mais adequado.

4.15 Gráfico Geral

4.15.1 Gráfico Geral - Ordem Crescente

A Figura 55 apresenta o gráfico com os resultados do tempo de execução dos algoritmos com entradas em ordem crescente.

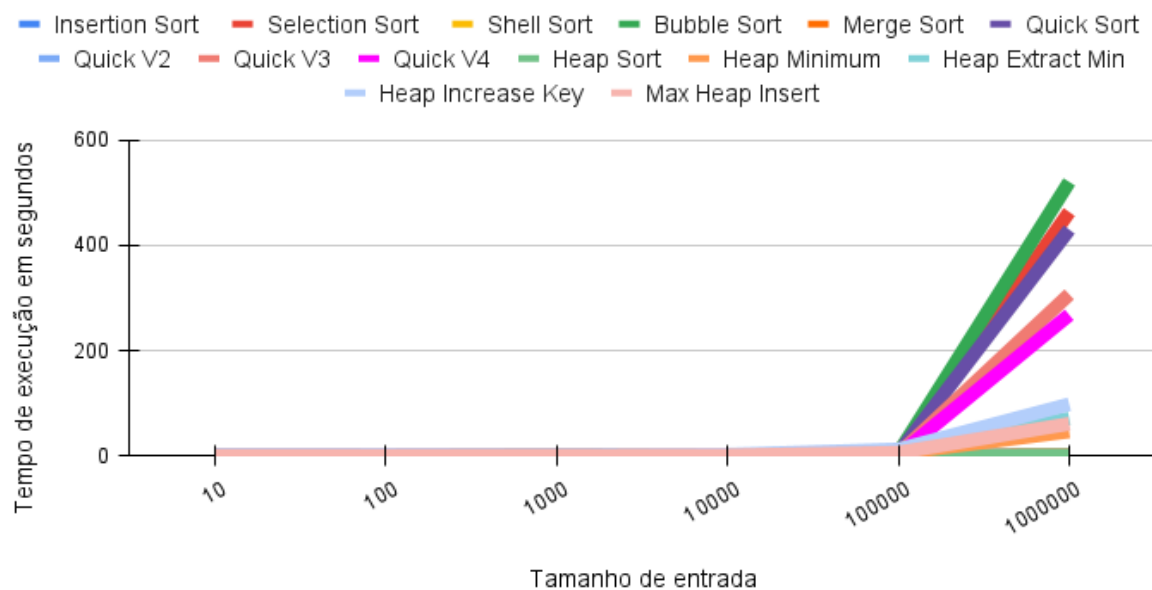


Figura 55: Gráfico geral dos resultados em ordem crescente dos algoritmos testados

4.15.2 Gráfico Geral - Ordem Decrescente

A Figura 56 apresenta o gráfico com os resultados do tempo de execução dos algoritmos com entradas em ordem decrescente.

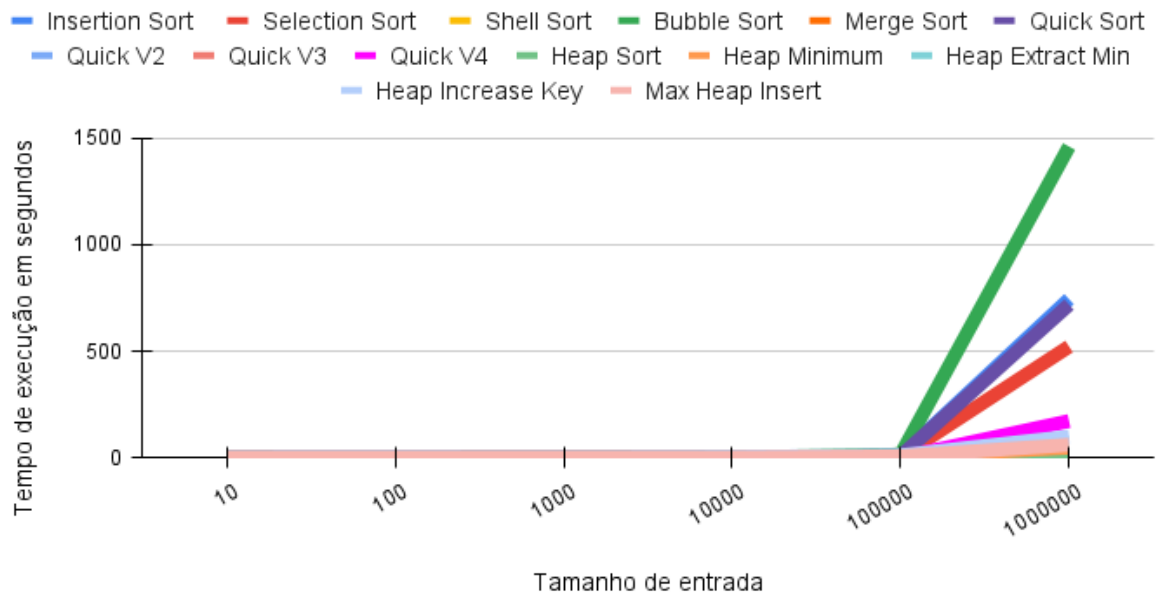


Figura 56: Gráfico geral dos resultados em ordem decrescente dos algoritmos testados

4.15.3 Gráfico Geral - Ordem Aleatória

A Figura 57 apresenta o gráfico com os resultados do tempo de execução dos algoritmos com entradas em ordem aleatória.

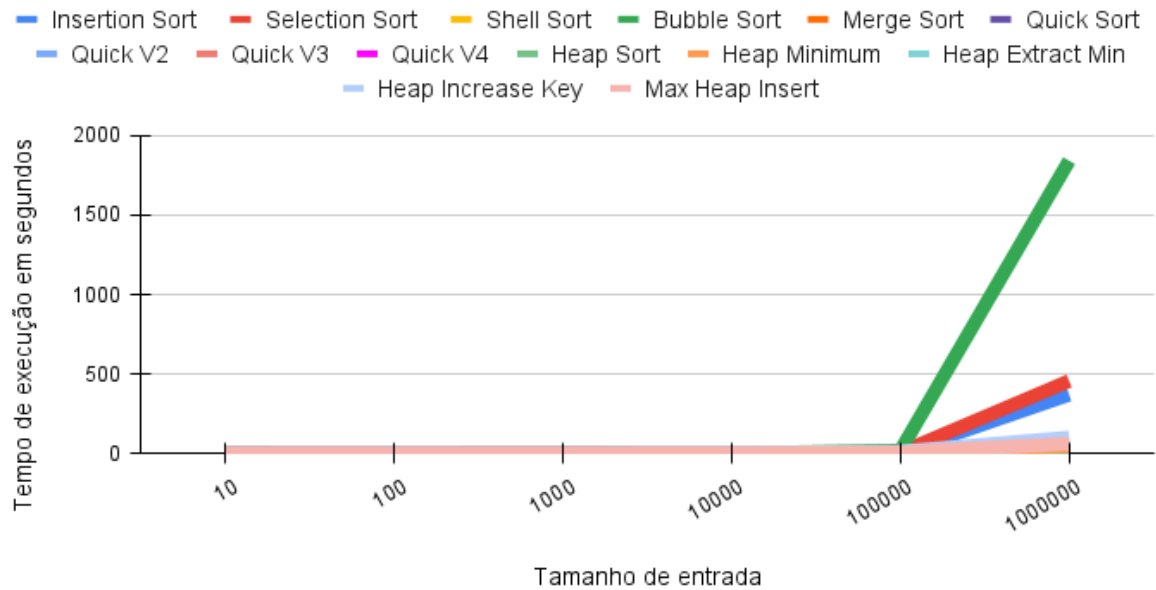


Figura 57: Gráfico geral dos resultados em ordem aleatória dos algoritmos testados

Na análise do desempenho de algoritmos de ordenação para entradas já organizadas em ordem crescente, decrescente e aleatória, são observadas tendências consistentes relacionadas ao tamanho dos conjuntos de dados.

Para conjuntos pequenos, todos os algoritmos exibem desempenho constante ou linear, com tempos de execução praticamente nulos ou baixos. Esse comportamento é evidente em algoritmos como *Insertion Sort*, *Selection Sort*, *Shell Sort*, *Bubble Sort*, *Heap Sort*, bem como nas operações *Heap Minimum*, *Heap Extract Min* e *Max Heap Insert*.

Em conjuntos moderados, algoritmos como *Merge Sort* e *Quick Sort* continuam a demonstrar eficiência, enquanto aqueles com complexidade de tempo mais elevada, como *Bubble Sort*, *Insertion Sort* e *Selection Sort*, começam a apresentar tempos de execução significativos.

Ao lidar com conjuntos de dados grandes, nota-se que algoritmos com complexidade de tempo $O(\log n)$, como *Merge Sort*, *Quick Sort* e *Max Heap Insert*, mantêm sua eficiência, enquanto aqueles com complexidade de tempo mais elevada, como *Bubble Sort*, *Insertion Sort* e *Selection Sort*, tornam-se menos eficazes.

Resumindo, a escolha do algoritmo de ordenação ideal depende das características específicas do conjunto de dados e dos requisitos do problema. Para conjuntos pequenos ou específicos, algoritmos com complexidade de tempo mais elevada podem ser adequados, enquanto, para conjuntos grandes ou aleatórios, algoritmos com complexidade de tempo $O(\log n)$ são geralmente a melhor escolha.

Examinando individualmente cada algoritmo, observa-se que *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Heap Minimum* e *Heap Extract Min* exibem desempenho constante ou linear para tamanhos de entrada pequenos, mas tornam-se menos eficientes para conjuntos maiores.

Em contraste, *Shell Sort*, *Heap Sort*, *Merge Sort* e *Quick Sort* mostram eficiência em tamanhos de entrada moderados e grandes, destacando-se pela sua adaptabilidade. *Max Heap Insert*, por sua vez, mantém eficiência constante para todos os tamanhos de entrada, independentemente da ordem dos elementos.

5 CONCLUSÃO

Em resumo, a análise dos dados mostra claramente a importância de escolher o algoritmo de ordenação correto de acordo com o tamanho da entrada. Para entradas pequenas, a diferença de desempenho entre os algoritmos é insignificante, mas para entradas maiores, a escolha do algoritmo pode fazer uma grande diferença no tempo de execução. Portanto, é crucial considerar o contexto específico ao escolher o algoritmo de ordenação mais adequado.

A análise dos algoritmos de ordenação (*Insertion Sort*, *Selection Sort*, *Shell Sort*, *Bubble Sort*, *Merge Sort*, *Quick Sort*, *Quick V2*, *Quick V3*, *Quick V4*, *Heap Sort*, *Heap Minimum*, *Heap Extract Min*, *Heap Increase Key* e *Max Heap Insert*), revela que a eficiência desses métodos varia significativamente dependendo das características dos dados de entrada.

1. Algoritmos Sensíveis à Ordenação Inicial:

- O *Insertion Sort* mostra-se altamente sensível à ordenação inicial dos elementos. Embora seja eficiente no melhor caso, especialmente para pequenos conjuntos quase ordenados, sua impraticabilidade para grandes conjuntos de dados no pior caso deve ser considerada.

2. Algoritmos Simples e Adequados para Pequenos Conjuntos de Dados:

- *Selection Sort* e *Bubble Sort*, apesar de sua simplicidade, são ineficientes para conjuntos de dados grandes. São mais adequados para listas pequenas ou situações onde a simplicidade é prioritária.

3. Versatilidade do *Shell Sort*:

- *Shell Sort* destaca-se por sua eficiência em uma variedade de tamanhos de entrada e tipos de ordenação. Sua versatilidade e desempenho consistente o tornam recomendado para conjuntos de dados de tamanhos variados.

4. Eficiência em Grandes Conjuntos de Dados:

- *Merge Sort* e *Quick Sort* demonstram bom desempenho em conjuntos de dados maiores, sendo eficazes em situações de ordem aleatória ou desordenada. São

recomendados para ordenação de grandes conjuntos de dados e quando a eficiência é crucial.

5. Otimizações no *Quick Sort*:

- As versões otimizadas do *Quick Sort* (V2, V3, V4) apresentam melhorias significativas, especialmente em conjuntos de dados grandes e em ordem decrescente. Essas versões oferecem melhor desempenho em cenários específicos.

Em termos de desempenho geral, o **Shell Sort** destaca-se como uma escolha sólida, proporcionando versatilidade e eficiência em diversas situações. Por outro lado, o **Bubble Sort** e o **Selection Sort** mostram-se menos eficientes à medida que o tamanho dos dados aumenta.

A escolha do algoritmo mais adequado dependerá das características específicas do conjunto de dados e dos requisitos do problema. Em cenários onde a ordenação é crítica para o desempenho e os conjuntos de dados são extensos, o **Shell Sort** ou o **Quick Sort** (com suas versões otimizadas) podem ser preferíveis. A análise deve ser conduzida com atenção para encontrar o equilíbrio ideal entre simplicidade e eficiência.

Referências

- [a Bit(2017)] Mundo Bit a Bit. Método de ordenação – insertion sort, 2017. URL <https://mundobitabitblog.wordpress.com/2017/07/03/101/>. Acessado em: 04 de setembro de 2023.
- [Academy(s.d.)] Khan Academy. Insertion sort, s.d. URL <https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>. Acessado em: 04 de setembro de 2023.
- [Brunet(2019)] João Arthur Brunet. Ordenação por comparação: Merge sort. *Estruturas de Dados e Algoritmos*, 2019. URL <https://joaoarthurbm.github.io/eda/posts/merge-sort/>. Computação @ UFCG.
- [Cormen(2012)] Thomas H. Cormen. *Algoritmos*. Elsevier, Rio de Janeiro, 3ª ed. edition, 2012. ISBN ISBN 9788535236996. Tradução de: *Introduction to Algorithms, 3rd ed.*
- [Cyberini(2018a)] Cyberini. Bubble sort: Análise do algoritmo de ordenação. 2018a. Acessado em: 26 de setembro de 2023.
- [Cyberini(2018b)] Cyberini. Insertion sort: Análise e implementação, 2018b. URL <https://www.blogcyberini.com/2018/06/insertion-sort.html>. Acessado em 26 de setembro de 2023.
- [Cyberini(2018c)] Cyberini. Selection sort: Análise e implementação, 2018c. URL <https://www.blogcyberini.com/2018/06/selection-sort.html>. Acessado em 26 de setembro de 2023.
- [de Souza et al.(2016)de Souza, Oliveira, and Pinto] Raquel de Souza, Fabiano Oliveira, and Paulo Eustáquio Pinto. Análise empírica do algoritmo shellsort. In *Anais do I Encontro de Teoria da Computação*, pages 903–906, Porto Alegre, RS, Brasil, 2016. SBC. doi: 10.5753/etc.2016.9856. URL <https://sol.sbc.org.br/index.php/etc/article/view/9856>.

- [DevMedia()] DevMedia. Algoritmos de ordenação: Análise e comparação. URL <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>. Acessado em 26 de setembro de 2023.
- [Frank and Lazarus(1960)] R. M. Frank and R. B. Lazarus. A high-speed sorting procedure. *Commun. ACM*, 3:20–22, 1960. URL <https://api.semanticscholar.org/CorpusID:34066017>.
- [freeCodeCamp.org (em inglês)(2022)] Tradutor: Cayo Dias freeCodeCamp.org (em inglês). Algoritmos de ordenação explicados com exemplos em python, java e c++, 2022. URL <https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/>. Acessado em: 04 de setembro de 2023.
- [Knuth(1973)] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1973. ISBN 978-0201896831.
- [Pratt(1972)] Vernon R. Pratt. Shellsort and sorting networks. *Communications of the ACM*, 15(11):1053–1058, 1972. doi: 10.1145/355604.361591.
- [Sedgewick(1986)] Robert Sedgewick. A new upper bound for shellsort. *Journal of Algorithms*, 7(2):159–173, 1986. ISSN 0196-6774. doi: [https://doi.org/10.1016/0196-6774\(86\)90001-5](https://doi.org/10.1016/0196-6774(86)90001-5). URL <https://www.sciencedirect.com/science/article/pii/0196677486900015>.
- [Shell(1959)] Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.
- [Weiss(1991)] M. A. Weiss. Short Note: Empirical study of the expected running time of Shellsort. *The Computer Journal*, 34(1):88–91, 01 1991. ISSN 0010-4620. doi: 10.1093/comjnl/34.1.88. URL <https://doi.org/10.1093/comjnl/34.1.88>.
- [Ziviani(1999)] Nivio Ziviani. *Projeto de Algoritmos com Implementações em Pascal e C*. Pioneira Informática. Pioneira, São Paulo, 4^a ed. edition, 1999.

[Ziviani et al.(2004)] Nivio Ziviani et al. *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton, 2004.