Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Kayky Cristof Eduardo Domingos Silva" - "8118"

"Análise e comparação de algoritmos de ordenação"

Rio Paranaíba - MG

2024

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Kayky Cristof Eduardo Domingos Silva" - "8118"

"Análise e comparação de algoritmos de ordenação"

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

2024

1 RESUMO

Algoritmos de ordenação são fundamentais para organizar dados em uma sequência ordenada, facilitando o acesso, análise e manipulação destes dados. Esses algoritmos recebem uma entrada e os reordenam gerando uma saída.

Existem diferentes tipos de algoritmos de ordenação, cada um com suas características e desempenho. Alguns dos mais conhecidos incluem o Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Bubble Sort. Cada um desses algoritmos é analisado de acordo com a complexidade de tempo e espaço, que mede a eficiência de execução e o uso de memória, especialmente em grandes volumes de dados. Em geral, os algoritmos de ordenação são estudados para identificar quais são mais adequados para situações específicas, como pequenas coleções de dados ou sistemas onde a memória é limitada.

Além disso, algoritmos de ordenação servem como introdução a conceitos mais complexos em estrutura de dados e análise de algoritmos, sendo utilizados em áreas que envolvem grande processamento de dados, como pesquisa e aprendizado de máquina.

Neste trabalho vamos, entender, implementar e analisar diferentes algoritmos de ordenação com o foco em estudar suas variações e comportamentos em diferentes contextos. Faremos uma análise de complexidade, considerando o tempo de execução, abordando os cenários de melhor, pior e caso médio para cada algoritmo.

A partir dos dados coletados, será feita uma comparação da eficiência dos algoritmos de ordenação, usando como critério o tempo de execução em diferentes tipos de entradas e tamanho: crescente, decrescente e aleatória. Cada algoritmo terá seu tempo de execução calculado e organizados em tabelas. Com essas tabelas, será possível construir gráficos para visualizar melhor o comportamento de cada algoritmo sob diferentes condições.

Essas representações visuais nos permitirão identificar padrões de desempenho e estabelecer quais algoritmos são mais eficientes em cada cenário. Esse processo possibilitará uma visão prática e teórica sobre o impacto da escolha dos algoritmos de ordenação em aplicações reais, resultando em uma conclusão fundamentada sobre a adequação de cada algoritmo para diferentes tipos de problemas.

Sumário

2 INTRODUÇÃO

2.1 Algoritmos e estruturas de dados

Os algoritmos fazem parte do nosso dia-a-dia. Instruções para medicamentos, uma receita culinária são exemplos de algoritmos. De acordo com [3] um algoritmo é como uma sequência de ações executáveis para obter a solução de um problema. Segundo Dijkstra (1971) um **algoritmo** é uma descrição de um padrão de comportamento que é expresso em conjuntos finitos de ações. Ao executar a operação $a + b$ percebe o mesmo padrão de comportamento, mesmo que os valores sejam diferentes para a e b.

Estruturas de dados e algoritmos estão diretamente ligados. Não é possível estudar estruturas de dados sem considerar os algoritmos ligado a elas, logo como a escolha dos algoritmos dependem da representação da estrutura de dados. Para resolver problemas é necessário abstrair a realidade, através de uma definição de um conjunto de dados que representa o mundo real.

A eficiência de um algoritmo muitas vezes depende da estrutura de dados utilizada para armazenar e manipular informações. Por exemplo, para um problema precisa de acesso rápido aos dados, pode ser mais adequado utilizar uma tabela hash, enquanto que para um conjunto de dados que necessite de ordenação, uma árvore binária pode ser melhor.

A análise da complexidade de algoritmos é uma área fundamental dentro da ciência da computação. Essa análise permite avaliar o desempenho de um algoritmo, possibilitando a escolha da solução mais adequada para um problema. Compreender o tempo de execução e o uso de memória é crucial, especialmente em aplicações que lidam com grandes volumes de dados, onde até pequenas melhorias de desempenho podem resultar em economias significativas de tempo e recursos.

A inter-relação entre algoritmos e estruturas de dados não apenas influencia a eficiência das soluções, mas também a sua implementação prática em diversas linguagens de programação. A escolha de uma linguagem pode impactar a maneira como um algoritmo é desenvolvido. A compreensão de ambos os conceitos é essencial para qualquer profissional que deseje atuar na área, desenvolvendo soluções eficientes e eficazes.

2.2 Análise assintótica de algoritmos

A análise assintótica de algoritmos é uma técnica utilizada na ciência da computação para descrever o comportamento de um algoritmo à medida que o tamanho da entrada cresce. Ela fornece uma estimativa do tempo de execução de um algoritmo, ignorando fatores constantes.

Seus principais objetivos estão ligados à compreensão e previsão do desempenho de um algoritmo de maneira abstrata. Um dos propósitos fundamentais é generalizar o desempenho, avaliar como o algoritmo se comporta independentemente do ambiente ou da máquina em que é executado. Isso permite que a análise seja aplicável em qualquer contexto, desconsiderando fatores específicos de hardware, como velocidade do processador ou quantidade de memória disponível.

Outro objetivo importante é facilitar a comparação entre algoritmos, principalmente quando lidamos com entradas grandes. Ao focar no comportamento assintótico, é possível determinar qual algoritmo é mais eficiente em termos de tempo, ajudando a escolher a solução mais adequada.

Além disso, a análise assintótica ajuda a identificar gargalos no algoritmo, partes que têm um impacto significativo no desempenho. Com essa identificação, é possível otimizar essas etapas críticas, melhorando a eficiência do algoritmo.

2.2.1 Notações da análise assintótica

A análise assintótica é frequentemente expressa usando três principais notações:

- **Notação Big-O (O)**: Descreve o limite superior do tempo de execução no pior caso.
  Exemplo: $O(n^2)$ indica que o tempo de execução cresce, no máximo, quadraticamente em relação ao tamanho da entrada.

- **Notação Ômega (Ω)**: Descreve o limite inferior do tempo de execução, ou seja, o melhor caso.
  Exemplo: $\Omega(n)$ indica que o algoritmo executa pelo menos linearmente no melhor caso.

- **Notação Theta (Θ)**: Descreve o tempo de execução exato quando o pior e melhor caso coincidem.

Exemplo: $\Theta(n \log n)$ indica que o tempo de execução é exatamente proporcional a $n \log n$.

2.2.2 Categorias de complexidade

As categorias de complexidade, em ordem crescente de tempo de execução, são:

- **Complexidade Constante** ($O(1)$): O tempo de execução não depende do tamanho da entrada, sendo fixo.
- **Complexidade Logarítmica** ($O(\log n)$): O tempo de execução cresce lentamente à medida que a entrada aumenta.
- **Complexidade Linear** ($O(n)$): O tempo de execução cresce proporcionalmente ao tamanho da entrada.
- **Complexidade Linearítmica** ($O(n \log n)$): O tempo de execução é proporcional ao produto do tamanho da entrada e seu logaritmo.
- **Complexidade Quadrática** ($O(n^2)$): O tempo de execução cresce com o quadrado do tamanho da entrada.
- **Complexidade Cúbica** ($O(n^3)$): O tempo de execução cresce com o cubo do tamanho da entrada.
- **Complexidade Exponencial** ($O(2^n)$): O tempo de execução cresce exponencialmente em relação ao tamanho da entrada.
- **Complexidade Fatorial** ($O(n!)$): O tempo de execução cresce de forma extremamente rápida, seguindo o fatorial do tamanho da entrada.

A análise assintótica é uma ferramenta fundamental na avaliação de algoritmos, permitindo prever seu desempenho. Ao focar no comportamento do algoritmo para entradas muito grandes, ela fornece uma visão clara sobre a eficiência relativa entre diferentes soluções, facilitando a escolha do algoritmo mais adequado para cada problema.

Por meio das notações O, Ω e Θ, é possível descrever de maneira precisa os limites superior, inferior e exato do tempo de execução, respectivamente. Além disso, a classificação em categorias de complexidade ajuda a entender como o desempenho do algoritmo evolui conforme o tamanho da entrada cresce, possibilitando a identificação de potenciais gargalos e áreas para otimização.

Assim, o uso da análise assintótica é indispensável para o desenvolvimento de algoritmos eficientes e escaláveis, contribuindo para a resolução eficaz de problemas computacionais.

3 ALGORITMOS

3.1 Insertion Sort

O primeiro algoritmo abordado será o algoritmo de ordenação por inserção para resolver problemas de ordenação.

Entrada: Uma sequência de n números $(a_1, a_2, \ldots, a_n)$.

Saída: Uma permutação (reordenação) $(a_1, a_2, \ldots, a_n)$ da sequência de entrada, tal que $a_1 \leq a_2 \leq \ldots \leq a_n$.

Os números que iremos ordenar são chamados de **chaves**.

De acordo com a Figura 18, o processo de ordenação por inserção é representado de maneira visual, destacando sua semelhança com o ato de organizar cartas em um baralho. Este algoritmo funciona inserindo cada elemento na posição correta em relação aos anteriores, como ilustrado na imagem, que foi adaptada do livro de referência [1].

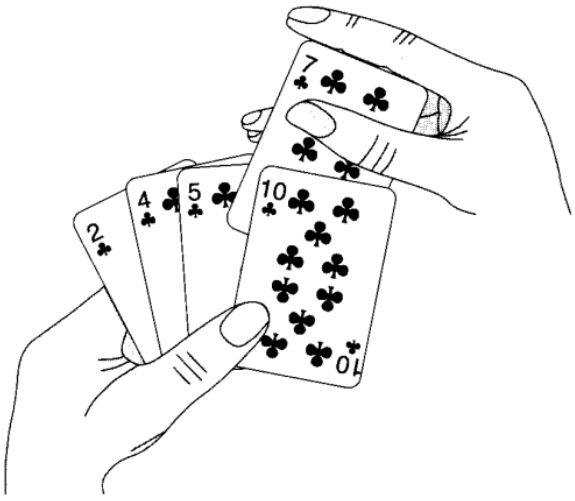

Figura 1: Ordenando cartas com o uso de ordenação por inserção do livro[1]

Este algoritmo é eficiente para ordenar um pequeno conjunto de elementos. Segundo o livro[1] sua ordenação funciona de maneira similar a ordenação de cartas de um jogo de pôquer. Começamos com a mão esquerda vazia com as cartas viradas com a face para baixo. Logo, removemos uma carta de cada vez, inserindo a mesma na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, comparamos esta com cada uma das cartas que já está na mão, da direita para a esquerda. Em cada instante, as cartas da

mão esquerda são ordenadas. Tomamos como parâmetro um arranjo A[1...n] contendo uma sequência de comprimento n que deverá ser ordenada. Os números da entrada são **ordenados no local**, os números são reorganizados dentro de um arranjo A com no máximo um número constante deles armazenado. O arranjo A conterá a sequência armazenada quando o insertion-sort terminar.

De acordo com a Figura 2, a operação do algoritmo de ordenação por inserção é aplicada ao arranjo $A = [5, 2, 4, 6, 1, 3]$. A figura ilustra cada etapa do processo, demonstrando como os elementos são reorganizados progressivamente para alcançar a ordenação final. Este exemplo foi extraído do livro de referência [1].

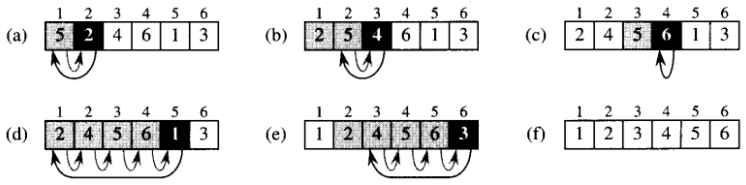

Figura 2: Operação de insertion sort sobre o arranjo A = [5,2,4,6,1,3] do livro[1].

Os índices do arranjo ficam em cima dos retângulos e seus valores armazenados dentro dos retângulos. Em cada iteração o retângulo preto contém a chave obtida de A[j] que é comparada com os valores contidos nos retângulos sombreados à sua esquerda. As setas sombreadas indicam valores do arranjo deslocados uma posição a direita e setas pretas para onde a chave é deslocada.

De acordo com a Figura 3, é apresentada a implementação do algoritmo de ordenação por inserção na linguagem C. O código exemplifica a lógica do algoritmo, demonstrando como os elementos são comparados e inseridos em suas posições corretas dentro do arranjo. Esta implementação foi desenvolvida pelo autor para fins ilustrativos.

```c
void insertion_Sort( int vetor, int tam ) {
    int i;
    int j;
    int chave;

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

Figura 3: Insertion-sort implementado na linguagem C pelo autor

3.1.1 Teste de Mesa para o Insertion Sort

De acordo com a Tabela 1, é apresentado um teste de mesa simplificado para o algoritmo de ordenação por inserção (Insertion Sort). O exemplo utiliza a sequência $A = [5, 2, 4, 6, 7]$ e detalha cada etapa do processo, incluindo as variáveis intermediárias e a modificação do arranjo em cada iteração. Este teste foi desenvolvido pelo autor, com base em explicações realizadas em sala de aula, para facilitar a compreensão do funcionamento do algoritmo.

j	chave = A[j]	i = j - 1	A[i+j] = A[i]	i = i - 1	A[i+1] = chave	Sequência
2	chave = A[2]	i = 2 - 1	A[1+1] = A[1]	i = 1 - 1	A[0+1] = 2	5, 2, 4, 6, 7
2	chave = 2	i = 1	A[2] = 5	i = 0	A[1] = 2	2, 5, 4, 6, 7
4	chave = A[3]	i = 3 - 1	A[2+1] = A[2]	i = 2 - 1	A[1+1] = 4	2, 5, 4, 6, 7
4	chave = 4	i = 2	A[3] = 5	i = 1	A[2] = 4	2, 4, 5, 6, 7
6	chave = A[4]	i = 4 - 1	null	null	A[3+1] = 6	2, 4, 5, 6, 7
6	chave = 6	i = 3	null	null	A[4] = 6	2, 4, 5, 6, 7
7	chave = A[5]	i = 2 - 1	null	null	A[4+1] = 6	2, 4, 5, 6, 7
7	chave = 7	i = 1	null	null	A[5] = 6	2, 4, 5, 6, 7

Tabela 1: Teste de Mesa simplificado para o Insertion Sort com a sequência $5, 2, 4, 6, 7$ feito pelo autor (Este teste foi inspirado no teste apresentado em sala de aula).

3.2 Bubble Sort

O algoritmo Bubble Sort é um método de ordenação simples, ideal para ordenar um pequeno conjunto de elementos. Compara pares de elementos adjacentes e faz a troca de posição quando estão na ordem contrária. Seu objetivo é ordenar os valores em forma crescente, então, a posição atual é comparada com a próxima posição e, se a posição atual for menor que a posterior é realizada a troca. Caso o contrário a troca não é feita é se passa para a próxima posição.

Na 1ª iteração, leva o maior elemento para a última posição.

Na 2 ª iteração, leva o segundo maior elemento, para a penúltima posição, e assim sucessivamente.

Esse processo lembra uma bolha de ar subindo em um líquido, é um método de fácil entendimento, de fácil implementação, porém muito **ineficiente**.

De acordo com a Figura 4, é apresentada a implementação do algoritmo Bubble Sort na linguagem C. O código exemplifica o funcionamento do algoritmo, que realiza comparações adjacentes e troca os elementos de posição para ordenar o arranjo. Esta implementação foi desenvolvida pelo autor como exemplo prático para ilustrar o método de ordenação.

```c
void bubble_Sort( int *vetor, int tam ) {

    int i;
    int j;
    int chave;

    for ( i = 0; i < tam - 1; i++ ) {
        for ( j = 0; j < tam - i - 1; j++ ) {
            if ( vetor[j] > vetor[j + 1] ) {
                chave = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = chave;
            }
        }
    }
}
```

Figura 4: Bubble-Sort implementado na linguagem C pelo autor

3.2.1 Teste de Mesa para o Bubble Sort

De acordo com a Tabela 2, é apresentado um teste de mesa simplificado para o algoritmo Bubble Sort. O exemplo utiliza uma sequência inicial e detalha o comportamento do algoritmo em cada iteração, destacando as comparações e trocas realizadas entre elementos adjacentes. Este teste foi desenvolvido pelo autor como uma ferramenta prática para demonstrar o funcionamento do Bubble Sort e suas etapas de ordenação.

i	j	$A[j] > A[j+1]$	$A[j] = A[j+1]$	Sequência
1	1	$5 > 3$	$A[1] = A[2]$	3, 5, 8, 4, 6
1	2	$5 > 8$	null	3, 5, 8, 4, 6
1	3	$8 > 4$	$A[3] = A[4]$	3, 5, 4, 8, 6
1	4	$8 > 6$	$A[4] = A[5]$	3, 5, 4, 6, 8
2	1	$3 > 5$	null	3, 5, 4, 6, 8
2	2	$5 > 4$	$A[2] = A[3]$	3, 4, 5, 6, 8
2	3	$5 > 6$	null	3, 4, 5, 6, 8
3	1	$3 > 4$	null	3, 4, 5, 6, 8
3	2	$4 > 5$	null	3, 4, 5, 6, 8
4	1	$3 > 4$	null	3, 4, 5, 6, 8

Tabela 2: Teste de Mesa simplificado para o Bubble Sort com a sequência feita pelo autor.

3.3 Selection Sort

O Selection Sort é um algoritmo de ordenação simples e intuitivo, ideal para pequenos conjuntos de dados. Sua ideia básica é dividir o arranjo em duas partes: uma parte ordenada e uma parte não ordenada. Ele encontra o menor elemento da parte não ordenada e o coloca na posição correta da parte ordenada. O processo é repetido até que todo o arranjo esteja ordenado.

De acordo com a Figura 5, é apresentada a implementação do algoritmo Selection Sort na linguagem C. O código exemplifica como o algoritmo encontra o menor (ou maior) elemento de uma sequência e o coloca na posição correta através de trocas sucessivas. Esta implementação foi desenvolvida pelo autor para ilustrar de maneira prática o funcionamento do Selection Sort.

```c
void selectionSort( int *vetor, int tam ) {
    int i;
    int j;
    int chave;
    int minIndex;

    for ( i = 0; i < tam - 1; i++ ) {
        minIndex = i;
        for ( j = i + 1; j < tam; j++ ) {
            if ( vetor[j] < vetor[minIndex] ) {
                minIndex = j;
            }
        }
        chave = vetor[i];
        vetor[i] = vetor[minIndex];
        vetor[minIndex] = chave;
    }
}
```

Figura 5: Selection-Sort implementado na linguagem C pelo autor

Em resumo, o Selection Sort identifica, a cada iteração, o menor elemento na parte não ordenada e o move para sua posição correta, expandindo a parte ordenada do arranjo até que ele esteja completamente ordenado.

3.3.1 Exemplo de Execução

Considere o arranjo `vetor = {64, 25, 12, 22, 11}`:

- **Primeira Iteração** (`i = 0`): menor elemento é `11`. Troca com `vetor[0]`. Resultado: {`11, 25, 12, 22, 64`}.
- **Segunda Iteração** (`i = 1`): menor elemento é `12`. Troca com `vetor[1]`. Resultado: {`11, 12, 25, 22, 64`}.
- **Terceira Iteração** (`i = 2`): menor elemento é `22`. Troca com `vetor[2]`. Resultado: {`11, 12, 22, 25, 64`}.
- **Quarta Iteração** (`i = 3`): menor elemento é `25`. Resultado final: {`11, 12, 22, 25, 64`}.

O arranjo agora está ordenado.

3.4 Shell Sort

O algoritmo **Shell Sort** é uma variação do algoritmo **Insertion Sort**, que busca melhorar sua eficiência em relação à ordenação de um conjunto de elementos. Em vez de comparar apenas elementos adjacentes, o Shell Sort compara e troca elementos distantes uns dos outros, permitindo que a ordenação ocorra de forma mais rápida.

Inicialmente, o algoritmo divide o conjunto em subgrupos com base em um valor chamado **gap** e realiza uma ordenação parcial nesses subgrupos. Conforme o algoritmo avança, o valor do gap diminui, e os elementos são comparados e trocados com distâncias menores entre si.

O processo continua até que o gap seja reduzido a 1, quando o algoritmo realiza uma última passagem de Insertion Sort para ordenar o conjunto completamente. O Shell Sort, portanto, melhora a eficiência do Insertion Sort ao permitir que o arranjo seja reorganizado de forma mais eficaz durante as etapas iniciais.

De acordo com a Figura 6, é apresentada a implementação do algoritmo Shell Sort na linguagem C. O código demonstra como o algoritmo realiza a ordenação utilizando intervalos (gaps) que diminuem progressivamente, permitindo que os elementos sejam reorganizados de maneira mais eficiente do que no método de ordenação por inserção simples. Esta implementação foi desenvolvida pelo autor para ilustrar o funcionamento do Shell Sort.

```c
void shellSort(int *vetor, int tam) {

    int i;
    int j;
    int gap;
    int chave;

    for ( gap = tam / 2; gap > 0; gap /= 2 ) {
        for ( i = gap; i < tam; i++ ) {
            chave = vetor[i];
            for ( j = i; j >= gap && vetor[j - gap] > chave; j -= gap ) {
                vetor[j] = vetor[j - gap];
            }
            vetor[j] = chave;
        }
    }
}
```

Figura 6: Shell-Sort implementado na linguagem C pelo autor

3.4.1 Exemplo de Execução

Considere o arranjo `vetor = {55, 42, 17, 63, 25}`:

- **Primeira Iteração** (`gap = 2`):
  - Comparar `vetor[2] = 17` com `vetor[0] = 55` sem troca.
  - Comparar `vetor[3] = 63` com `vetor[1] = 42` sem troca.
  - Comparar `vetor[4] = 25` com `vetor[2] = 17` troca `vetor[4]` e `vetor[2]`. Resultado: {`55, 42, 25, 63, 17`}.
- **Segunda Iteração** (`gap = 1`):
  - Comparar `vetor[1] = 42` com `vetor[0] = 55` troca `vetor[1]` e `vetor[0]`. Resultado: {`42, 55, 25, 63, 17`}.
  - Comparar `vetor[2] = 25` com `vetor[1] = 55` troca `vetor[2]` e `vetor[1]`. Resultado: {`42, 25, 55, 63, 17`}.
  - Comparar `vetor[3] = 63` com `vetor[2] = 55` sem troca.
  - Comparar `vetor[4] = 17` com `vetor[3] = 63` troca `vetor[4]` e `vetor[3]`. Resultado: {`42, 25, 55, 17, 63`}.
  - Comparar `vetor[3] = 17` com `vetor[2] = 55` troca `vetor[3]` e `vetor[2]`. Resultado: {`42, 25, 17, 55, 63`}.
  - Comparar `vetor[2] = 17` com `vetor[1] = 25` troca `vetor[2]` e `vetor[1]`. Resultado: {`42, 17, 25, 55, 63`}.
  - Comparar `vetor[1] = 17` com `vetor[0] = 42` troca `vetor[1]` e `vetor[0]`. Resultado: {`17, 42, 25, 55, 63`}.
- **Resultado Final**: {`17, 25, 42, 55, 63`}.

3.5 Merge Sort

O **Merge Sort**, é um de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses sub-problemas através da recursividade) e Conquistar (após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).

3.5.1 Divisão e Conquista

O paradigma divisão e conquista consiste em dividir o problema a ser resolvido em partes menores para encontrar as soluções para as partes e então combinar as soluções obtidas em uma solução global.

Etapas:

1. **Dividir**: O Problema em subproblemas menores e similares.
2. **Conquistar**: Resolvendo os subproblemas recursivamente.
3. **Combinar**: As soluções dos subproblemas na solução do problema original.

Segundo [2] o funcionamento do Merge Sort se baseia, numa rotina cujo nome é **Merge**.

3.5.2 Merge

Merge é a rotina que combina dois arranjos ordenados em um outro arranjo também ordenado. Durante o processo de Merge Sort, a lista é repetidamente dividida em subarranjos até que cada subarranjo tenha um único elemento, que é considerado ordenado. A função **merge** é responsável por combinar esses subarranjos ordenados em uma único arranjo ordenado.

O processo de **merge** ocorre da seguinte forma:

- Dois subarranjos ordenados, denominados ‘começo‘ e ‘fim‘, são passados como entrada.
- A função começa comparando o primeiro elemento de cada subarranjo.
- O menor valor entre os dois primeiros elementos das subarranjos é adicionado ao arranjo final.

- Esse processo de comparação e adição ao arranjo final continua até que todos os elementos de uma das subarranjos sejam consumidos.

- Após isso, os elementos restantes do outro subarranjo, que já estão ordenados, são simplesmente adicionados ao arranjo final.

A função **merge** garante que a combinação das subarranjos seja feita de forma eficiente, preservando a ordem dos elementos, o que resulta em uma arranjo ordenado.

De acordo com a Figura 7, é apresentada a implementação da função Merge, que é um componente essencial do algoritmo Merge Sort, na linguagem C. O código ilustra como os sub-arranjos são combinados de maneira ordenada para formar o arranjo final. Esta implementação foi desenvolvida pelo autor para demonstrar o funcionamento da função Merge.

```c
void merge( int *vetor, int comeco, int meio, int fim ) {
    int com1 = comeco;
    int com2 = meio+1;
    int comAux = 0;
    int tam = fim-comeco+1;
    int *vetAux = NULL;

    aloca_Memoria( &vetAux, tam );

    while( com1 <= meio && com2 <= fim ) {
        if( vetor[com1] < vetor[com2] ) {
            vetAux[comAux] = vetor[com1];
            com1++;
        } else {
            vetAux[comAux] = vetor[com2];
            com2++;
        }
        comAux++;
    }

    while( com1 <= meio ) {
        vetAux[comAux] = vetor[com1];
        comAux++;
        com1++;
    }

    while( com2 <= fim ) {
        vetAux[comAux] = vetor[com2];
        comAux++;
        com2++;
    }

    for( comAux = comeco; comAux <= fim; comAux++ ) {
        vetor[comAux] = vetAux[comAux-comeco];
    }

    free( vetAux );
}
```

Figura 7: Função Merge implementada utilizando a linguagem C pelo autor

3.5.3 O Merge Sort

A função **merge sort** é a parte recursiva do algoritmo **Merge Sort**. Esta função divide o arranjo recursivamente até que o mesmo seja reduzido a subarranjos com um elemento.Em seguida a função **merge** é utilizado para combinar estes subarranjos de maneira ordenada.

Assim a função **merge sort** trabalha da seguinte forma:

1. **Dividir**: Divide o arranjo original em duas metades, repetindo esse processo até que o arranjo seja reduzido a subarranjos de tamanho 1.

2. **Combinar**: Após os subarranjos serem divididos, a função começa a combinar esses subaranjos utilizando a função **merge** de forma ordenada.

De acordo com a Figura 8, é apresentada a implementação completa do algoritmo Merge Sort na linguagem C. O código exemplifica como o algoritmo divide recursivamente o arranjo em sub-arranjos menores e, em seguida, utiliza a função Merge para combinar essas partes de maneira ordenada. Esta implementação foi desenvolvida pelo autor para ilustrar o funcionamento do Merge Sort de forma prática e detalhada.

```c
void mergeSort( int *vetor, int comeco, int fim ) {

    if ( comeco < fim ) {
        int meio = ( fim+comeco ) / 2;

        mergeSort( vetor, comeco, meio );
        mergeSort( vetor, meio+1, fim );
        merge( vetor, comeco, meio, fim );
    }
}
```

Figura 8: Função Merge Sort implementada utilizando a linguagem C pelo autor

3.6 Quick Sort

O Quick Sort é um algoritmo de ordenação eficiente que utiliza a técnica de divisão e conquista. Seu objetivo é ordenar um arranjo, o dividindo em subarranjos menores e os ordenando de forma recursiva, até que o arranjo esteja ordenado.

3.6.1 Funcionamento do Quick Sort

Pivô: O Quick Sort, possui um **pivô**, no qual este é um elemento escolhido do arranjo. que serve como ponto de refêrencia para dividir o arranjo em duas partes durante o processo de ordenação. Seu objetivo é que os elementos menores do que ele fiquem a sua esquerda e os maiores a sua direita, com a condição na qual o pivô será colocado na posição correta no final da partição.

Dividir: O arranjo $A[p...r]$ é particionado em dois subarranjos $A[p...q-1]$ e $A[q+1...r]$, tal que cada elemento de $A[p...q-1]$ é menor que ou igual $A[q]$ que é igual ou menor a cada elemento de $A[q+1...r]$.

Conquistar: Os dois subarranjos $A[p...q-1]$ e $A[q+1...r]$ são ordenados de forma recursiva.

Combinar: Como os dois subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los.

3.6.2 Escolha do Pivô como primeiro elemento

A Figura 9 apresenta a implementação da função **particiona_primeiro** do algoritmo Quick Sort, onde o primeiro elemento do arranjo é escolhido como pivô. Essa abordagem é simples e direta, mas pode comprometer o desempenho do algoritmo em certos casos, como quando o arranjo está quase ordenado ou completamente ordenado. Nesses casos, a complexidade do algoritmo pode chegar a $O(n^2)$, caracterizando o pior cenário para o Quick Sort.

```c
int particiona_primeiro( int *vetor, int inicio, int fim ) {
    int pivo_indice = inicio;

    troca( vetor, pivo_indice, fim );

    return particiona( vetor, inicio, fim );
}
```

Figura 9: Função particiona_primeiro implementada utilizando a linguagem C pelo autor

3.6.3 Pivô Baseado na Média dos Elementos

A Figura 10 apresenta a implementação da função **particiona_media** do algoritmo Quick Sort, onde o pivô é escolhido com base na média dos elementos do arranjo. Essa abordagem visa melhorar o desempenho do algoritmo em relação à escolha do primeiro elemento como pivô, pois tende a reduzir a probabilidade de ocorrer o pior caso ($O(n^2)$) quando o arranjo está quase ordenado.

```c
int particiona_media( int *vetor, int inicio, int fim ) {
    double soma = 0;
    int tamanho = fim - inicio + 1;
    int i;

    for ( i = inicio; i <= fim; i++ ) {
        soma += vetor[i];
    }

    double media = soma / tamanho;

    int pivo_indice = inicio;
    double menor_diferenca = fabs( vetor[inicio] - media );

    for ( i = inicio + 1; i <= fim; i++ ) {
        double diferenca = fabs( vetor[i] - media );
        if ( diferenca < menor_diferenca ) {
            menor_diferenca = diferenca;
            pivo_indice = i;
        }
    }

    troca( vetor, pivo_indice, fim );

    return particiona( vetor, inicio, fim );
}
```

Figura 10: Função particiona_media implementada utilizando a linguagem C pelo autor

3.6.4 Pivô Escolhido de Forma Aleatória

A Figura 11 apresenta a implementação da função **particiona_random** do algoritmo Quick Sort, onde o pivô é escolhido de forma aleatória entre os elementos do arranjo. Essa abordagem tem como objetivo evitar os cenários que podem ocorrer quando o pivô é escolhido no início, melhorando o desempenho do algoritmo.

```
int particiona_random( int *vetor, int inicio, int fim ) {
    int pivo_indice = ( rand() % ( fim - inicio + 1 ) ) + inicio;

    troca( vetor, pivo_indice, fim );

    return particiona( vetor, inicio, fim) ;
}
```

Figura 11: Função particiona_random implementada utilizando a linguagem C pelo autor

3.6.5 Particiona

De acordo com a Figura 12, é apresentada a implementação da função particiona do algoritmo Quick Sort na linguagem C. O código exemplifica como o algoritmo seleciona um pivô (neste caso, o último elemento do vetor), organiza os elementos de modo que os menores fiquem à esquerda e os maiores à direita, e, por fim, coloca o pivô na posição correta. Esta implementação foi desenvolvida para ilustrar o funcionamento do particionamento de forma prática e detalhada.

```c
int particiona( int *vetor, int inicio, int fim ) {
    int pivo, pivo_indice, i;

    pivo = vetor[fim];
    pivo_indice = inicio;

    for( i = inicio; i < fim; i++ ) {
        if( vetor[i] <= pivo ) {

            troca( vetor, i, pivo_indice );
            pivo_indice++;
        }
    }

    troca( vetor, pivo_indice, fim );

    return pivo_indice;
}
```

Figura 12: Função particiona implementada utilizando a linguagem C pelo autor

3.6.6 Quick Sort

De acordo com a Figura 13, é apresentada a implementação da função **quickSort** do algoritmo Quick Sort na linguagem C. O código exemplifica como o algoritmo divide recursivamente o vetor em subvetores menores, utilizando a função de particionamento para organizar os elementos ao redor de um pivô escolhido de acordo com o parâmetro **escolha_Pivo**. O pivô pode ser selecionado de três formas: pelo primeiro elemento do vetor, pela média dos elementos ou de forma aleatória. Após a organização dos elementos em torno do pivô, a função recursivamente ordena os subarranjos. Esta implementação foi desenvolvida para ilustrar o funcionamento do Quick Sort de forma prática e detalhada.

```c
void quickSort( int *vetor, int inicio, int fim, int escolha_Pivo ) {

    if ( inicio < fim ) {
        int pivo_indice;

        switch ( escolha_Pivo ) {
            case 1:
                pivo_indice = particiona_primeiro( vetor, inicio, fim );
                break;
            case 2:
                pivo_indice = particiona_media( vetor, inicio, fim );
                break;
            case 3:
                pivo_indice = particiona_random( vetor, inicio, fim) ;
                break;
        }

        quickSort( vetor, inicio, pivo_indice - 1, escolha_Pivo );
        quickSort( vetor, pivo_indice + 1, fim, escolha_Pivo );
    }

}
```

Figura 13: Função Quick Sort implementada utilizando a linguagem C pelo autor

3.7 Heap Sort

O heap sort é um algoritmo de ordenação eficiente, no qual utiliza uma estrutura de dados conhecida como **Heap**, que é utilizado para gerenciar informações durante a execução do algoritmo.

3.7.1 Heaps

Essa estrutura de dados é um objeto do arranjo que pode ser vista como uma árvore binária completa. Cada nó desta árvore representa um elemento do arranjo. Esta árvore está completamente preenchida, exceto talvez no nível mais baixo. Um **arranjo**[**A**] irá representar um heap é possui dois atributos: **comprimento**[**A**], número de elementos do arranjo, e **tamanho-do-heap**[**A**], número de elementos do heap armazenado dentro do arranjo A. A raiz da árvore é A[1], dado o indice i de um nó, os índices do pai **PARENT(i)**, do filho da esquerda **LEFT(i)** e do filho da direita **RIGHT(i)** são cálculados de forma simples. Abaixo segue a representação de um heap máximo visto como uma árvore binária e seu determinado arranjo.

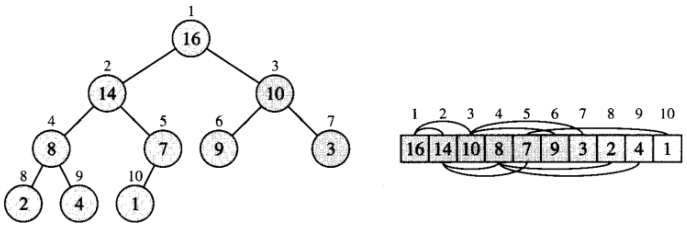

Figura 14: Representação de um heap máximo e o seu respectivo arranjo do livro[1]

Existem dois tipos de heaps binários, **heaps máximos e heaps mínimos**, ambos os casos os valores dos nós tem que satisfazer uma **propriedade do heap**.

3.7.2 Heaps máximos

Para todo nó diferente da raiz:

$$A[\text{PARENT}(i)] \geq A[i]$$

O valor de um nó é no máximo o valor de seu pai. O maior elemento em um heap máximo é armazenado na raiz, e a sub-árvore que tem a raiz em um nó contém valores maiores do que o próprio nó.

3.7.3 Heaps mínimos

Para todo nó diferente da raiz:

$$A[\text{PARENT}(i)] \leq A[i]$$

É organizado de forma oposta do heap máximo. O menor elemento em um heap está na raiz.

3.7.4 Implementação do Heap Sort

Implementamos o algoritmo Heap Sort, utilizando um heap máximo. Para isso, foram definidas três funções: `MAX_HEAPIFY`, `BUILD_MAX_HEAP` e `HEAP_SORT`. A seguir, explicaremos cada uma delas em detalhes.

As figuras abaixo ilustram o funcionamento dessas funções:

```c
void MAX_HEAPIFY( int *vetor, int n, int i ) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if ( left < n && vetor[left] > vetor[largest] ) {
        largest = left;
    }

    if ( right < n && vetor[right] > vetor[largest] ) {
        largest = right;
    }

    if ( largest != i ) {
        int temp = vetor[i];
        vetor[i] = vetor[largest];
        vetor[largest] = temp;

        MAX_HEAPIFY( vetor, n, largest );
    }
}
```

Figura 15: Função `MAX_HEAPIFY` implementada utilizando a linguagem C pelo autor

A função `MAX_HEAPIFY` é responsável por manter a propriedade de max-heap em uma sub-árvore de um arranjo. Caso o nó analisado não seja o maior em relação aos seus filhos, a função troca o valor do nó com o maior de seus filhos e repete o processo recursivamente.

```c
void BUILD_MAX_HEAP( int *vetor, int n ) {
    int i;
    for ( i = n / 2 - 1; i >= 0; i-- ) {
        MAX_HEAPIFY( vetor, n, i );
    }
}
```

Figura 16: Função `BUILD_MAX_HEAP` implementada utilizando a linguagem C pelo autor

A função `BUILD_MAX_HEAP` transforma um arranjo desordenado em uma max-heap. Ela percorre os nós não-folhas do arranjo, aplicando a função `MAX_HEAPIFY` em cada um deles, começando pelo último nó não-folha e indo em direção à raiz.

```c
void HEAP_SORT( int *vetor, int n ) {
    int i;

    BUILD_MAX_HEAP( vetor, n );

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

        MAX_HEAPIFY( vetor, i, 0 );
    }
}
```

Figura 17: Função `HEAP_SORT` implementada utilizando a linguagem C pelo autor

A função `HEAP_SORT` implementa o algoritmo de ordenação Heap Sort. Primeiramente, ela utiliza a função `BUILD_MAX_HEAP` para organizar o arranjo como uma max-heap. Em seguida, remove o maior elemento (raiz da heap) repetidamente, trocando-o com o último elemento do arranjo, reduzindo o tamanho da heap e aplicando `MAX_HEAPIFY` para restaurar a propriedade de max-heap. Abaixo temos uma figura que representa um arranjo que foi ordenado pelo Heap Sort.

```
ARRANJO ORIGINAL: 38 42 7 75 12 98 97 47 62 17
CHAMADO A BUILD_MAX_HEAP: 98 75 97 62 17 7 38 47 42 12
ARRANJO ORDENADO: 7 12 17 38 42 47 62 75 97 98
```

Figura 18: Representação de um arranjo ordenado pelo Heap Sort

3.7.5 Filas de prioridades

Uma das aplicações mais populares de um Heap, são as **filas de prioridades**. Existem dois tipos de filas de prioridades: as **filas de prioridade máxima** e **filas de prioridade mínima**.

Uma fila de prioridades é uma estrutura de dados para manutenção de um conjunto S de elementos. Uma fila de prioridade máxima faz as seguintes operações.

$INSERT(S, x)$: Insere o elemento x no conjunto S.

$MAXIMUM(S)$: Retorna o elemento de S com o maior valor.

$EXTRACT - MAX(S)$: Remove e retorna o elemento com o maior valor.

$INCREASE - KEY(S, x, k)$: Aumenta o tamanho do arranjo para inserir um novo elemento no arranjo.

Uma fila de prioridade máxima como exemplo citado no livro [1] é programar trabalhos em um computador compartilhado. A fila mantém sobre controle os trabalhos a seres executados de acordo com a sua prioridade. Ao um trabalho ser completado ou interrompido, o trabalho de prioridade mais alta começa a executar. Um novo trabalho pode ser inserido a qualquer instante nessa fila de prioridade utilizando a função $INSERT(S, X)$.

Agora iremos falar e explicar um ponto sobre as filas de prioridade mínima, ela irá admitir as operações de:

$INSERT(S, x)$: Insere o elemento x no conjunto S.

$MINIMUM(S)$: Retorna o elemento de S com o menor valor.

$EXTRACT - MIN(S)$: Remove e retorna o elemento com o maior valor.

$DECREASE - KEY(S, x, k)$: Aumenta o tamanho do arranjo para inserir um novo elemento no arranjo.

Basicamente seu funcionamento é similar as filas de prioridade máxima, porém de forma contrária, os filhos sempre serão menores do que pais, a função $BUILD - MIN - HEAP(A)$ garante isto. De acordo com o livro [1] uma aplicação para uma fila de prioridades mínima é um simulador orientado a eventos. Os itens na fila são eventos a serem simulados, cada qual com um tempo de ocorrência associado. Os eventos a serem simulados seguem a essa ordem de ocorrência, pois esse simulação de um evento podem provocar outros eventos a serem simulados no futuro. O programa utiliza a função $EXTRACT - MIN(S)$ para cada etapa de evento simulado. Assim de maneira breve explicamos como o Heap Sort pode ser utilizado como uma fila de prioridades.

O Heap Sort é um algoritmo de ordenação eficiente, utilizado em cenários onde a estabilidade da ordenação não é um requisito. Ele explora a estrutura de dados **Heap**, que permite realizar operações de inserção e remoção de elementos na qual iremos abordar posteriormente.

Embora o Heap Sort não seja estável, sua eficiência e simplicidade tornam-no uma excelente escolha em muitas aplicações práticas, especialmente quando não há necessidade de

preservar a ordem relativa de elementos iguais. Além disso, sua implementação em linguagem C explica como algoritmos de ordenação podem ser otimizados e aplicados de forma prática em sistemas computacionais.

4 ANÁLISE DE COMPLEXIDADE

4.1 Insertion Sort

4.1.1 Análise da ordenação por inserção

O **tempo de execução** do insertion-sort depende da entrada: a ordenação de mil números demora mais do que a ordenação de 10 números. Além disso, o insertion-sort pode demorar tempos diferentes para ordenar duas sequências de entrada do mesmo tamanho pois depende do quão ordenada a entrada está. No geral o tempo de execução de um algoritmo cresce com o **tamanho da entrada**.

Assim apresentamos o insertion-sort com o custo de tempo de cada instrução e o número de vezes que cada instrução é executada. Seja t_j o número de vezes que o teste do loop while é executado para esse valor de j. Quando um loop **for** ou **while** termina, o teste é executado mais uma vez além do corpo do loop testando se a condição é verdadeira ou falsa.

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
2 **do** *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *chave*	c_8	$n-1$

Figura 19: Insertion sort com seus respectivos custos do livro[1]

O tempo de execução do Insertion Sort é a soma dos tempos de execução para cada instrução, uma instrução que demanda c_i passos para ser executada e é executada n vezes, contribuirá com c_in para o tempo de execução total. Para calcular T(n), somamos os produtos das colunas custo e vezes obtendo.

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

4.1.2 Melhor caso

O melhor caso ocorre se o arranjo já se encontra ordenado. Para cada j = 2,3...,n, $A[i] \leq b$ quando i tem seu valor inicial j - 1. Portanto $t_j = 1$ para j = 2,3,...,n, e o tempo de execução do melhor caso é:

$$T(n) = c_i \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot (n-1) + c_8 \cdot (n-1) = (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)$$

Esse tempo de execução pode ser expresso com $a \cdot n + b$ para constantes a e b que dependem dos custos de instrução c_i, assim ele é uma **função linear** de n.

4.1.3 Pior caso

Se o arranjo estiver ordenado em ordem inversa(ordem decrescente) resulta no pior caso. Devemos comparar cada elemento A[j] com cada elemento do subarranjo ordenado inteiro, A[1..j-1], e então $t_j = j$ para 2,3,...,n. Observando que:

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

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

O tempo de execução do pior caso do insertion sort é:

$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c5 \cdot (\frac{n \cdot (n+1)}{2} - 1) + c_6 \cdot (\frac{n \cdot (n-1)}{2}) + c_7 \cdot (\frac{n \cdot (n-1)}{2}) + c_8 \cdot (n-1)$
$= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) \cdot n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)$

Esse tempo de execução no pior caso é expresso por $a \cdot n^2 + b \cdot n + c$ para constantes a, b e c que dependem dos custos de instrução c_i, logo é uma **função quadrática** de **n**.

4.1.4 Caso médio

O caso médio de um algoritmo é quase tão ruim quanto o pior caso. Suponha que seja escolhidos aleatoriamente n números e que se apliquem a eles a ordenação por inserção. Qual será o tempo para descobrir o lugar no subarranjo A[1..j-1] em que se deve inserir o elemento A[j]?Em média, metade dos elementos em A[1..j-1] são menores do que A[j], e metade dos

elementos são maiores. Logo, verificamos metade do subarranjo A[1..j-1], e então $t_j = \frac{j}{2}$. Podemos observar que:

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

$$\sum_{j=2}^{n}(\frac{j}{2} - 1) = (\frac{1}{2})\cdot\sum_{j=2}^{n}(j-1) = (\frac{1}{2})\cdot(\frac{n\cdot(n-1)}{2}) = \frac{n^2-n}{4}$$

Logo o tempo de execução do médio caso é:

$T(n) = c_1\cdot n + c_2\cdot(n-1) + c_4\cdot(n-1) + c5\cdot(\frac{n\cdot(n+1)}{4} - \frac{1}{2}) + c_6\cdot(\frac{n\cdot(n-1)}{4}) + c_7\cdot(\frac{n\cdot(n-1)}{4}) + c_8\cdot(n-1)$
$= (\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4})\cdot n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{4} - \frac{c_6}{4} - \frac{c_7}{4} + c_8)\cdot n - (c_2 + c_4 + \frac{c5}{2} + c_8)$

Desenvolvemos o tempo de execução o tempo de execução do caso médio, e também será uma função quadrática assim como o pior caso.Um problema na análise do caso médio é que não pode ser aparente, o que é uma entrada "média"para um determinado problema.

4.2 Bubble Sort

4.2.1 Análise da ordenação por bolha

O algoritmo de ordenação Bubble Sort, é um algoritmo de fácil implementação como citado alteriormente, ele deve ser utilizado em pequenos conjuntos de elementos, pois na medida no qual a entrada cresce, seu **tempo de execução** também cresce.

Seu funcionamento se baseia, em percorrer repetidamente o arranjo e comparar elementos adjacentes, os trocando de posição, se estiverem na posição errada. Este processo irá se repetir até que o arranjo esteja completamente ordenado.

Na tabela abaixo iremos analisar os custos para cada linha de código do Bubble Sort.

Linha	Código	Custo	Vezes
1	`for i ← 1 to comprimento [A] - 1`	C_1	$n-1$
2	`for j ← 1 to comprimento [A] - i`	C_2	$\sum_{j=1}^{n-1} t_j$
3	`Se A[j] > A[j + 1] então`	C_3	$\sum_{j=1}^{n-1} (t_j - 1)$
4	`Troca A[j] = A[j+1]`	C_4	$\sum_{j=1}^{n-1} (t_j - 1)$

Tabela 3: Análise de Complexidade do Bubble Sort com os custos e número de vezes que cada linha do código é executada.

Iremos agora calcular o $T(n)$ no Bubble Sort.

$$T(n) = c_1 \cdot (n-1) + c_2 \cdot \sum_{j=1}^{n-1} t_j + c_3 \cdot \sum_{j=1}^{n-1} (t_j - 1) + c_4 \cdot \sum_{j=1}^{n-1} (t_j - 1)$$

Sabendo que:

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

$$\sum_{j=1}^{n-1} (t_j - 1) = \frac{n\cdot(n-1)}{2} - (n-1) = \frac{n^2-n-2\cdot(n-1)}{2} = \frac{n^2-n-2n+2}{2} = \frac{n^2-3n+2}{2}$$

Substituindo na fórmula:

$$T(n) = c_1\cdot(n-1) + c_2\cdot(\tfrac{n^2-n}{2}) + c_3\cdot(\tfrac{n^2-3n+2}{2}) + c_4\cdot(\tfrac{n^2-3n+2}{2}) = c_1\cdot n - c_1 + \tfrac{c_2\cdot n^2}{2} - \tfrac{c_2\cdot n}{2} + \tfrac{c_3\cdot n^2}{2} -$$

$\frac{3 \cdot c_3 \cdot n}{2} + \frac{2 \cdot c_3}{2} + \frac{c_4 \cdot n^2}{2} - \frac{3 \cdot c_4 \cdot n}{2} + \frac{2 \cdot c_4}{2} = n^2 \cdot (\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}) + n \cdot (c_1 - \frac{c_2}{2} - \frac{3 \cdot c_3}{2} - \frac{3 \cdot c_4}{2}) + (c_1 + \frac{2 \cdot c_3}{2} + \frac{2 \cdot c_4}{2})$

Esse tempo de execução pode ser expresso como $a \cdot n^2 + b \cdot n + c$ para constantes a, b e c que dependem dos custos de instrução de c_1, logo é uma **função quadrática** de **n**.

O Bubble Sort possui uma peculiaridade, o tempo de execução para o **melhor caso**, **pior caso** e **médio caso** são expressos por uma **função quadrática**, isto não implica necessariamente que o tempo de execução do algoritmo será o mesmo para todos os casos, iremos explicar isto com detalhes.

Embora a complexidade assintótica seja a mesma $O(n^2)$ para o melhor caso, médio caso e pior caso, o número real de operações pode variar. No melhor caso, por mais que a entrada esteja ordenada, irá percorrer (n - 1) elementos do arranjo. Isto significa que, para um arranjo de tamanho n, ele continuará verificando todos os pares de elementos adjacentes. No pior caso, o algoritmo realiza o número máximo de trocas e comparações, pois a entrada está totalmente desordenada. No médio caso, com uma entrada aleatória, as trocas podem ser menores, pois os elementos tendem a se organizar ao longo das passagens.

Em resumo, embora o tempo de execução do Bubble Sort seja sempre $O(n^2)$ em termos de complexidade assintótica, a quantidade de trabalho real pode variar dependendo da ordem da entrada, ele irá percorrer todo o arranjo em todos os casos, porém a quantidade de comparações e trocas de elementos irá variar, resultando em tempos de execução diferentes embora a complexidade assintótica seja a mesma para todos os casos.

4.3 Selection Sort

4.3.1 Análise da ordenação por seleção

O algoritmo de ordenação Selection Sort é um algoritmo de fácil implementação, ideal para pequenos conjuntos de elementos. À medida que a entrada aumenta, o tempo de execução do algoritmo cresce , tornando-o menos eficiente para grandes conjuntos de dados.

O funcionamento do Selection Sort baseia-se em percorrer repetidamente o arranjo para encontrar o menor elemento na parte não ordenada e colocá-lo na posição correta. Esse processo se repete até que todos os elementos estejam ordenados, com a parte ordenada crescendo a cada iteração e a parte não ordenada diminuindo.

Na tabela abaixo, analisamos os custos para cada linha de código do Selection Sort.

Linha	**Código**	**Custo**	**Vezes**
1	`for i ← 1 até comprimento - 1`	C_1	n
2	`menor ← i`	C_2	$n-1$
3	`for j ← i + 1 até comprimento`	C_3	$\sum_{j=2}^{n} T_j$
4	`Se A[j] < A[menor] então`	C_4	$\sum_{j=2}^{n} (T_j - 1)$
5	`menor ← j`	C_5	$\sum_{j=2}^{n} (T_j - 1)$
6	`Troca(A[i], A[menor])`	C_6	$n-1$

Tabela 4: Análise de Complexidade do Selection Sort com os custos e número de vezes que cada linha do código é executada.

Iremos agora calcular o $T(n)$ para o Selection Sort.

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot \sum_{j=2}^{n} T_j + c_4 \cdot \sum_{j=2}^{n} (T_j - 1) + c_5 \cdot \sum_{j=2}^{n} (T_j - 1) + c_6 \cdot (n-1)$$

Sabendo que:

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

e

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

Substituindo na fórmula:

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot \left(\frac{n \cdot (n+1)}{2} - 1 \right) + c_4 \cdot \frac{n \cdot (n-1)}{2} + c_5 \cdot \frac{n \cdot (n-1)}{2} + c_6 \cdot (n-1) =$$

$$c_1 \cdot n + c_2 \cdot n - c_2 + \frac{c_3 \cdot n^2}{2} + \frac{c_3 \cdot n}{2} - c_3 + \frac{c_4 \cdot n^2}{2} - \frac{c_4 \cdot n}{2} + \frac{c_5 \cdot n^2}{2} - \frac{c_5 \cdot n}{2} + c_6 \cdot n - c_6 =$$

$$\left(\frac{c_3 + c_4 + c_5}{2} \right) \cdot n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6 \right) \cdot n + (-c_2 - c_3 - c_6)$$

Esse tempo de execução pode ser expresso como $a \cdot n^2 + b \cdot n + c$ para constantes a, b e c que dependem dos custos de instrução de c_i, logo é uma **função quadrática** de **n**.

Assim como no algoritmo Bubble Sort, o Selection Sort, também possui o tempo de execução para o **melhor caso**, **pior caso** e **médio caso** sendo expressos por uma **função quadrática**, embora o Selection Sort tenha complexidade assintótica $O(n^2)$ em todos os casos, o tempo de execução pode variar conforme a entrada:

- **Melhor Caso** (Arranjo já ordenado):

  O Selection Sort ainda realiza o mesmo número de comparações, mas o número de trocas será mínimo (no máximo uma por iteração). Embora a complexidade continue sendo $O(n^2)$, o tempo real de execução será mais rápido, pois há menos operações de troca (que também consomem tempo).

- **Médio Caso** (Arranjo em ordem aleatória):

  Para um arranjo com uma ordem intermediária (parcialmente ordenado ou aleatório), haverá um número variável de trocas, mas ainda menor do que no pior caso. Em média, o algoritmo terá um desempenho um pouco mais rápido que no pior caso, mas a diferença será pequena.

- **Pior Caso** (Arranjo em ordem inversa):

  No pior caso, o algoritmo precisará fazer a maioria das trocas possíveis. Ele realizará $n-1$ trocas no total, uma para cada elemento, resultando no maior tempo de execução dentro de $O(n^2)$.

Em resumo, enquanto o **número de comparações** é fixo e mantém a complexidade quadrática para todos os casos, o **número de trocas** é o que faz o tempo de execução real variar entre os casos, com o melhor caso sendo o mais rápido e o pior caso o mais lento.

4.4 Shell Sort

4.4.1 Análise da ordenação por concha

A complexidade do **Shell Sort** ainda não foi provada de forma definitiva devido à grande dependência da sequência de incrementos utilizada no algoritmo. Diferentes sequências de gaps podem resultar em comportamentos de tempo de execução muito distintos, o que torna a análise assintótica do algoritmo complexa e dependente dessa escolha.

Originalmente, Shell propôs uma sequência de incrementos simples, onde o valor de gap é reduzido pela metade em cada iteração, passando por valores como $(\frac{n}{2}, \frac{n}{4}, \frac{n}{8}), \ldots$. No entanto, esta escolha de sequência leva a uma complexidade no pior caso de $O(n^2)$, o que torna o algoritmo ineficiente para grandes entradas de dados.

Com o tempo, outras sequências de incrementos foram propostas, e a complexidade do algoritmo foi sendo aprimorada. Abaixo, apresentamos as análises para as principais sequências de incremento conhecidas.

4.4.2 Sequência Original de Shell

A sequência proposta por Shell consiste em começar com um gap de $\frac{n}{2}$ e, em seguida, dividir o valor de gap por 2 a cada iteração até que o valor de gap seja igual a 1. Esta abordagem leva a uma complexidade de $O(n^2)$ no pior caso, devido ao fato de que, em cada passagem, o algoritmo realiza uma série de comparações e trocas.

4.4.3 Sequência de Hibbard

Uma sequência mais eficiente foi proposta por Hibbard em 1963, que utiliza gaps da forma $2^k - 1$, resultando em valores como $1, 3, 7, 15, 31, \ldots$. A análise assintótica para o Shell Sort usando essa sequência mostra que a complexidade no pior caso é $O(n^{\frac{3}{2}})$, o que representa uma melhoria significativa em relação à sequência original de Shell. Isso ocorre porque a escolha de gaps maiores permite que os elementos sejam ordenados de maneira mais eficaz, reduzindo o número total de comparações e trocas necessárias.

4.4.4 Sequência de Sedgewick

Uma sequência ainda mais eficiente foi proposta por Robert Sedgewick em 1982. A sequência de Sedgewick é composta por valores de gaps gerados pela fórmula $(4^k + 3 \cdot 2^k + 1)$, o que gera gaps como $1, 5, 19, 41, 109, \ldots$. Com esta sequência, a complexidade do Shell Sort no pior caso é reduzida para $O(n \log^2 n)$, o que representa uma melhoria considerável em relação às sequências anteriores. A escolha de gaps da sequência de Sedgewick proporciona um equilíbrio entre a redução de distâncias entre os elementos e a redução de comparações, resultando em um algoritmo mais rápido para grandes conjuntos de dados.

4.4.5 Análise Comparativa das Sequências de Incremento

A tabela abaixo resume a análise assintótica do Shell Sort para as principais sequências de incremento:

Sequência de Incremento	Complexidade de Tempo no Pior Caso
Sequência Original de Shell	$O(n^2)$
Sequência de Hibbard	$O(n^{\frac{3}{2}})$
Sequência de Sedgewick	$O(n \log^2 n)$

Como mostrado na tabela, a escolha da sequência de incrementos tem um impacto direto na complexidade do algoritmo. A sequência de Sedgewick é a mais eficiente, alcançando uma complexidade de $O(n \log^2 n)$ no pior caso, enquanto a sequência original de Shell apresenta uma complexidade quadrática de $O(n^2)$.

4.5 Merge Sort

4.5.1 Análise da ordenação por mistura

O Merge Sort se destaca pela sua eficiência na ordenação de grandes conjuntos de dados devido à sua estrutura baseada no paradigma de divisão e conquista, que divide recursivamente o conjunto de dados em subproblemas menores até que se tornem triviais, e combina as soluções parciais de forma ordenada. Essa abordagem garante que o algoritmo consiga lidar de maneira eficiente com grandes conjuntos de dados.

Quando um algoritmo possui uma chamada recursiva, ou seja chama a si próprio, seu tempo de execução pode ser descrito por uma **equação de recorrência** que descreve seu tempo de execução **global** sobre um problema de tamanho **n** em termos do tempo de execução de entradas menores. De acordo com a Tabela 5, apresentamos uma análise detalhada das operações realizadas pelo algoritmo Merge Sort. Cada passo do algoritmo é descrito em termos de sua operação, custo constante e complexidade de tempo associada. Essa tabela facilita a compreensão do comportamento do Merge Sort, destacando os custos computacionais em cada etapa do processo. O cálculo da complexidade global $O(n \log n)$ pode ser deduzido a partir dessa análise, considerando que o algoritmo segue o paradigma "dividir para conquistar".

Passo	**Operação**	**Custo (Constante)**	**Complexidade de Tempo**
Passo 1	$if\ (p \leftarrow r)$	C_1	$O(1)$
Passo 2	$int\ q = (p + r)/2$	C_2	$O(1)$
Passo 3	$merge\_sort(vet, p, q)$	C_3	$O(n/2)$
Passo 4	$merge\_sort(vet, q, r)$	C_4	$O(n/2)$
Passo 5	$merge(vet, p, q, r)$	C_5	$O(n)$

Tabela 5: Tabela de Análise do Merge Sort

Assim a partir desta tabela podemos definir a seguinte relação de recorrência:

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

Iremos utilizar o **Teorema mestre** para provar a complexidade do Merge Sort.

Onde:

- $a = 2$ (número de subproblemas),
- $b = 2$ (fator de divisão do problema),
- $d = 1$ (grau do termo $O(n^d)$).

Para aplicar o Teorema Mestre, calculamos o valor de $\log_b a$:

$$\log_b a = \log_2 2 = 1$$

O Teorema Mestre possui três casos:

- **Caso 1:** Se $\log_b a > d$, então $T(n) = O(n^{\log_b a})$.
- **Caso 2:** Se $\log_b a = d$, então $T(n) = O(n^d \log n)$.
- **Caso 3:** Se $\log_b a < d$, então $T(n) = O(n^d)$.

No nosso caso, temos $\log_b a = 1$ e $d = 1$, ou seja, $\log_b a = d$, o que nos coloca no **Caso 2**. Assim, a complexidade de tempo total é:

$$T(n) = O(n^d \log n) = O(n \log n)$$

Portanto, a complexidade do algoritmo Merge Sort é $O(n \log n)$.

O Merge Sort possui a mesma complexidade para todos os casos, $O(n \log n)$, independente do arranjo estar ordenado ou não. Seu funcionamento segue um padrão fixo, realizando as mesmas etapas de divisão e conquista, independente da entrada, o que o torna menos eficiente em comparação com outros algoritmos de ordenação que podem adaptar seu comportamento ao tipo de entrada. Contudo, a grande vantagem do Merge Sort é que sua complexidade é sempre previsível, garantindo sempre $O(n \log n)$, independentemente da entrada.

4.6 Quick Sort

4.6.1 Análise da ordenação rápida

O Quick Sort é um dos algoritmos de ordenação mais eficientes na prática devido à sua estrutura baseada no paradigma de divisão e conquista. Ele funciona dividindo recursivamente o conjunto de dados em subproblemas menores com base em um elemento pivô, ordenando os elementos em torno do pivô, e combinando as soluções parciais de forma eficiente. Essa abordagem torna o Quick Sort muito utilizado em situações práticas, apesar de sua complexidade no pior caso ser maior do que a do Merge Sort.

O procedimento de particionamento do Quick Sort tem tempo linear $O(n)$ porque percorre os n elementos do array uma única vez. Durante essa varredura, cada elemento é comparado com o pivô e, possivelmente, trocado de posição. Como o número de operações realizadas é proporcional ao tamanho da entrada e não depende da ordem inicial dos elementos, o particionamento é $O(n)$.

Quando um algoritmo possui chamadas recursivas, seu tempo de execução pode ser descrito por uma **equação de recorrência**, que modela o tempo de execução **global** para um problema de tamanho n em termos do tempo de execução de entradas menores. A seguir, apresentamos uma análise detalhada das operações realizadas pelo Quick Sort. Cada passo do algoritmo é descrito em termos de sua operação, custo constante e complexidade de tempo associada. A tabela 6 facilita a compreensão do comportamento do Quick Sort, destacando os custos computacionais em cada etapa do processo.

Passo	Operação	Custo (Constante)	Complexidade de Tempo
Passo 1	`if (p ← r - 1)`	C_1	$O(1)$
Passo 2	`int q = particiona(a, p, r)`	C_2	$O(n)$
Passo 3	`quick_sort(a, p, q - 1)`	C_3	$O(n/2)$
Passo 4	`quick_sort(a, q + 1, r)`	C_4	$O(n/2)$

Tabela 6: Tabela de Análise do Quick Sort

4.6.2 Melhor caso

Assim a partir desta tabela podemos definir a seguinte relação de recorrência:

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

Iremos utilizar o **Teorema mestre** para provar a complexidade do melhor caso do Quick Sort.

Onde:

- $a = 2$ (número de subproblemas),
- $b = 2$ (fator de divisão do problema),
- $d = 1$ (grau do termo $O(n^d)$).

Para aplicar o Teorema Mestre, calculamos o valor de $\log_b a$:

$$\log_b a = \log_2 2 = 1$$

O Teorema Mestre possui três casos:

- **Caso 1:** Se $\log_b a > d$, então $T(n) = O(n^{\log_b a})$.
- **Caso 2:** Se $\log_b a = d$, então $T(n) = O(n^d \log n)$.
- **Caso 3:** Se $\log_b a < d$, então $T(n) = O(n^d)$.

No nosso caso, temos $\log_b a = 1$ e $d = 1$, ou seja, $\log_b a = d$, o que nos coloca no **Caso 2**. Assim, a complexidade de tempo total é:

$$T(n) = O(n^d \log n) = O(n \log n)$$

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

4.6.3 Pior caso

O comportamento do pior caso para o Quick Sort ocorre quando a rotina de particionamento produz um subproblema com $(n - 1)$ elementos e um com 0 elementos. Vamos supor

que esse particionamento não balanceado surge em cada chamada recursiva. O particionamento custa o tempo $O(n)$. Tendo em vista que a chamada recursiva sobre um arranjo de tamanho 0 simplesmente retorna, $T(0) = 1$, a recorrência para o tempo de execução é:

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

Iremos utilizar o **método da expansão** para provar a complexidade do pior caso do Quick Sort.

$$\begin{aligned} T(K) &= T(K-1) + K \\ &= T(K-2) + (K-1) + K \\ &= T(K-3) + (K-2) + (K-1) + K \\ &= T(K-i) + \cdots + (K-2) + (K-1) + K \end{aligned}$$

Temos que:

$$K - i = 1$$

$$i = K - 1$$

Substituindo:

$$\begin{aligned} T(K) &= T(K-(K-1)) + \cdots + (K-2) + (K-1) + K \\ &= 1 + (K-2) + (K-1) + K \end{aligned}$$

Assim temos uma sequência de P.A:

$$S_n = n \cdot \frac{(a_1 + a_n)}{2}$$

Substituindo:

$$\frac{(1+K) \cdot K}{2} = \frac{K + K^2}{2}$$

Temos a fórmula FF:

$$\frac{n + n^2}{2}$$

Assim temos a complexidade $O(n^2)$, e iremos realizar a prova por indução: Caso base:

$$P(1) = T(1) = \frac{1 + 1^2}{1} = \frac{2}{2} = 1$$

Hipótese:

$$P(K) = T(K) = \frac{K + K^2}{2}$$

Prova:

$$P(K+1) = T(K+1) = \frac{(K+1) + (K+1)^2}{2} = \frac{(K+1) + (K^2 + 2K + 1)}{2} = \frac{K^2 + 3K + 2}{2}$$

$$T(K+1-1) + (K+1) = \frac{K + K^2}{2} + (K+1) = \frac{K + K^2 + 2K + 2}{2} = \frac{K^2 + 3K + 2}{2}$$

Portanto, a complexidade do algoritmo Quick Sort no seu pior caso é $O(n^2)$.

O Quick Sort, apesar de apresentar uma complexidade $O(n^2)$ no pior caso, destaca-se por sua eficiência prática e complexidade $O(n \log n)$ no melhor e caso médio, o que o torna amplamente utilizado em diversas aplicações. Sua abordagem baseada no paradigma de divisão e conquista, faz com que seja um dos algoritmos de ordenação mais populares.

Contudo, é importante considerar o comportamento da rotina de particionamento, que pode impactar o desempenho dependendo da escolha do pivô e do arranjo inicial dos dados.

4.7 Heap Sort

4.7.1 Análise da ordenação por heap

O Heap Sort é um algoritmo de ordenação baseado na estrutura de dados chamada heap binário, que garante uma relação de ordem entre os elementos, facilitando a extração do maior ou menor elemento em **tempo constante**. A eficiência do Heap Sort decorre de duas etapas principais: a **construção do heap** e a **ordenação** por meio da repetida extração do elemento raiz. Para validar seu funcionamento e eficiência, é necessário provar o tempo de execução da construção do heap, que será feito abaixo. Utilizaremos a construção de um heap mínimo como exemplo.

Algorithm 1 MIN-HEAPIFY

```
1:  function MIN-HEAPIFY(A, i)
2:      L ← LEFT(i)
3:      R ← RIGHT(i)
4:      if L ≤ tamanho-do-heap[A] and A[L] < A[i] then
5:          menor ← L
6:      else
7:          menor ← i
8:      end if
9:      if R ≤ tamanho-do-heap[A] and A[R] < A[menor] then
10:         menor ← R
11:     end if
12:     if menor ≠ i then
13:         Trocar A[i] ↔ A[menor]
14:         MIN-HEAPIFY(A, menor)
15:     end if
16: end function
```

Basicamente o pseudocódigo acima descreve a manipulação de um heap mínimo. Suas entradas é o arranjo no qual será construído o Heap, e o seu índice. Quando está função é chamada, supomos que as árvores com raízes em LEFT(i) e RIGHT(i) são heaps mínimos, mas A[i] pode ser maior do que seus filhos, violando a propriedade do heap máximo. A função deixa os valores em A[i] no heap mínimo. O tempo de execução da função MIN-HEAPIFY em uma sub-árvore de tamanho **n** com raiz em um dado no **i** é o tempo $\theta(1)$ para corrigir os relacionamentos entre os elementos A[i], A[LEFT(i)] e A[RIGHT(i)], mais o tempo para executar MIN-HEAPIFY em uma sub-árvore. Cada sub-árvore tem tamanho $\frac{2 \cdot n}{3}$ o pior caso

ocorre quando a última linha da árvore está metade cheia. O tempo de execução da função MIN-HEAPIFY pode ser descrito pela seguinte relação de recorrência:

$$T(n) = T\left(\frac{2 \cdot n}{3}\right) + \Theta(1)$$

vamos resolvê-la utilizando o Teorema Mestre.

A relação de recorrência é da forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

onde:

- $a = 1$,
- $b = \frac{3}{2}$ (ou seja, $\log_b n = \log_{3/2} n$),
- $f(n) = \Theta(1)$.

O Teorema Mestre nos diz para comparar $f(n)$ com $n^{\log_b a}$:

$$\log_b a = \log_{3/2} 1 = 0.$$

Agora, comparamos $f(n) = \Theta(1)$ com $n^0 = \Theta(1)$. Como $f(n)$ está no mesmo crescimento assintótico que $n^{\log_b a}$, estamos no caso 2 do Teorema Mestre.

Portanto, a solução da recorrência é:

$$T(n) = O(\log n).$$

Agora iremos de fato construir um heap, utilizando a função BUILD-MIN-HEAP na qual é representada abaixo:

Algorithm 2 BUILD-MIN-HEAP

```
1: function BUILD-MIN-HEAP(A)
2:     tamanho-do-heap[A] ← comprimento[A]
3:     for i ← ⌊comprimento[A]/2⌋ downto 1 do
4:         MIN-HEAPIFY(A, i)
5:     end for
6: end function
```

O código BUILD-MIN-HEAP transforma um arranjo em um heap mínimo. Cada chamada a MIN-HEAPIFY custa o tempo $O(\log n)$, existem $O(n)$ dessas chamadas. Somando esses custos podemos utilizar as propriedades do **Big O** para isto, logo:

$$O(f(n) + O(g)) = O(max(f(n), g(n))$$

$$O(max(f(n), g(n)) = O(max(n, \log n)$$

$$O(n)$$

Agora, partimos para o HEAP-SORT, que é onde a ordenação é basicamente feita, primeiramente ele constrói o Heap e por último faz uma chamada ao MIN-HEAPIFY. E então o Heap sort repete esse processo para um heap de tamanho $(n - 1)$, segue o pseudo-código abaixo.

Algorithm 3 HEAP-SORT

```
1: function HEAP-SORT(A)
2:     BUILD-MIN-HEAP(A)
3:     for i ← comprimento[A] downto 2 do
4:         Trocar A[1] ↔ A[i]
5:         tamanho-do-heap[A] ← tamanho-do-heap[A] − 1
6:         MIN-HEAPIFY(A, 1)
7:     end for
8: end function
```

Devido ao loop for, há $O(n)$ chamadas a função MIN-HEAPIFY na qual está função tem tempo de execução de $O(\log n)$, assim podemos concluir que, o tempo de execução do Heap Sort é:

$$O(n \cdot \log n)$$

5 TABELA E GRÁFICO

5.1 Insertion Sort

A seguir, são apresentados os dados coletados durante os testes de desempenho do algoritmo Insertion Sort, com base em diferentes tipos de entrada: crescente, decrescente e randômica. A tabela mostra os tempos de execução em segundos para diferentes tamanhos de entrada, variando de 10 a 1.000.000 elementos. Esses dados foram obtidos para avaliar como o desempenho do algoritmo muda de acordo com o tipo de entrada e o tamanho dos dados a serem ordenados.

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.2670	0.0480
100000	0.0000	21.4750	9.7210
1000000	0.0160	1839.450	796.112

Tabela 7: Tabela de tempo por segundo do algoritmo Insertion Sort feito pelo autor

Com os dados coletados, podemos concluir que o algoritmo Insertion Sort tem um desempenho satisfatório para conjuntos pequenos de dados, mas seu tempo de execução aumenta significativamente à medida que o tamanho da entrada cresce, especialmente em entradas decrescentes, que são o pior caso para este algoritmo. Isso ocorre devido ao grande número de comparações necessárias, tornando-o ineficiente para grandes volumes de dados.

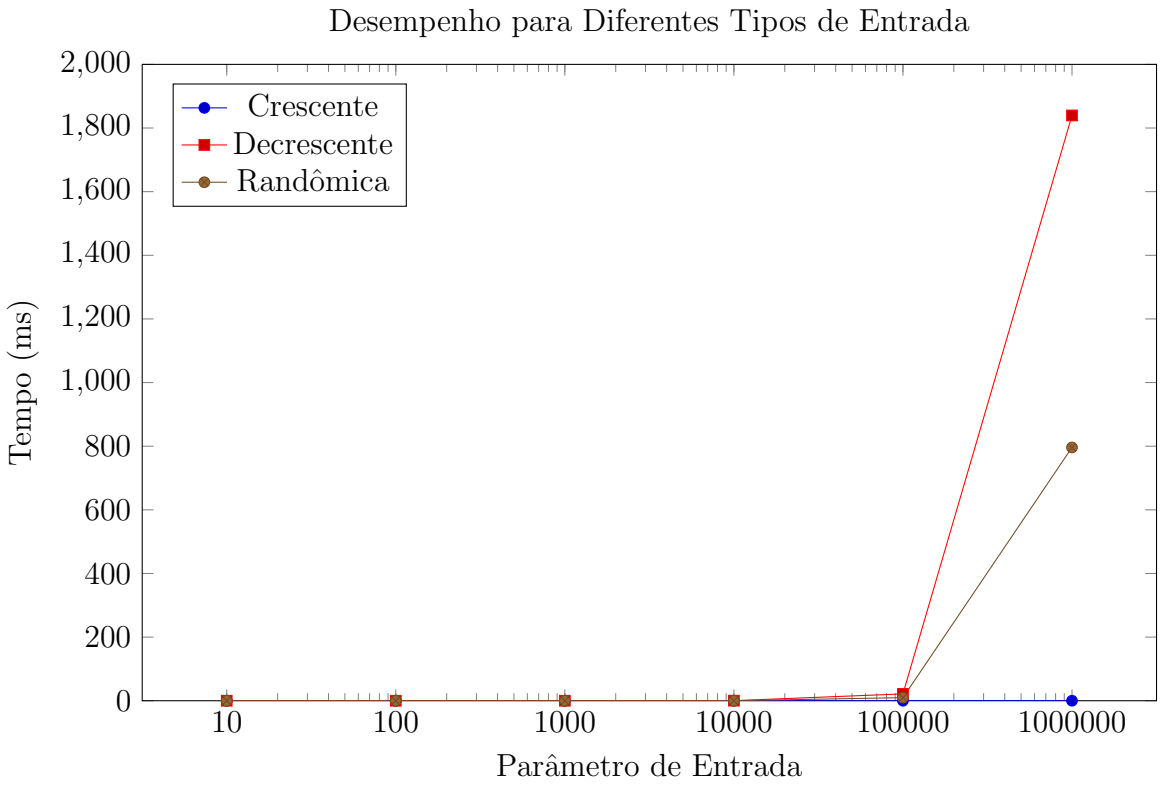

Figura 20: Gráfico de tempo por segundo do algoritmo Insertion Sort feito pelo autor

5.2 Bubble Sort

A seguir, são apresentados os resultados de desempenho do algoritmo Bubble Sort para diferentes tipos de entrada: crescente, decrescente e randômica. A tabela abaixo exibe os tempos de execução em segundos para entradas com tamanhos variando de 10 a 1.000.000 elementos. Os dados coletados visam analisar como o desempenho do algoritmo se comporta em diferentes cenários e com variações no tamanho da entrada.

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0160
10000	0.0940	0.3750	0.2340
100000	9.3710	23.634	30.940
1000000	959.49	3666.587	3991.286

Tabela 8: Tabela de tempo por segundo do algoritmo Bubble Sort feito pelo autor

Com os dados coletados, podemos concluir que o Bubble Sort funciona de maneira simples

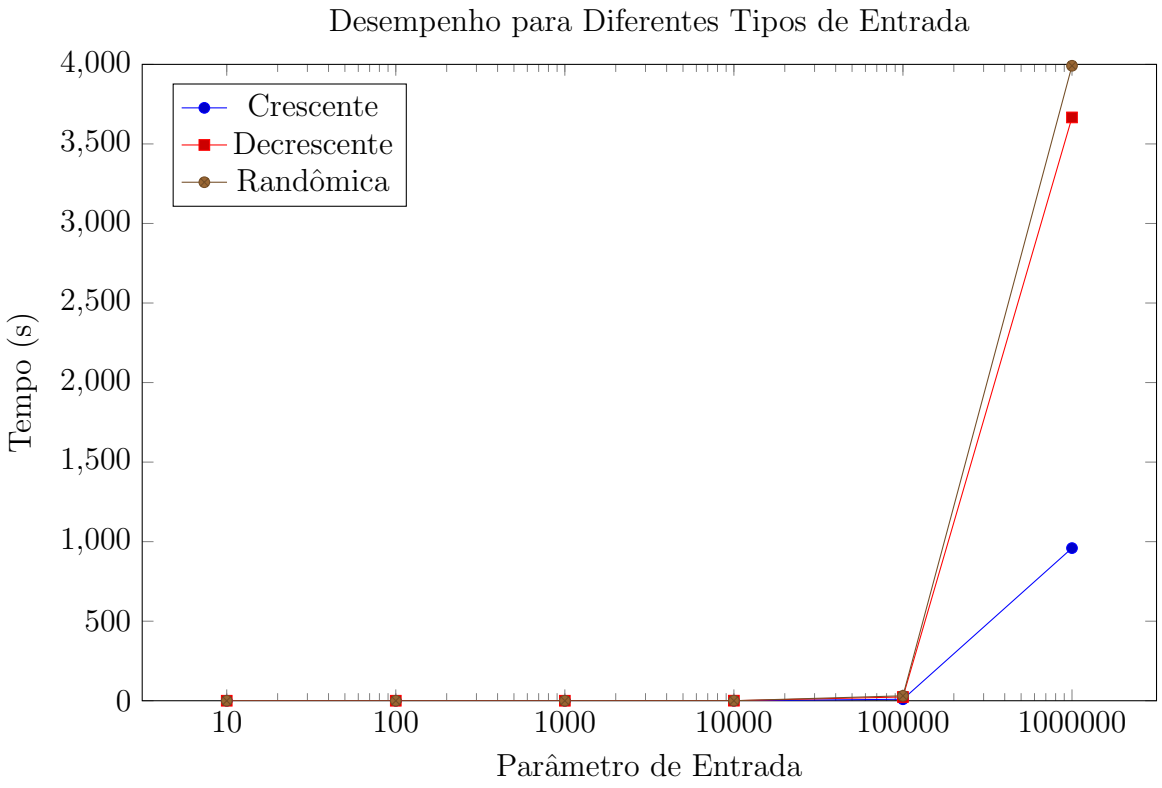

Figura 21: Gráfico de tempo por segundo do algoritmo Bubble Sort feito pelo autor

e direta, mas sua eficiência diminui conforme o tamanho da entrada aumenta. Pelo fato deste algoritmo percorrer todo o arranjo e realizar muitas comparações desnecessárias, se torna ineficiente para grandes conjuntos de dados.

5.3 Selection Sort

A tabela e o gráfico a seguir apresentam os tempos de execução do algoritmo Selection Sort para diferentes tipos de entrada: crescente, decrescente e randômica. Os dados coletados mostram o tempo em segundos para entradas com tamanhos que variam de 10 a 1.000.000 elementos. O objetivo é observar como o algoritmo se comporta em diferentes cenários de entrada e com o aumento da quantidade de dados. A análise desses resultados é importante para avaliar a eficiência do algoritmo em diferentes contextos.

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.1090	0.1090	0.1100
100000	10.6340	9.6190	10.2510
1000000	1075.644	981.562	3148.752

Tabela 9: Tabela de tempo por segundo do algoritmo Selection Sort feito pelo autor

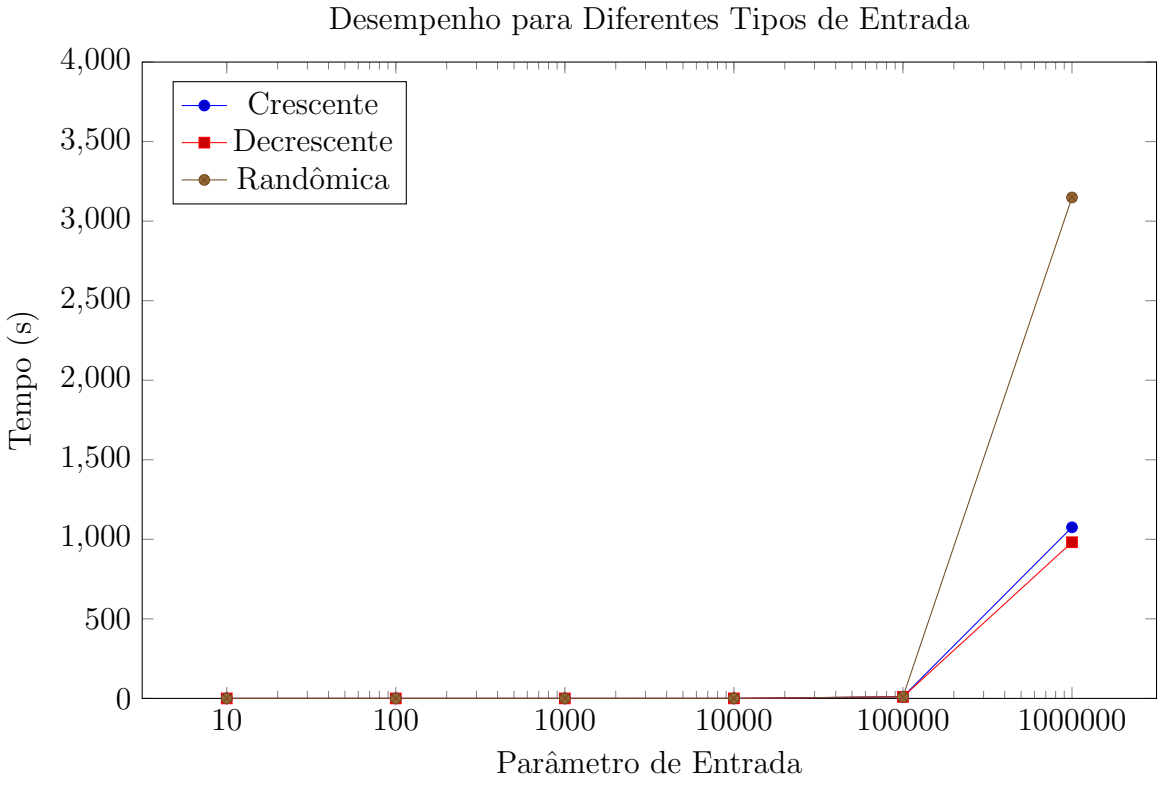

Figura 22: Gráfico de tempo por segundo do algoritmo Selection Sort feito pelo autor

Com os dados coletados, podemos concluir que o Selection Sort é um algoritmo relativamente simples de implementar, mas sua eficiência é limitada em conjuntos de dados maiores. Ele percorre repetidamente o arranjo em busca do menor elemento e realiza várias operações de troca, mesmo que o arranjo já esteja parcialmente ordenado. Isso faz com que seu tempo de execução cresça rapidamente conforme a entrada aumenta, tornando-o mais indicado para pequenos conjuntos de dados.

5.4 Shell Sort

A tabela e o gráfico apresentados a seguir ilustram os tempos de execução do algoritmo Shell Sort para diferentes tipos de entrada: crescente, decrescente e randômica. Os dados coletados mostram o tempo em segundos para entradas com tamanhos variando de 10 a 1.000.000 elementos. O objetivo deste experimento é avaliar o desempenho do Shell Sort em cenários variados e com a alteração do tamanho da entrada, fornecendo uma visão geral da sua eficiência em diferentes tipos de dados.

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0000	0.0000
100000	0.0000	0.0160	0.0310
1000000	0.0460	0.0780	0.2660

Tabela 10: Tabela de tempo por segundo do algoritmo Shell Sort feito pelo autor

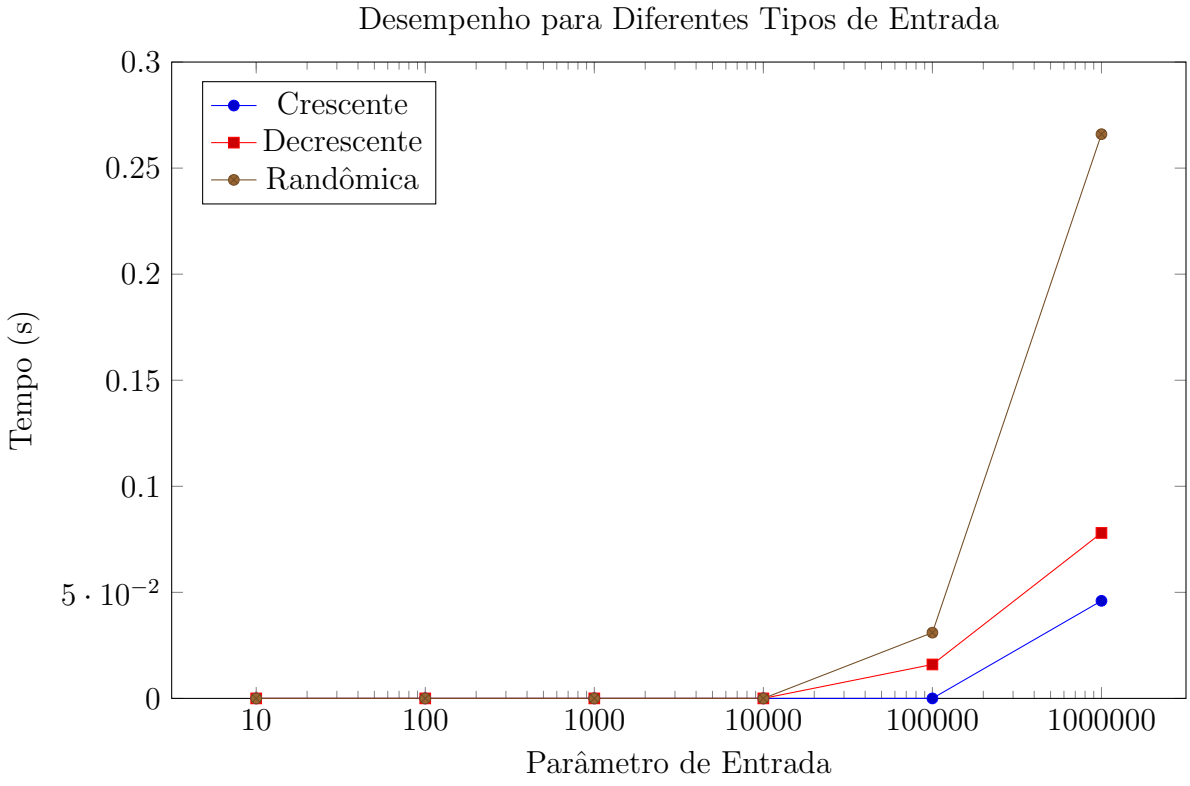

Figura 23: Gráfico de tempo por segundo do algoritmo Shell Sort feito pelo autor

Embora a complexidade do Shell Sort tenha sido amplamente estudada e melhorada com o tempo, ela ainda não foi provada de forma definitiva para todas as possíveis sequências de incremento. O algoritmo pode atingir uma complexidade de $O(n^2)$ no pior caso para a sequência original de Shell, mas sequências mais sofisticadas, como as de Hibbard e Sedgewick, podem reduzir a complexidade para $O(n^{\frac{3}{2}})$ e $O(n \log^2 n)$, respectivamente. Isso torna o Shell Sort um algoritmo eficiente para entradas de tamanho médio, mas ainda inferior a algoritmos como Merge Sort e Quick Sort para entradas grandes.

5.5 Merge Sort

A tabela e o gráfico apresentados a seguir mostram os tempos de execução do algoritmo Merge Sort para diferentes tipos de entrada: crescente, decrescente e randômica. Os dados foram coletados para entradas de tamanho variado, desde 10 até 1.000.000 elementos, com o objetivo de avaliar a performance do algoritmo em diferentes cenários e com entradas de diferentes tamanhos. O Merge Sort é conhecido por sua complexidade de tempo $O(n \log n)$, o que garante uma boa eficiência, especialmente em grandes volumes de dados.

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0000	0.0010
100000	0.0160	0.0160	0.0320
1000000	0.1430	0.2380	0.2230

Tabela 11: Tabela de tempo por segundo do algoritmo Merge Sort feito pelo autor

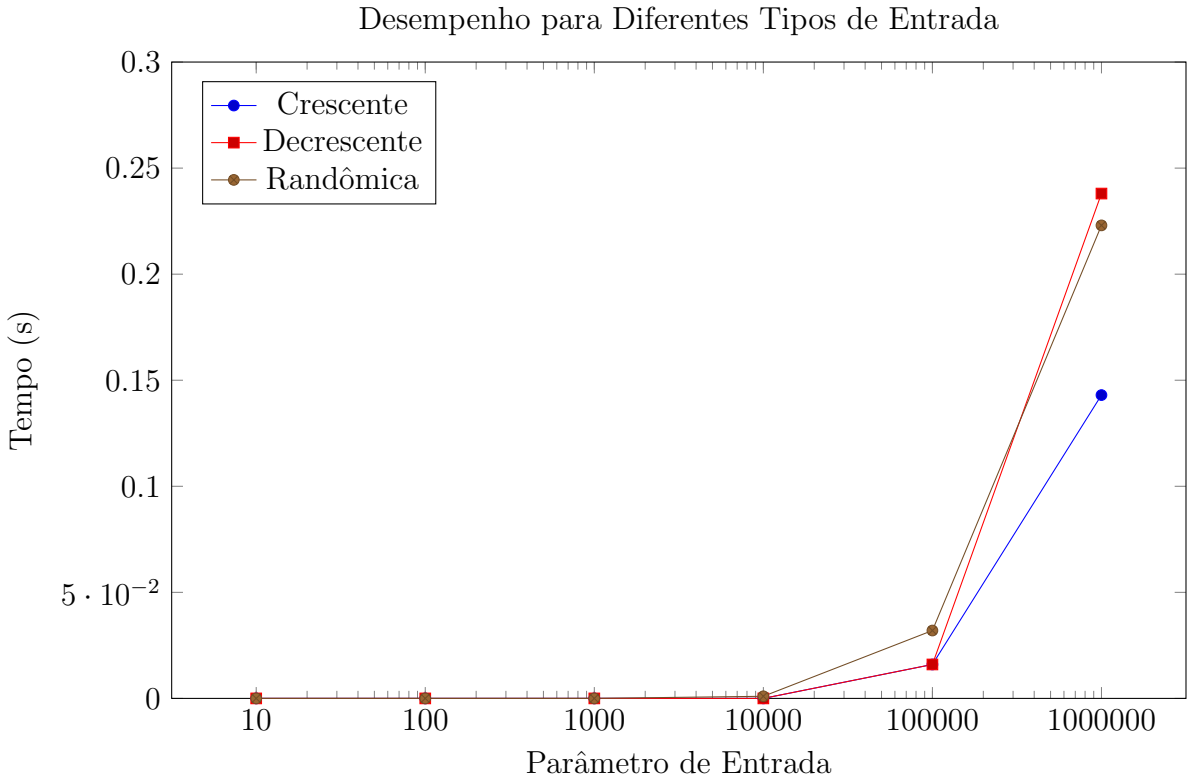

Figura 24: Gráfico de tempo por segundo do algoritmo Merge Sort feito pelo autor

Com os dados analisados, podemos concluir que o Merge Sort é um algoritmo de ordenação eficiente e com uma complexidade de tempo previsível de $O(n \log n)$, para todos os casos. Sua principal vantagem está na consistência de desempenho, já que, independentemente do tipo de entrada (crescente, decrescente ou randômica), o tempo de execução será sempre o mesmo.

O algoritmo utiliza a técnica de dividir e conquistar, o que permite que ele divida o problema recursivamente em subproblemas menores e depois os combine de forma eficiente. Isso torna o Merge Sort particularmente adequado para grandes conjuntos de dados, já que sua complexidade de $O(n \log n)$ é muito mais eficiente do que algoritmos como o Selection Sort ou o Bubble Sort, que possuem complexidade $O(n^2)$.

No entanto, o Merge Sort exige espaço adicional para armazenar os subarranjos temporários durante o processo de combinação, o que pode ser um fator limitante em ambientes com recursos de memória restritos.

Além disso, uma característica importante do Merge Sort é que ele não aproveita entradas parcialmente ordenadas. Mesmo que o arranjo já esteja parcialmente ordenado, o algoritmo executará o mesmo número de operações, o que pode ser visto como uma desvantagem em relação a algoritmos como o Quick Sort, que pode se beneficiar de dados ordenados ou quase ordenados.

Em resumo, o Merge Sort é ideal para grandes conjuntos de dados, onde sua eficiência $O(n \log n)$ é melhor. No entanto, seu uso em cenários com limitações de memória ou em entradas já ordenadas pode ser menos eficiente.

5.6 Quick Sort

O algoritmo Quick Sort é utilizado devido à sua eficiência, com uma complexidade de tempo média de $O(n \log n)$. A tabela e o gráfico a seguir ilustram os tempos de execução do Quick Sort para diferentes tipos de entrada: crescente, decrescente e randômica. Os dados foram coletados para entradas de tamanhos variados, desde 10 até 1.000.000 elementos, com o objetivo de avaliar o desempenho do algoritmo em diferentes cenários.

A principal vantagem do Quick Sort é sua performance média, com a capacidade de dividir recursivamente o problema em subproblemas menores e, em seguida, resolvê-los eficientemente. Contudo, a complexidade de tempo do Quick Sort pode se degradar para $O(n^2)$ no pior caso, que ocorre quando o pivô escolhido divide os dados de maneira desequilibrada.

Além disso, o Quick Sort pode ser sensível ao tipo de entrada. Como é evidente na tabela e no gráfico, o desempenho do algoritmo varia dependendo de como os dados são fornecidos. Para entradas crescentes e decrescentes, o tempo de execução aumenta à medida que o tamanho da entrada cresce, com um desempenho ligeiramente pior para dados ordenados de forma decrescente. Já para entradas randômicas, o algoritmo mantém um desempenho consistente, o que é uma vantagem em cenários com dados não ordenados. Nesse exemplo, utilizamos o quick sort em três cenários, com o pivô sendo o primeiro elemento, o pivô sendo escolhido como a média dos elementos do arranjo e por último o pivô sendo escolhido de forma aleatória.

A seguir, apresentamos tabelas e gráficos que comparam o desempenho do algoritmo Quick Sort em diferentes cenários, considerando três estratégias de escolha do pivô: o primeiro elemento, a média dos elementos e a seleção aleatória.

5.6.1 Pivô sendo o primeiro elemento

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0040	0.0040
100000	0.0570	0.0630	0.0790
1000000	0.8750	0.9810	1.2700

Tabela 12: Tempo de execução (em segundos) do Quick Sort com o pivô sendo o primeiro elemento.

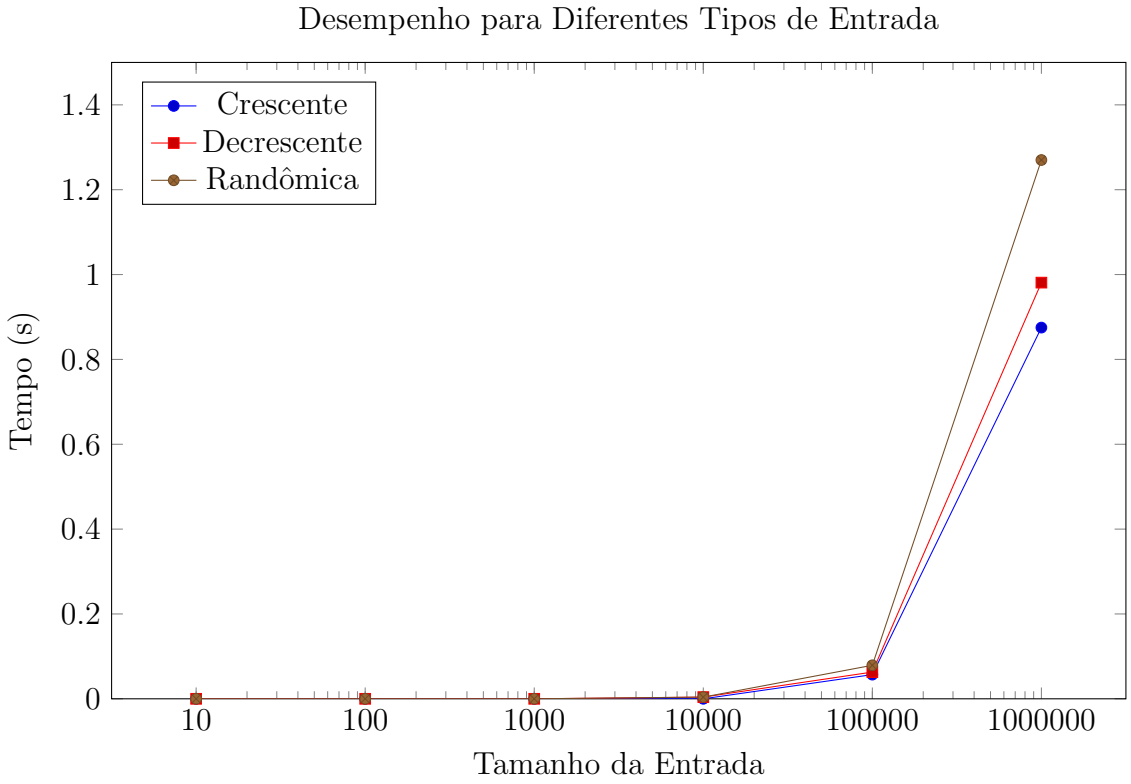

Figura 25: Gráfico de desempenho do Quick Sort com o pivô sendo o primeiro elemento.

O gráfico acima demonstra o desempenho do Quick Sort quando o pivô é sempre escolhido como o primeiro elemento do arranjo. Esse método é particularmente sensível a entradas ordenadas (crescente ou Decrescente), resultando em maior tempo de execução devido ao aumento de chamadas recursivas desbalanceadas.

5.6.2 Pivô como sendo a média dos elementos

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0000	0.0000
100000	0.0160	0.0160	0.0210
1000000	0.1750	0.1580	0.3480

Tabela 13: Tempo de execução (em segundos) do Quick Sort com o pivô sendo a média dos elementos.

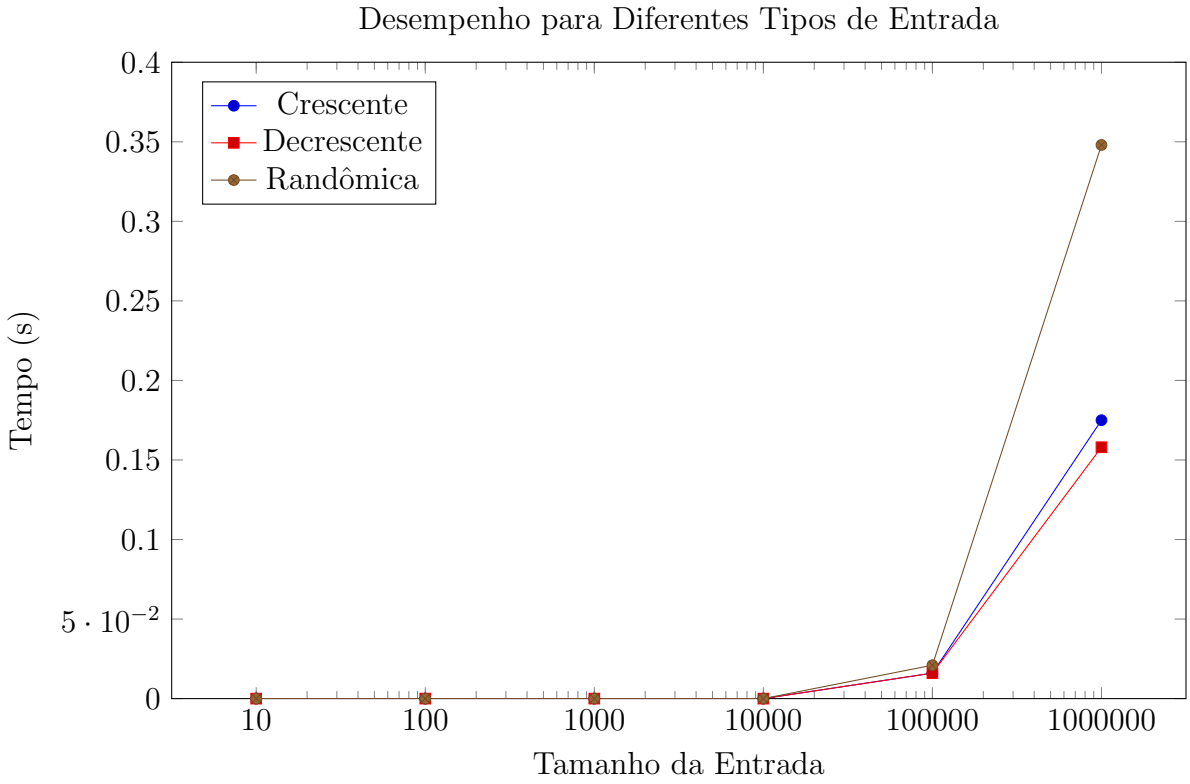

Figura 26: Gráfico de desempenho do Quick Sort com o pivô sendo a média dos elementos.

Neste caso, o Quick Sort utiliza a média dos elementos como pivô, o que tende a melhorar o balanceamento das partições. Como resultado, o desempenho é mais estável, independentemente do tipo de entrada.

5.6.3 Pivô sendo escolhido de forma aleatória

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0020	0.0020
100000	0.0170	0.0270	0.0320
1000000	0.2220	0.3000	0.3760

Tabela 14: Tempo de execução (em segundos) do Quick Sort com o pivô sendo escolhido de forma aleatória.

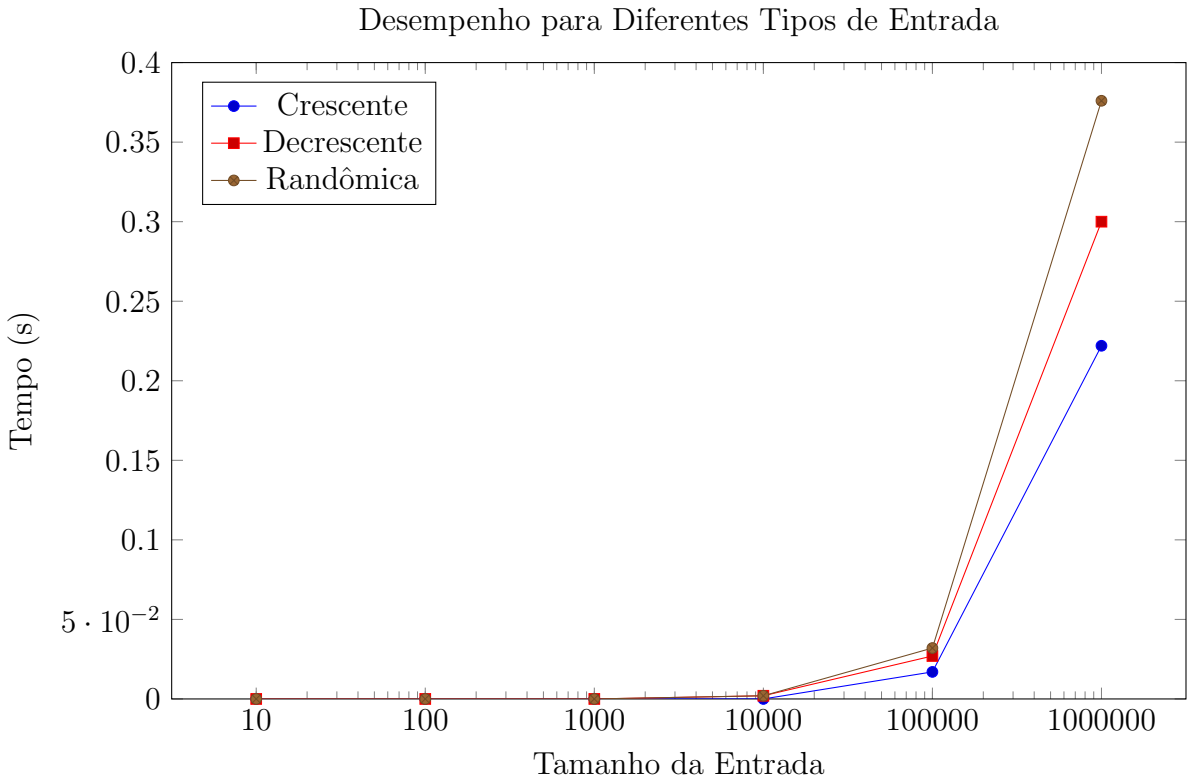

Figura 27: Gráfico de desempenho do Quick Sort com o pivô sendo escolhido de forma aleatória.

A escolha aleatória do pivô visa minimizar os piores casos ao garantir maior aleatoriedade nas partições. Este método apresenta um desempenho consistente, mesmo para entradas ordenadas.

O Quick Sort, com sua complexidade média de $O(n \log n)$, é amplamente reconhecido por sua eficiência. Contudo, o desempenho do algoritmo depende significativamente da estratégia

de escolha do pivô e da distribuição dos dados de entrada.

Os resultados demonstram que:

- Escolher o pivô como o primeiro elemento pode levar a partições desbalanceadas em entradas ordenadas, prejudicando o desempenho.
- A escolha do pivô com base na média dos elementos reduz o impacto de partições desbalanceadas, resultando em maior estabilidade.
- Selecionar o pivô de forma aleatória oferece uma abordagem robusta, especialmente para entradas randômicas, garantindo um desempenho próximo ao ideal.

Essas análises destacam a importância de selecionar a estratégia de pivô adequada para otimizar o desempenho do Quick Sort em diferentes cenários.

5.7 Heap Sort

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0000	0.0000
100000	0.0160	0.0000	0.0160
1000000	0.1600	0.1740	0.2380

Tabela 15: Tabela de tempo por segundo do algoritmo Heap Sort feito pelo autor

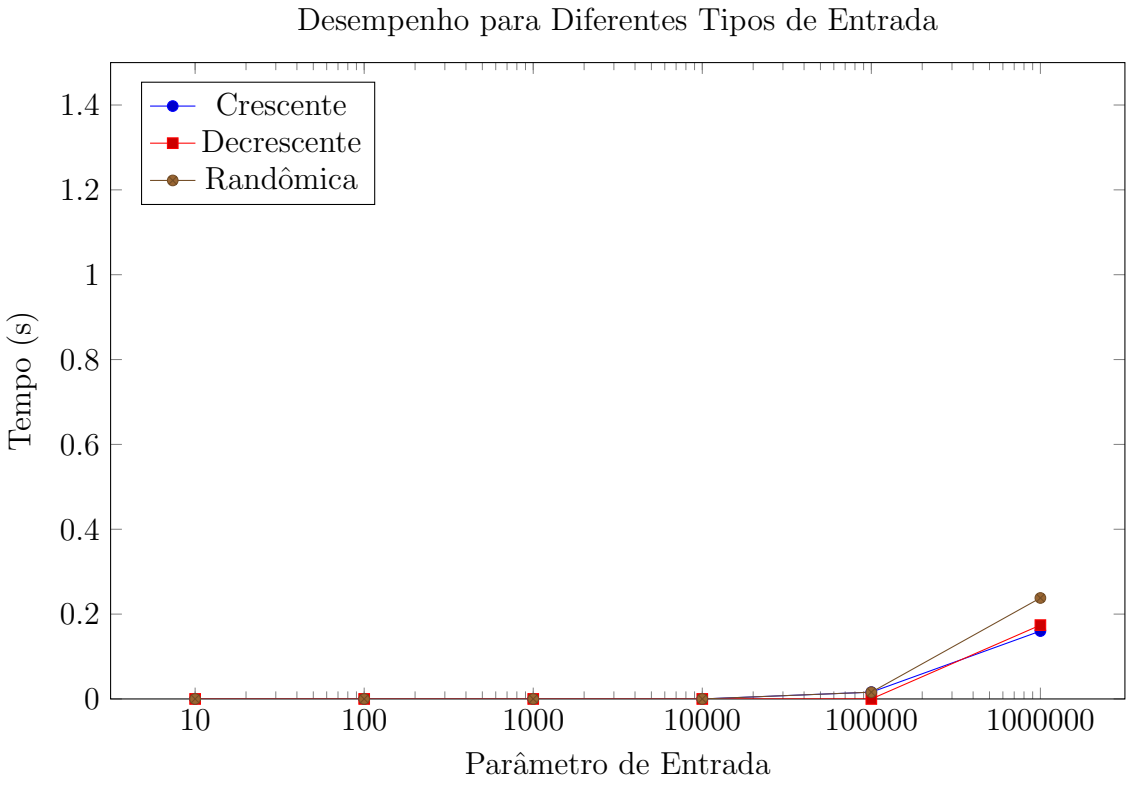

Figura 28: Gráfico de tempo por segundo do algoritmo Heap Sort feito pelo autor

5.8 Tabela comparativa e gráfico geral dos algoritmos de troca

A tabela abaixo fornece uma análise comparativa dos tempos de execução, em segundos, de quatro algoritmos de ordenação clássicos: **Insertion Sort**, **Bubble Sort**, **Selection Sort** e **Shell Sort**. Os dados foram obtidos utilizando entradas de tamanhos variados (10 a 1.000.000 elementos), organizadas em três categorias: **crescente**, **decrescente**, **randômica**.

Entrada	Insertion Sort	Bubble Sort	Selection Sort	Shell Sort
Crescente				
10	0.0000	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000	0.0000
10000	0.0000	0.0940	0.1090	0.0000
100000	0.0000	9.3710	10.6340	0.0000
1000000	0.0160	959.490	1075.644	0.0460
Decrescente				
10	0.0000	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000	0.0000
10000	0.2670	0.3750	0.1090	0.0000
100000	21.4750	23.634	9.6190	0.0160
1000000	1839.450	3666.587	981.562	0.0780
Randômica				
10	0.0000	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000	0.0000
1000	0.0000	0.0160	0.0000	0.0000
10000	0.0480	0.2340	0.1100	0.0000
100000	9.7210	30.940	10.2510	0.0310
1000000	796.112	3991.286	3148.752	0.2660

Tabela 16: Comparação dos tempos de execução dos algoritmos de ordenação para entradas crescentes, decrescentes e randômicas (em segundos).

5.8.1 Entrada crescente

A entrada crescente é uma das situações mais favoráveis para a maioria dos algoritmos de ordenação, pois muitos algoritmos, como o Insertion Sort, possuem um desempenho muito bom quando a sequência de dados já está em ordem crescente. Isso ocorre porque esses algoritmos precisam fazer poucas ou nenhuma troca. Ao observar a tabela, podemos notar que os tempos de execução para o Insertion Sort e o Shell Sort são praticamente nulos para

entradas pequenas (como 10 e 100 elementos), e o tempo de execução aumenta gradualmente conforme o tamanho da entrada aumenta.

Por outro lado, o Bubble Sort e o Selection Sort apresentam um aumento considerável no tempo de execução, especialmente quando a entrada atinge grandes dimensões, como 1.000.000 de elementos. Mesmo em uma entrada já ordenada, esses algoritmos continuam a percorrer toda a lista em busca de alterações, o que torna o tempo de execução mais elevado em comparação com algoritmos mais eficientes, como o Insertion Sort.

Portanto, para entradas em ordem crescente, algoritmos como o Insertion Sort e Shell Sort apresentam um desempenho superior, especialmente para entradas pequenas a médias, enquanto Bubble Sort e Selection Sort demonstram uma eficiência inferior, mesmo em entradas ordenadas.

De acordo com a Figura 15, apresentamos uma análise comparativa dos tempos de execução de diferentes algoritmos de ordenação Insertion Sort, Bubble Sort, Selection Sort e Shell Sort considerando entradas crescentes.

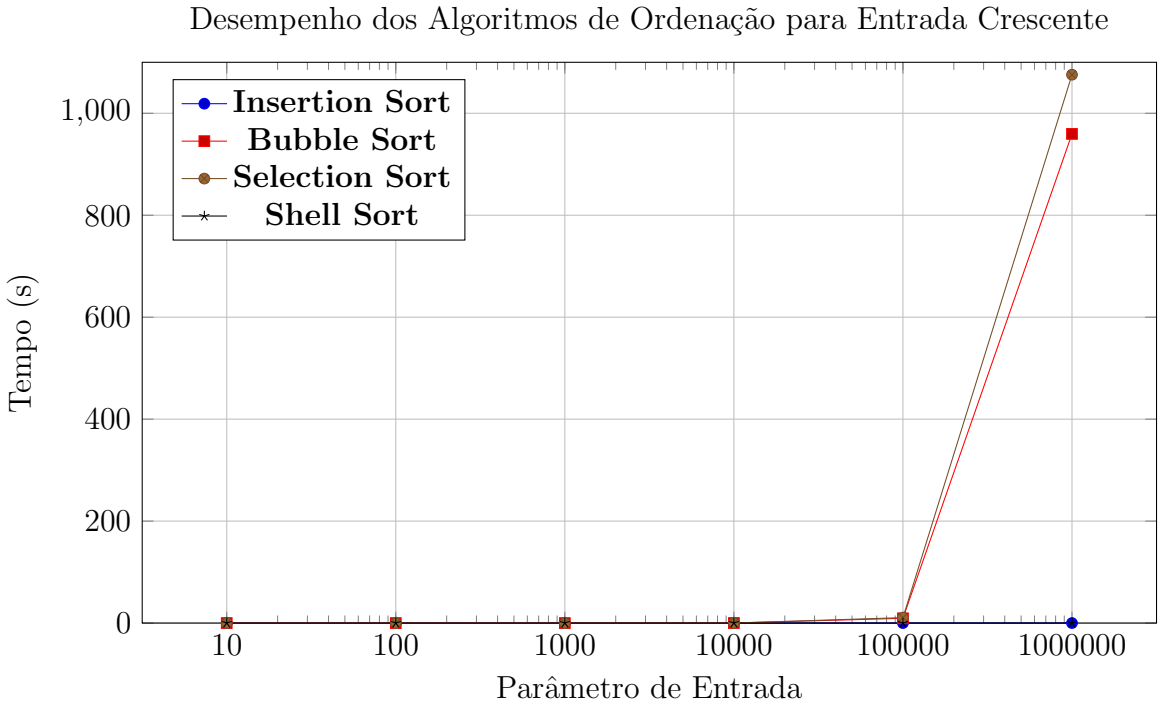

Figura 29: Comparação dos tempos de execução dos algoritmos de ordenação para entrada crescente.

5.8.2 Entrada decrescente

A entrada decrescente é uma das situações mais desafiadoras para os algoritmos de ordenação, pois exige que cada algoritmo reordene toda a sequência de dados para alcançar a ordem crescente.

Como observado na tabela, o Insertion Sort e o Shell Sort possuem tempos de execução elevados para entradas grandes, como 100.000 ou 1.000.000 de elementos. Isso ocorre nesses casos, o algoritmo precisa percorrer e mover elementos por uma grande quantidade de vezes. O Bubble Sort e o Selection Sort, por sua vez, continuam a exibir tempos elevados mesmo em entradas menores, refletindo a natureza ineficiente desses algoritmos.

Dessa forma, para entradas em ordem decrescente, algoritmos como o Bubble Sort e o Selection Sort mostram uma eficiência inferior em comparação ao Insertion Sort e o Shell Sort.

De acordo com a Figura 16, apresentamos uma análise comparativa dos tempos de execução de diferentes algoritmos de ordenação Insertion Sort, Bubble Sort, Selection Sort e Shell Sort para entradas em ordem decrescente.

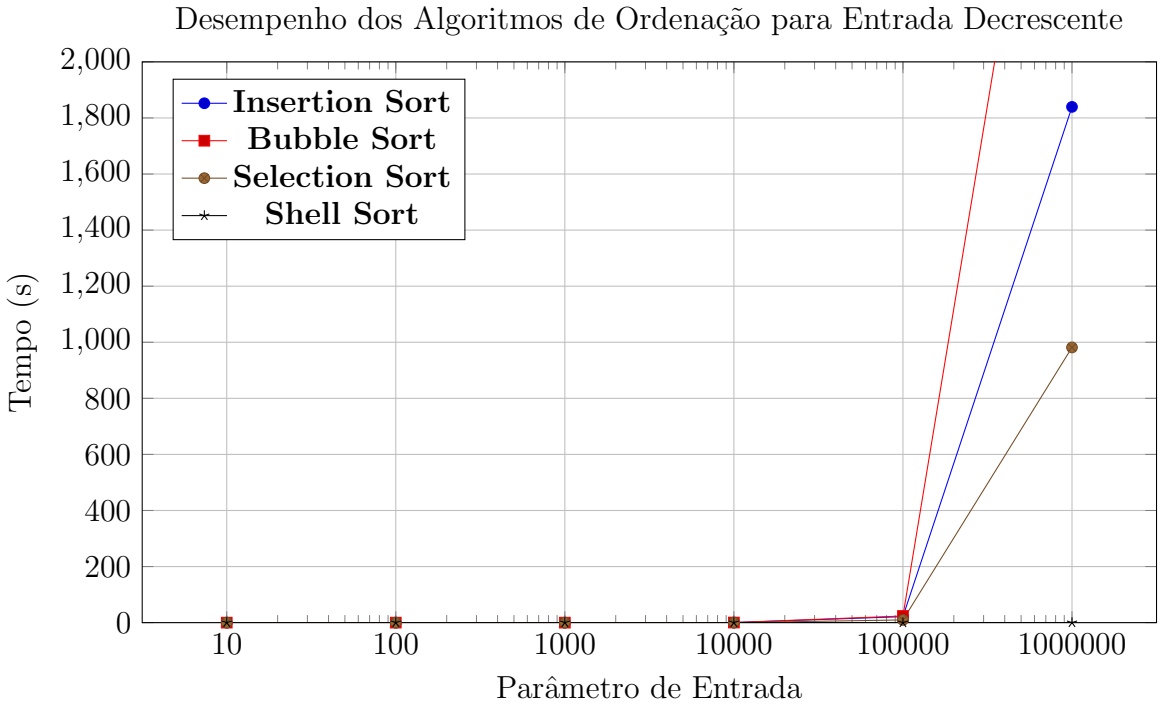

Figura 30: Comparação dos tempos de execução dos algoritmos de ordenação para entrada decrescente.

5.8.3 Entrada randômica

A entrada randômica é uma das situações mais representativas do desempenho real dos algoritmos de ordenação, já que simula dados que não seguem nenhuma ordem específica.

Ao observar os tempos de execução na tabela, vemos que o Insertion Sort apresenta um bom desempenho para entradas pequenas, mas o tempo de execução aumenta à medida que o número de elementos cresce. O Bubble Sort e o Selection Sort, ainda apresentam tempos elevados.

Por outro lado, o Shell Sort se destaca por ter tempos de execução mais baixos. Isso mostra que ele é mais eficiente em comparação com os algoritmos mais simples, portanto, o Shell Sort oferece uma vantagem em termos de desempenho em comparação com os outros algoritmos.

De acordo com a Figura 15, apresentamos uma análise comparativa dos tempos de execução de diferentes algoritmos de ordenação Insertion Sort, Bubble Sort, Selection Sort e Shell Sort considerando entradas randômicas.

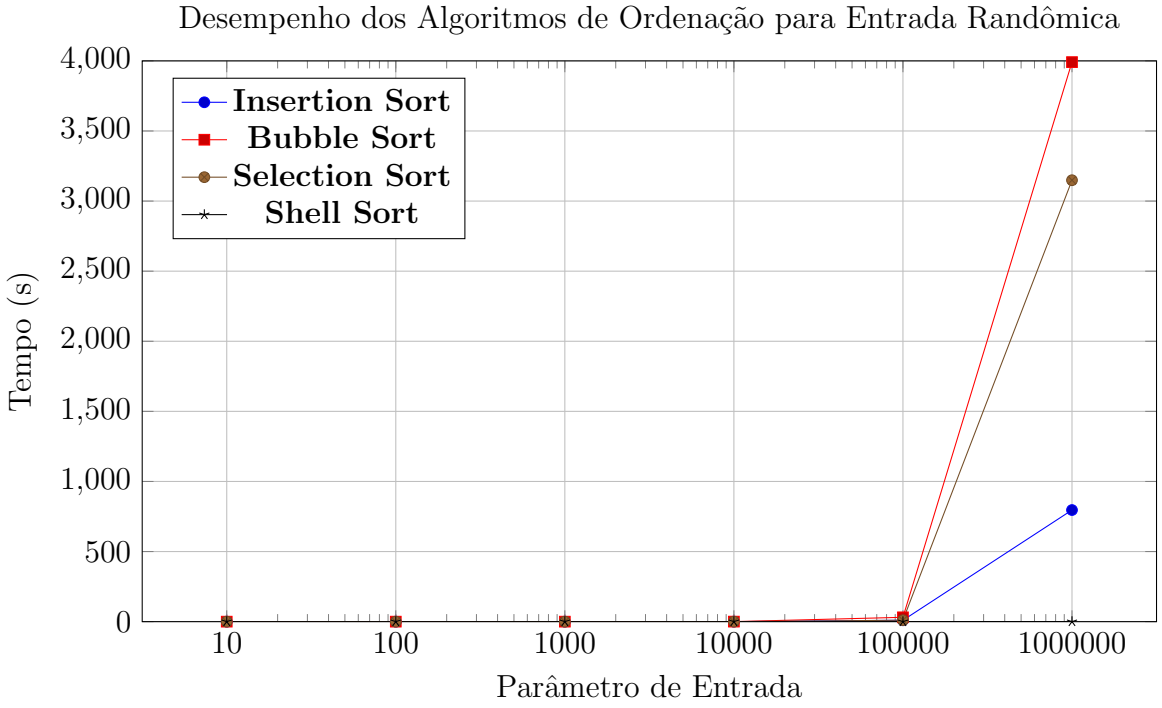

Figura 31: Comparação dos tempos de execução dos algoritmos de ordenação para entrada randômica.

5.9 Tabela comparativa e gráfico geral dos algoritmos de divisão e conquista

A tabela abaixo fornece uma análise comparativa dos tempos de execução, em segundos, de três algoritmos de ordenação baseados em divisão e conquista: **Merge Sort**, **Quick Sort** e **Heap Sort**. Os dados foram obtidos utilizando entradas de tamanhos variados (10 a 1.000.000 elementos), organizadas em três categorias: **crescente**, **decrescente**, **randômica**.

Entrada	Merge Sort	Quick Sort	Heap Sort
Crescente			
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0000	0.0000
100000	0.0160	0.0570	0.0160
1000000	0.1430	0.8750	0.1600
Decrescente			
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.0040	0.0000
100000	0.0160	0.0630	0.0000
1000000	0.2380	0.9810	0.1740
Randômica			
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0010	0.0040	0.0000
100000	0.0320	0.0790	0.0160
1000000	0.2230	1.2700	0.2380

Tabela 17: Comparação dos tempos de execução dos algoritmos Merge Sort, Quick Sort e Heap Sort para entradas crescentes, decrescentes e randômicas (em segundos).

5.9.1 Entrada crescente

No caso de entradas crescentes, os algoritmos de divisão e conquista apresentam comportamentos diferentes. O **Merge Sort** demonstra um desempenho consistente, com tempos de execução baixos, como 0,016 segundos para 100.000 elementos e 0,143 segundos para 1.000.000 de elementos. Isso se deve à sua abordagem baseada na divisão uniforme e fusão,

que é independente da ordem inicial dos dados. Sua complexidade de tempo é $O(n \log n)$, tornando-o uma escolha eficiente para entradas crescentes.

O **Quick Sort**, apresenta um desempenho inferior em comparação ao Merge Sort para entradas crescentes. Seus tempos de execução são inferiores, como 0,057 segundos para 100.000 elementos e 0,875 segundos para 1.000.000 de elementos. Essa diferença ocorre devido ao aumento no número de partições desbalanceadas causadas pela ordem inicial dos dados, o que compromete o desempenho do Quick Sort.

O **Heap Sort**, apresenta tempos de execução semelhantes aos do Merge Sort para entradas crescentes, como 0,016 segundos para 100.000 elementos e 0,160 segundos para 1.000.000 de elementos. A estabilidade do Heap Sort decorre de sua construção inicial do heap, que é seguida por operações de extração repetidas. Esse método é insensível à ordem inicial dos dados, resultando em uma boa performance.

É importante observar que, embora a entrada crescente possa ser considerada um ponto de partida favorável em comparação com outros cenários, ela não é a configuração ideal para algoritmos de divisão e conquista. Esses algoritmos foram projetados para lidar eficientemente com dados desordenados. A dependência de partições, fusões e ajustes de estrutura de dados pode levar a pequenas ineficiências em cenários onde os dados já estão ordenados. Esses algoritmos acabam realizando operações que poderiam ser otimizadas.

Ao analisar o desempenho em entradas crescentes, percebe-se que os algoritmos de divisão e conquista ainda apresentam bons resultados, mas com uma leve desvantagem em relação a algoritmos especialmente otimizados para dados ordenados, como o Insertion Sort e o Shell Sort, conforme demonstrado no estudo. Os resultados sugerem que, para entradas ordenadas, a escolha do algoritmo deve considerar o comportamento específico de cada método e suas implicações no tempo de execução.

De acordo com a Figura 25, apresentamos uma análise comparativa dos tempos de execução de diferentes algoritmos de ordenação Merge Sort, Quick Sort e Heap Sort considerando entradas crescentes.

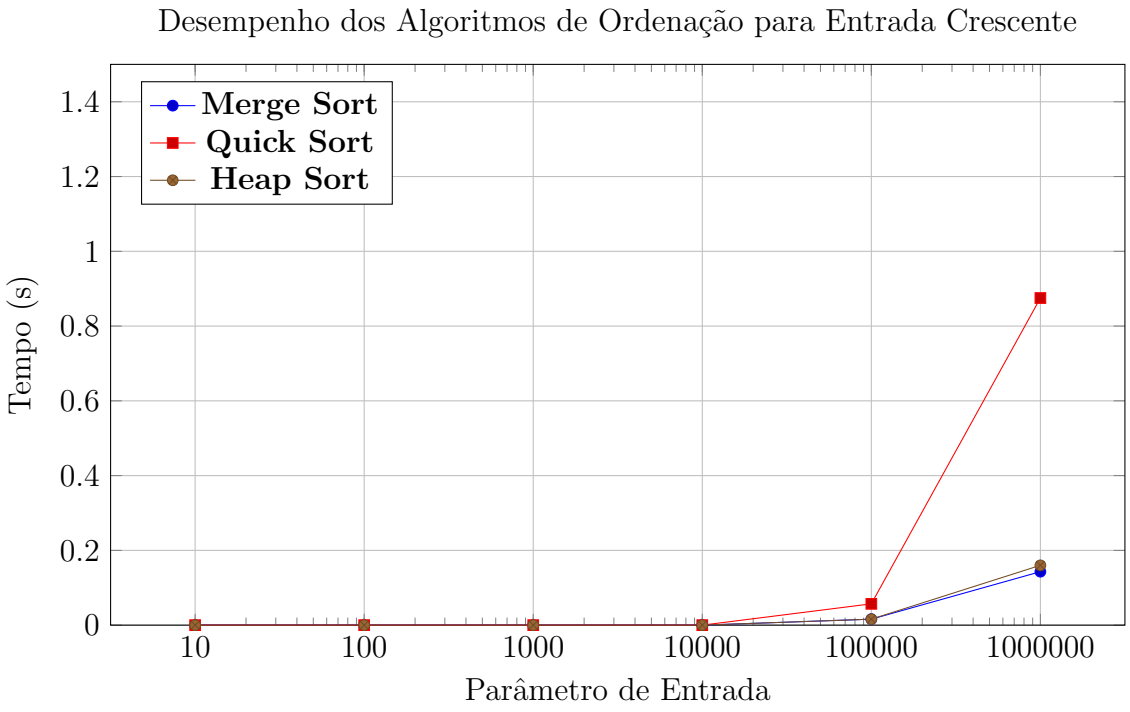

Figura 32: Comparação dos tempos de execução dos algoritmos de ordenação para entrada crescente.

5.9.2 Entrada decrescente

No caso de entradas **decrescentes**, os algoritmos de divisão e conquista apresentam um comportamento diferente. O **Merge Sort** continua demonstrando um bom desempenho, com tempos de execução baixos, como 0,016 segundos para 100.000 elementos e 0,238 segundos para 1.000.000 de elementos. Sua abordagem de divisão e fusão garante que o tempo de execução seja constante. Isso ocorre porque o Merge Sort divide os dados em subproblemas menores e sempre realiza a fusão de forma eficiente. Sua complexidade permanece $O(n \log n)$, o que o torna uma escolha estável e eficaz para entradas decrescentes.

Por outro lado, o **Quick Sort** mostra um desempenho inferior em relação ao Merge Sort para entradas decrescentes. Embora o tempo de execução para entradas pequenas (10, 100, 1000) seja insignificante, a performance se deteriora com o aumento do tamanho da entrada. Para 100.000 elementos, o tempo de execução sobe para 0,063 segundos, e para 1.000.000 de elementos, atinge 0,981 segundos. Esse aumento no tempo é uma consequência da dificuldade do Quick Sort em lidar com dados em ordem inversa. O algoritmo depende de

um bom particionamento dos dados, mas em entradas decrescentes, o particionamento pode resultar em desequilíbrios.

O **Heap Sort**, mantém uma performance mais estável. Para entradas pequenas, seus tempos de execução são praticamente nulos, como 0 segundos para 10, 100 e 1000 elementos. Mesmo com 1.000.000 de elementos, o tempo de execução permanece em 0,174 segundos, um valor que é consideravelmente melhor do que o Quick Sort para entradas decrescentes. A razão para isso é que o Heap Sort não depende da ordem dos dados, realizando uma série de operações de extração a partir da construção inicial do heap, o que o torna insensível à ordem inicial dos elementos.

Embora o Merge Sort ainda se destaque em entradas decrescentes pela sua estabilidade, o Quick Sort e o Heap Sort apresentam desafios adicionais, principalmente o Quick Sort, que sofre com partições desbalanceadas. O Heap Sort, apesar de um aumento no tempo de execução com entradas maiores, ainda se mantém competitivo em termos de desempenho, especialmente em dados decrescentes.

É importante notar que, para entradas decrescentes, a escolha do algoritmo deve considerar a sensibilidade à ordem dos dados. O Merge Sort é uma boa opção quando a estabilidade do tempo de execução é crucial, enquanto o Quick Sort, embora eficiente em muitos casos, pode ser penalizado com entradas decrescentes. O Heap Sort, oferece uma solução eficiente que não depende da ordem dos dados.

Ao analisar o desempenho dos algoritmos para entradas decrescentes, podemos concluir que o Merge Sort se sai melhor em termos de consistência, enquanto o Quick Sort enfrenta maiores dificuldades em dados decrescentes. O Heap Sort, com sua estabilidade, também é uma boa opção.

De acordo com a Figura 26, apresentamos uma análise comparativa dos tempos de execução dos algoritmos Merge Sort, Quick Sort e Heap Sort considerando entradas decrescentes.

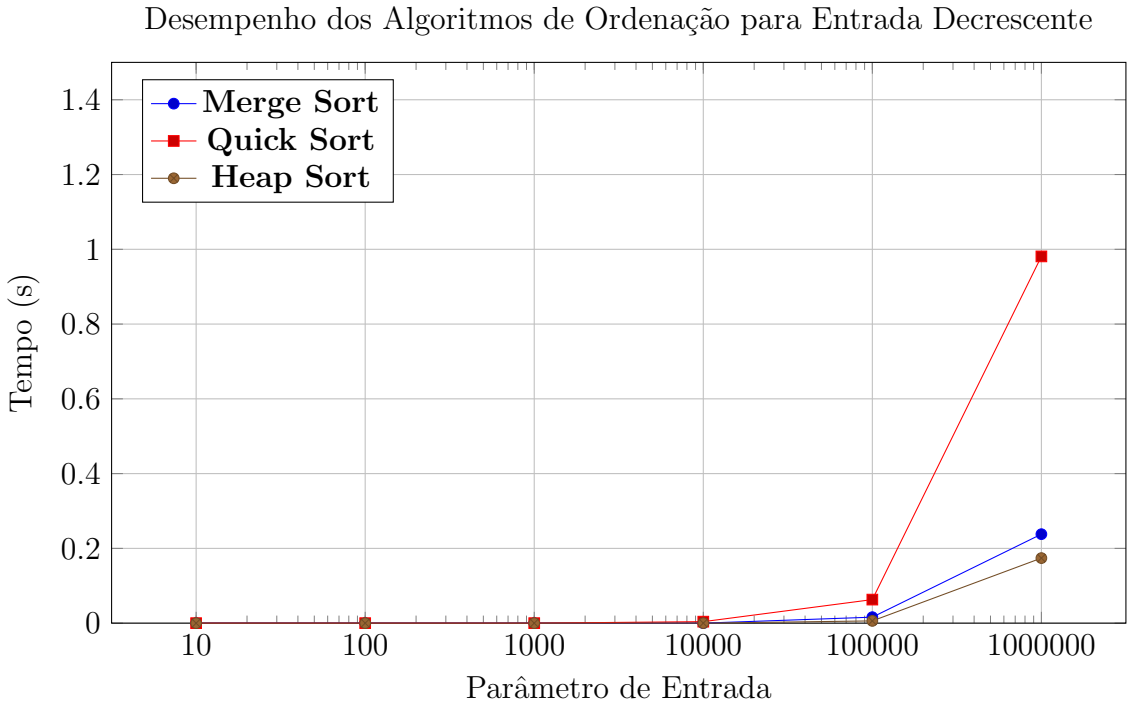

Figura 33: Comparação dos tempos de execução dos algoritmos de ordenação para entrada decrescente.

5.9.3 Entrada Randômica

Quando lidamos com entradas randômicas, os algoritmos de ordenação de divisão e conquista demonstram grandes diferenças.

O **Merge Sort**, apesar de ser um algoritmo eficiente, mantém tempos de execução em entradas randômicas, como 0,032 segundos para 100.000 elementos e 0,223 segundos para 1.000.000 de elementos. A abordagem de divisão uniforme e fusão do Merge Sort continua a ser eficaz, mesmo com a aleatoriedade dos dados, proporcionando um bom desempenho geral.

O **Quick Sort**, que é normalmente eficiente em cenários randômicos, apresenta um desempenho pior do que o Merge Sort e o Heap Sort para entradas grandes. Para 100.000 elementos, o tempo de execução é de 0,079 segundos, e para 1.000.000 de elementos, o tempo sobe para 1,270 segundos, sendo o maior tempo de execução entre os três algoritmos. Esse aumento no tempo é atribuído a um maior número de partições desbalanceadas, que ocorrem devido à aleatoriedade dos dados e à natureza do algoritmo.

O **Heap Sort** mostra-se o mais eficiente entre os três para entradas randômicas, com tempos de execução de 0,016 segundos para 100.000 elementos e 0,238 segundos para 1.000.000 de elementos. A construção inicial do heap e as operações de extração subsequentes são bem adaptadas para dados aleatórios, permitindo ao Heap Sort apresentar resultados rápidos e estáveis.

Embora a entrada randômica seja um cenário desafiador para todos os algoritmos de ordenação, o Heap Sort se destaca em termos de eficiência. O Merge Sort também oferece uma boa performance, embora seus tempos de execução sejam um pouco mais elevados, enquanto o Quick Sort sofre uma degradação de desempenho à medida que o tamanho dos dados aumenta. Esse comportamento sugere que, para dados aleatórios, algoritmos como o Heap Sort ou o Merge Sort devem ser preferidos, dependendo do contexto e das necessidades de desempenho.

De acordo com a Figura 27, apresentamos uma análise comparativa dos tempos de execução dos algoritmos Merge Sort, Quick Sort e Heap Sort considerando entradas randômicas.

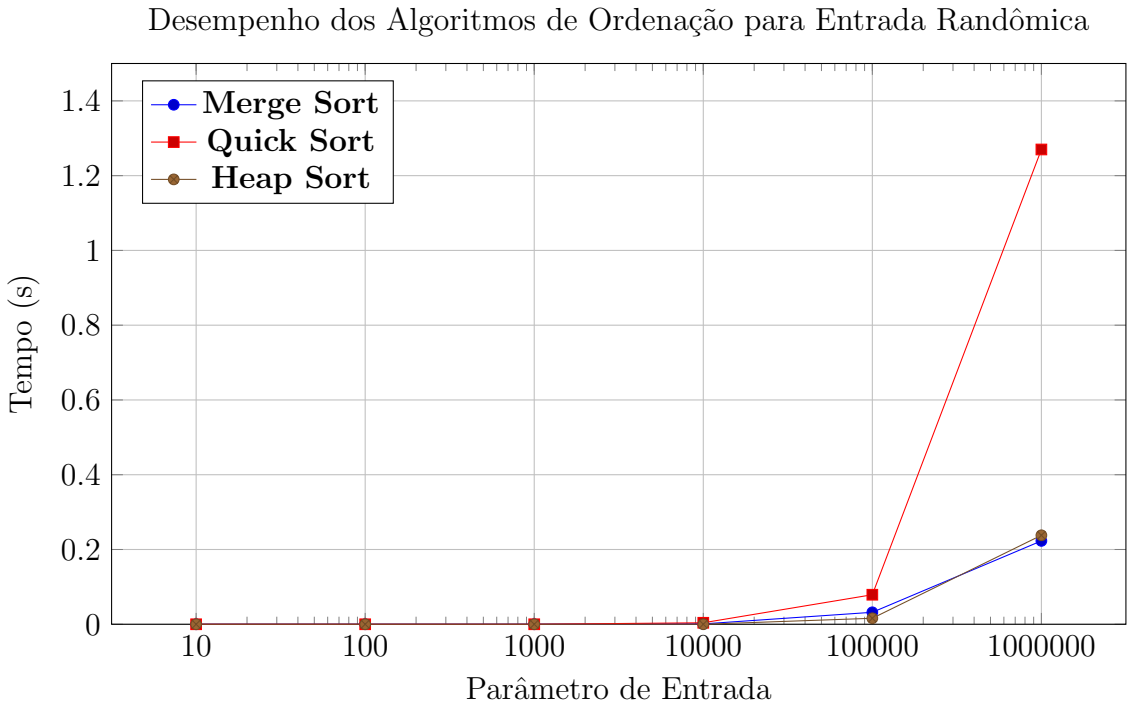

Figura 34: Comparação dos tempos de execução dos algoritmos de ordenação para entrada randômica.

6 CONCLUSÃO

A análise comparativa dos tempos de execução dos algoritmos de ordenação **Insertion Sort**, **Bubble Sort**, **Selection Sort**, **Shell Sort**, **Merge Sort**, **Quick Sort** e **Heap Sort** revela diferenças em seu desempenho, dependendo do tipo de entrada e do tamanho dos dados.

Para entradas em ordem crescente, os algoritmos Insertion Sort e Shell Sort demonstram eficiência, com tempos de execução quase nulos, mesmo para grandes volumes de dados, como no caso de 1.000.000 de elementos. Esses algoritmos são mais adequados para dados já ordenados ou quase ordenados, onde suas vantagens se tornam evidentes. Por outro lado, o Bubble Sort e o Selection Sort continuam a apresentar tempos de execução elevados, mesmo em dados que já estão ordenados. Entre os algoritmos baseados em divisão e conquista, o Merge Sort e o Quick Sort mantêm desempenho consistente, mas não se destacam nesse cenário, enquanto o Heap Sort apresenta tempos razoáveis, embora inferiores ao Shell Sort.

Em cenários com entradas decrescentes, que representam um dos casos mais desafiadores para os algoritmos, o Insertion Sort e o Shell Sort sofrem um aumento no tempo de execução, principalmente em entradas de grandes. Esses algoritmos precisam realizar muitas movimentações de elementos para ordenar os dados. O Bubble Sort e o Selection Sort, embora igualmente ineficientes, mantêm-se com tempos elevados de execução. Por outro lado, o Merge Sort e o Heap Sort se destacam pela estabilidade e pela capacidade de lidar bem com grandes conjuntos de dados, enquanto o Quick Sort, apesar de eficiente, pode enfrentar dificuldades devido ao aumento no número de particionamentos, especialmente se a escolha do pivô for desfavorável.

Para entradas randômicas, o Shell Sort apresenta um desempenho superior ao do Bubble Sort e Selection Sort, que continuam a ser ineficazes em dados aleatórios. O Insertion Sort, embora eficiente para entradas pequenas, também mostra crescimento nos tempos de execução à medida que a entrada se torna maior. Nesse cenário, os algoritmos baseados em divisão e conquista, como Merge Sort e Quick Sort, são os mais eficientes, com o Quick Sort geralmente liderando em desempenho devido à sua menor necessidade de espaço adicional. O Heap Sort também é competitivo, mas pode ser ligeiramente mais lento devido à

complexidade das operações de heapify.

Em resumo, a escolha do algoritmo ideal depende do cenário. Para entradas ordenadas ou quase ordenadas, o Insertion Sort e o Shell Sort são os mais indicados. Para entradas decrescentes, o Merge Sort e o Heap Sort se destacam pela estabilidade e eficiência. Para entradas randômicas, o Quick Sort é geralmente o mais eficiente, seguido de perto pelo Merge Sort. Os algoritmos Bubble Sort e Selection Sort, demonstram um desempenho inferior em todas as situações testadas, tornando-se ineficazes à medida que o tamanho dos dados aumenta.

7 REFERÊNCIAS BIBLIOGRÁFICAS

Referências

[1] Thomas H. Cormen et al. *Algoritmos: Teoria e Prática*. 2ª ed. Rio de Janeiro: Editora Campus, 2002, p. 296.

[2] João Arthur Batista Moraes. *Merge Sort*. Acessado em 04 de dezembro de 2024. 2019. URL: `https://joaoarthurbm.github.io/eda/posts/merge-sort/`.

[3] Nivio Ziviani e et al. *Projeto de Algoritmos: com Implementações em Pascal e C*. Vol. 2. Luton: Thomson Luton, 2004.

8 Função para calcular o tempo

void operacoes(int tipo, int tamanho)

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

double tempoGasto;

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

//Salva a entrada de numeros

salvarEntrada(tipo, tamanho, vetor);

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

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

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

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

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

//Salva a saida do programa

salvarSaida(tipo, tamanho, vetor);

//Libera a memoria

free(vetor);