Ordenação por seleção de raiz quadrada

Cesar Tallys, Emerson Souza

Setembro 2024

1 Introdução

A ordenação de dados é uma necessidade recorrente em diversas áreas da computação, como no gerenciamento de bancos de dados e no processamento de grandes volumes de dados em tempo real. À medida que lidamos com conjuntos de informações cada vez maiores, o desafio de encontrar métodos que permitam uma ordenação rápida e eficiente torna-se cada vez mais relevante para garantir a performance das aplicações. Algoritmos de ordenação são fundamentais em ambientes comerciais e governamentais, como bancos e instituições financeiras, onde é necessário organizar informações de maneira eficiente, além de serem essenciais em pesquisa operacional para otimização de tarefas, como a atribuição de jobs a processadores em problemas de balanceamento de carga[4].

Neste estudo, exploraremos a fundo o impacto direto da ordenação de dados em um programa computacional. Abordaremos variações do método de Seleção por Raiz Quadrada, para comparações de eficiência. O método da seleção por raiz quadrada divide o conjunto de dados original em partes menores de tamanhos iguais, depois remove o maior elemento de cada parte, compara entre estes qual é o maior de todos, e vai aplicando os maiores ordenados em um array final[2]. O método escolhido durante a etapa de seleção do maior elemento de cada partição pode ser crucial para alguns quesitos, como o desempenho da execução, por exemplo. A eficiência da execução está entre os principais fatores que influenciam o desempenho geral[1].

Nosso objetivo é demonstrar como a escolha entre esses métodos de ordenação pode impactar diretamente o tempo de execução de uma aplicação. Através de testes de carga e análise assintótica, buscaremos evidenciar que a eficiência na ordenação dos dados afeta significativamente o desempenho geral de sistemas que lidam com volumes de grande escala, influenciando a experiência do usuário e o consumo de recursos computacionais[3].

A comparação será feita entre dois algoritmos para ordenar essas partes menores do método de seleção: o bubble sort, conhecido por sua complexidade quadrática[1], e o uso de uma estrutura de heaps, que facilita a obtenção do maior elemento devido à sua característica de manter o maior valor na posição inicial[2].

2 Ordenação por seleção de raiz quadrada

O método de seleção por raiz quadrada é uma técnica de otimização usada em algoritmos que envolvem a seleção de itens de um conjunto, como algoritmos de busca ou ordenação. A ideia principal é dividir o problema original em subproblemas menores para reduzir o custo de busca ou seleção. Neste projeto, trataremos da ordenação de dados.

O processo começa dividindo o conjunto de dados original em partes (arrays) de tamanhos iguais, ou aproximadamente iguais, pois podemos ter na última parte deste conjunto um tamanho menor, caso o número total de elementos não seja múltiplo da raiz quadrada do tamanho do conjunto. Em seguida, o algoritmo seleciona o maior elemento de cada array e o insere em um Array final, garantindo a ordenação final do conjunto. Cada elemento que é inserido no Array final deve ser removido do

subarray de origem, para que não haja elementos repetidos. O processo é executado continuamente até que todos os subarrays estejam vazios.

Para ilustrar a explicação de forma mais didática, consideramos um vetor com tamanho que tenha raiz exata, como o 9. Tendo o tamanho do vetor, que chamaremos de n, calculamos a raiz quadrada de n para determinar em quantas partes vamos dividir o vetor. Neste exemplo, $\sqrt{9} = 3$, ou seja, dividiremos nosso vetor em 3 partes iguais (com 3 elementos cada).

Exemplo:
Conjunto = [2, 5, 1, 6, 7, 3, 18, 22, 15]
$n = 9$
$\sqrt{9} = 3$
Subconjuntos = [{2, 5, 1},{ 6, 7, 3},{18, 22, 15}]

Tendo o conjunto partido, empregamos algum método para obtenção do maior elemento de cada subconjunto, nessa fase, podemos escolher entre os diversos métodos de ordenação ou usar alguma estrura que lhe garanta ter o maior elemento e a posição dele no subconjunto. O resultado é armazenado em um vetor auxiliar (Array final):

Para explicação, ordenamos todos os subconjuntos, garantindo que o maior elemento esteja na última posição:

Conjunto = [{1, 2, 5},{ 3, 6, 7},{15, 18, 22}]

Em seguida, procura-se os maiores elementos de cada partição e faz-se uma comparação para encontrar o maior entre eles, nesse caso temos como os maiores de cada partição o [5, 7, 22] e o maior entre eles é o 22, então inserimos esse valor no vetor auxiliar.

Vetor auxiliar: 22

Removemos o elemento do seu subconjunto de origem e repetimos os passos acima até termos inseridos todos os elementos do conjunto no vetor auxiliar.

Conjunto = [{1, 2, 5},{ 3, 6, 7},{15, 18}]
Vetor auxiliar: 22, 18

Conjunto = [{1, 2, 5},{ 3, 6, 7},{15}]
Vetor auxiliar: 22, 18, 15

Conjunto = [{1, 2, 5},{ 3, 6, 7}}]
Vetor auxiliar: 22, 18, 15, 7

Conjunto = [{1, 2, 5},{ 3, 6}]
Vetor auxiliar: 22, 18, 15, 7, 6

Conjunto = [{1, 2, 5},{ 3}]
Vetor auxiliar: 22, 18, 15, 7, 6, 5

Conjunto = [{1, 2},{ 3}]
Vetor auxiliar: 22, 18, 15, 7, 6, 5, 3

Conjunto = [{1, 2}]
Vetor auxiliar: 22, 18, 15, 7, 6, 5, 3, 2

Conjunto = [{1}]

Vetor auxiliar: 22, 18, 15, 7, 6, 5, 3, 2, 1

Conjunto = [ ]

Vetor auxiliar: 22, 18, 15, 7, 6, 5, 3, 2, 1

No final, teremos o vetor auxiliar ordenado.

O impacto no desempenho do algoritmo de seleção por raiz quadrada depende significativamente da escolha do método usado para obtenção dos maiores elementos de cada subparte. Algoritmos mais rápidos, como estruturas de árvore, podem reduzir o tempo total de execução ao lidar com blocos menores, enquanto algoritmos menos eficientes podem aumentar o custo computacional. Portanto, a escolha adequada do método pode fazer uma diferença significativa no desempenho global, otimizar o tempo de busca e melhorar a eficiência ao lidar com grandes volumes de dados. Agora vamos analisar a implementação deste método através de duas abordagens: uma usando o bubble sort para ordenação e obtenção dos maiores elementos e outra usando estrutura de Heaps. Será realizada uma análise assintótica para compreensão da complexidade dos algoritmos, assim entenderemos o comportamento da curva de crescimento da quantidade de cálculos executados em relação ao tamanho da entrada de dados.

3 Análise assintótica dos algoritmos

3.1 Bubblesort:

Pela forma como o bubblesort funciona, comparando cada elemento do array com o elemento adjacente a ele, e os trocando caso os elementos estiverem fora de ordem, quando tivermos um array de tamanho n, o número que teremos de comparações e trocas feitas no array será de aproximadamente n. E como o bubblesort precisa fazer isso para n vezes, ou seja um para cada elemento, teremos que o número total de trocas e comparações será de $n \times n = n^2$. Logo, temos que a complexidade assintótica para o bubblesort é $O(n^2)$, onde n é o número de elementos a serem ordenados. Em resumo temos então:

Pior caso:
Trocas: $\frac{n(n-1)}{2} = O(n^2)$
Comparações: $\frac{n(n-1)}{2} = O(n^2)$

A complexidade do bubblesort é $O(n^2)$ no pior caso, onde n é o número de elementos a serem ordenados. Quando você divide um vetor de tamanho n em k partes, cada subarray terá um tamanho de aproximadamente $\frac{n}{k}$.

Para cada subarray de tamanho $\frac{n}{k}$, o tempo de execução do bubblesort será:

$$O\left(\left(\frac{n}{k}\right)^2\right)$$

Se houver k subarrays, o tempo total necessário para ordenar todas as partes será a soma dos tempos de execução para cada subarray:

$$T_{\text{total}} = k \times O\left(\left(\frac{n}{k}\right)^2\right)$$

Substituindo $\frac{n}{k}$ na fórmula, obtemos:

$$T_{\text{total}} = k \times O\left(\frac{n^2}{k^2}\right) = O\left(\frac{n^2}{k}\right)$$

Agora, se considerarmos dividir o vetor original em $\sqrt{n}$ partes ($k = \sqrt{n}$):

$$T_{\text{total}} = O\left(\frac{n^2}{\sqrt{n}}\right) = O\left(n^{3/2}\right)$$

Portanto, a análise original estava correta. O tempo total para ordenar todas as partes usando bubblesort, quando dividido em $\sqrt{n}$ partes, é de fato $O(n^{3/2})$. Embora a divisão do array em subarrays possa melhorar a eficiência do Bubblesort, essa técnica ainda não torna o algoritmo competitivo em comparação com outros algoritmos de ordenação mais eficientes.

3.2 Heap:

Uma forma de entender a complexidade assintótica de um heap é analisando como ele é mantido e manipulado. Em um heap, a principal operação envolvida é a de *heapify*, que é usada para garantir que as propriedades da heap sejam mantidas ao adicionar ou remover elementos. A operação de *heapify* possui complexidade $O(\log n)$, onde n é o número de elementos na heap, pois ela percorre a altura da árvore binária para reposicionar um elemento.

Quando construímos um heap a partir de um array de n elementos, a operação de *heapify* é aplicada repetidamente para cada elemento, resultando em uma complexidade de $O(n \log n)$ no pior caso. Isto é porque, para cada inserção ou remoção de elemento, é necessário realizar uma reorganização da estrutura de heap, que leva tempo $O(\log n)$, e, ao aplicarmos esta operação a n elementos, temos uma complexidade total de $O(n \log n)$.

No entanto, ao apenas manter ou inserir elementos em um heap existente, a complexidade para cada operação de inserção ou remoção é $O(\log n)$, pois estas operações envolvem ajustes ao longo da altura da árvore, que é logarítmica em relação ao número de elementos. No caso médio e no pior caso, a complexidade de tempo associada ao uso do heap, especialmente ao manter sua propriedade, é geralmente $O(\log n)$ para operações individuais, e $O(n \log n)$ quando aplicadas a todos os elementos de uma vez.

Em resumo, a complexidade de tempo para a manutenção e manipulação de um heap é geralmente limitada por $O(\log n)$ para operações individuais e $O(n \log n)$ para operações que envolvem todos os elementos. Em resumo temos:

Pior caso:
Trocas: $n \log n = O(n \log n)$
Comparações: $n \log n = O(n \log n)$

3.2.1 Construção da Heap para Cada Subarray

Se você está dividindo o array original em k subarrays de tamanho $\frac{n}{k}$, o tempo necessário para construir a heap para cada subarray será:

$$O\left(\frac{n}{k}\right)$$

Portanto, se você tiver k subarrays, o tempo total para construir a heap em todos os subarrays será:

$$T_{\text{heap total}} = k \times O\left(\frac{n}{k}\right) = O(n)$$

Isso significa que o tempo total para construir a heap de cada lado (em todos os subarrays) será proporcional ao tamanho total do array, ou seja, $O(n)$.

3.2.2 Resumo

- **Tempo para construir a heap para cada subarray**: $O\left(\frac{n}{k}\right)$
- **Tempo total para construir a heap em todos os subarrays**: $O(n)$

Assim, o tempo para construir a heap de cada lado (considerando todos os subarrays) é $O(n)$.

4 Análise empírica

A análise empírica consiste em avaliar o desempenho de algoritmos através de experimentos práticos, observando como eles se comportam em cenários reais ao invés de apenas estudar suas propriedades teóricas. Esse tipo de análise envolve a implementação dos algoritmos e a execução de testes em diferentes conjuntos de dados, medindo o tempo de execução, o número de operações realizadas e outros indicadores de desempenho. Neste relatório, a análise empírica será realizada para a implementação das duas abordagens citadas do método de seleção por raiz quadrada: usando árvore heap e ordenação por bubble sort para obteção do maior elemento dos subconjuntos e uma estruturação que facilite a obtenção dos próximos maiores após remoção do elemento maior. A finalidade dessas implementações é fazer uso do método para ordenar o conjunto de dados. O objetivo é compreender como esses algoritmos se comportam em diferentes tamanhos de dados e distribuições. Em seguida, os resultados empíricos serão comparados com a análise assintótica, nos dando uma visão prática detalhada sobre a eficiência e escalabilidade desses algoritmos, confirmando ou refutando as observações teóricas.

Os testes foram realizados em uma máquina equipada com um processador AMD Ryzen 5 4600G e 16 GB de RAM a 3200 MHz. Os algoritmos foram desenvolvidos utilizando a linguagem *JavaScript* [7] e executados em *Node.js*, que compila o programa com o motor V8. Para a avaliação dos métodos, foram utilizados como entrada arrays de inteiros desordenados com tamanhos $N = 10^4$, $N = 10^5$, $N = 10^6$ e $N = 10^7$ preenchidos com números aleatórios entre 1 e N. Em cada tamanho de array, foram medidas as performances das duas abordagens do método de seleção por raiz quadrada. Isso resultou em quatro conjuntos distintos de medições de tempo (em segundos) de execução para cada uma das duas abordagens. Os tempos registrados foram então dispostos em gráficos para facilitar a visualização e a comparação dos desempenhos dos algoritmos em relação às diferentes variáveis testadas. Esses gráficos serão apresentados a seguir para fornecer uma análise visual detalhada dos resultados obtidos:

4.1 Bubble sort

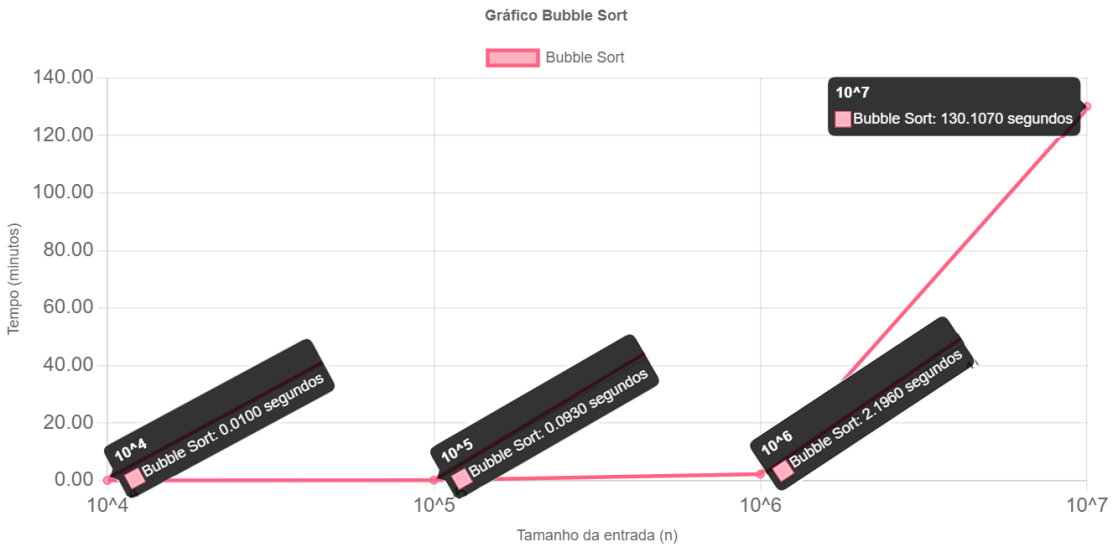

Figure 1: Método da seleção por raiz quadrada com Bubblesort

Ao analisar a curva do gráfico de desempenho do método de seleção por raiz quadrada utilizando BubbleSort para ordenar subconjuntos, nota-se um comportamento que reflete o ganho de eficiência proporcionado pela divisão do array original em subconjuntos menores. Este método utiliza o BubbleSort para ordenar cada um dos subconjuntos e, em seguida, seleciona os maiores elementos de cada subconjunto para compor o resultado final. Para entradas menores, como $n = 10^4$ e $n = 10^5$, o tempo de execução é relativamente baixo, variando de 10 milissegundos ($n = 10^4$) a cerca de 93 milissegundos ($n = 10^5$). Quando o tamanho da entrada aumenta para $n = 10^6$, o tempo de execução sobe para cerca de 2,2 segundos, ainda indicando uma eficiência aceitável.

Contudo, à medida que o tamanho da entrada aumenta, como para $n = 10^7$, o tempo de execução atinge 130 segundos (cerca de 2 minutos), demonstrando uma perda gradual de eficiência. Se considerarmos o comportamento assintótico do BubbleSort e o dividirmos em $\sqrt{n}$ subconjuntos, a complexidade do tempo se torna $O(n^{3/2})$. Para entradas maiores, como $n = 10^8$, essa técnica poderia reduzir o tempo de execução em comparação ao uso direto do BubbleSort sobre o array inteiro, pois somente os subconjuntos precisam ser ordenados uma vez.

No entanto, mesmo utilizando o método de seleção por raiz quadrada, o desempenho pode não ser tão eficiente quanto o de algoritmos de ordenação mais avançados. A complexidade $O(n^{3/2})$ continua significativamente maior do que a de algoritmos com complexidade $O(n \log n)$. Assim, embora esta abordagem represente uma melhoria sobre o BubbleSort tradicional, ela ainda não é ideal para grandes volumes de dados, devido ao comportamento quadrático e à necessidade de manipulação direta dos elementos de cada subconjunto. Portanto, essa abordagem tem sua aplicação limitada a cenários onde a simplicidade de implementação é priorizada sobre a eficiência de tempo.

4.2 Árvores Heap

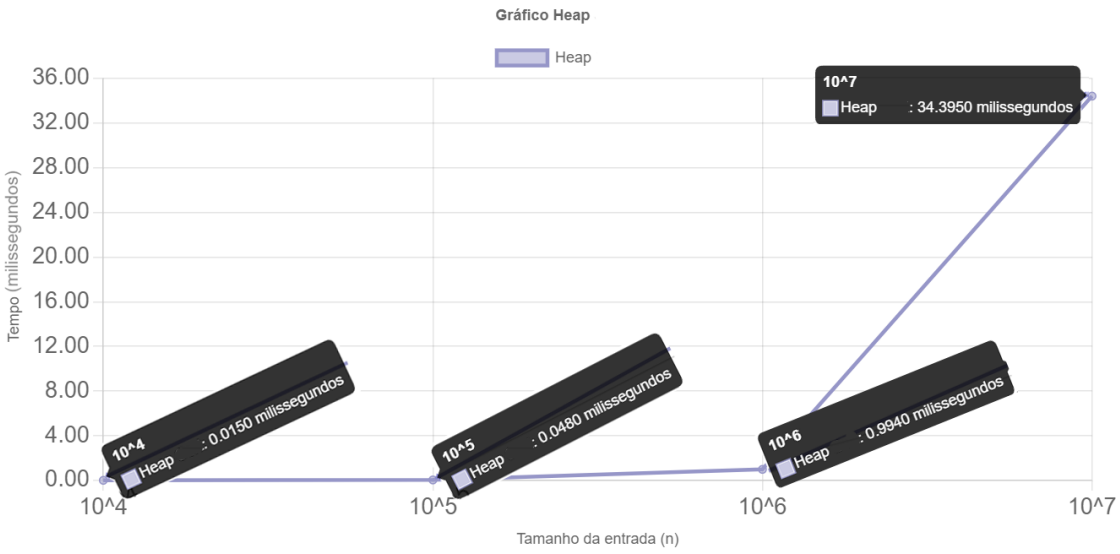

Figure 2: Método da seleção por raiz quadrada com Heaps

A curva do gráfico de desempenho do método de seleção por raiz quadrada utilizando estruturas de Heaps para organizar subconjuntos, reflete a eficiência da estrutura de Heap em lidar com grandes volumes de dados. A principal vantagem deste método é que a heap precisa ser construída apenas uma vez. Após a remoção de elementos, é necessário apenas executar a operação de *heapify* para reestruturar a heap, garantindo assim que as propriedades sejam mantidas.

Para entradas menores, como $n = 10^4$ e $n = 10^5$, o tempo de execução é bastante reduzido, levando 15 milissegundos ($n = 10^4$) e 48 milissegundos ($n = 10^5$), respectivamente. Conforme o tamanho da entrada aumenta para $n = 10^6$, o tempo de execução sobe para aproximadamente 1 segundo, o que ainda indica um desempenho eficiente. Para $n = 10^7$, o tempo de execução é de cerca de 34 segundos, confirmando que o uso de heaps continua a ser eficiente mesmo para volumes maiores de dados.

De acordo com a análise assintótica, a complexidade da construção da heap para um array de n elementos é de $O(n \log n)$. Uma vez que a heap é construída, cada operação subsequente de remoção e reestruturação (*heapify*) possui complexidade $O(\log n)$. Ao dividir o array original em $\sqrt{n}$ subconjuntos e aplicar heaps para cada um, o tempo total de execução se mantém em $O(n \log n)$.

Portanto, ao comparar com o método de ordenação com BubbleSort, a abordagem com heaps mostra-se significativamente mais eficiente devido à sua capacidade de manter uma estrutura ordenada com operações de tempo logarítmico após a construção inicial. Para a entrada $n = 10^8$, espera-se que o tempo de execução seja substancialmente menor do que métodos quadráticos, reforçando a eficiência do uso de heaps para a ordenação de grandes volumes de dados, mesmo que algumas operações adicionais sejam necessárias para manter a propriedade da heap.

Apesar das vantagens em relação a métodos mais básicos como o BubbleSort, o uso de heaps ainda pode ser menos eficiente do que algoritmos otimizados de ordenação, como QuickSort ou MergeSort, que também possuem complexidade $O(n \log n)$, mas com coeficientes constantes mais baixos e otimizações específicas que podem reduzir ainda mais o tempo de execução em cenários práticos.

5 Discussão

Os tempos de ambos os algoritmos foram plotados em um único gráfico (figura 3) para uma melhor comparação de eficiência. Observe:

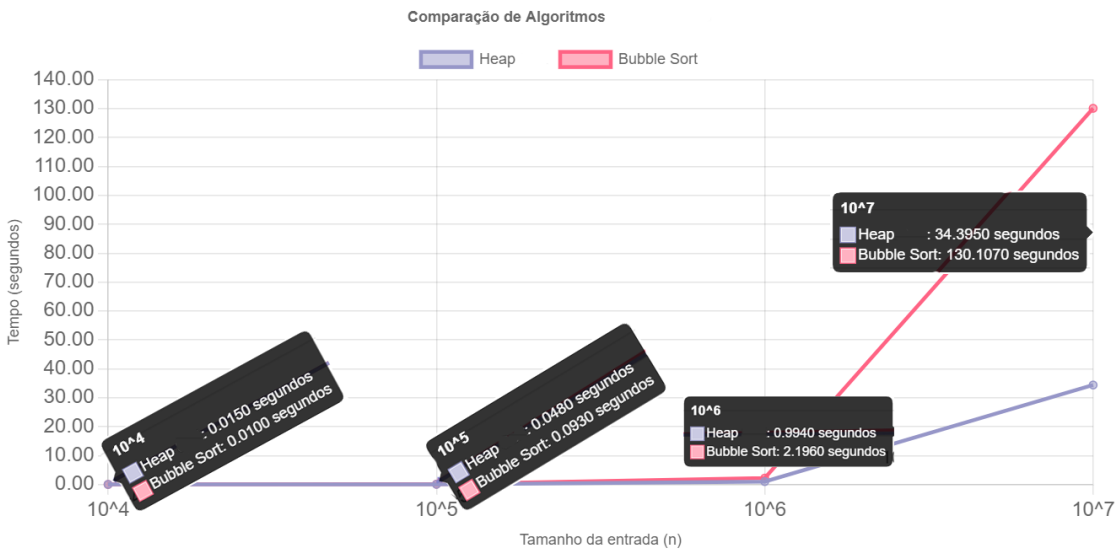

Figure 3: Comparação: Bubble X Heap

Com base na análise do gráfico, torna-se evidente a superioridade do desempenho do método de seleção utilizando Heap em comparação com outras abordagens de ordenação. A curva de crescimento do tempo de execução do algoritmo é visivelmente menos acentuada, indicando uma escalabilidade muito melhor ao lidar com grandes conjuntos de dados. Essa menor inclinação da curva demonstra que o algoritmo é significativamente mais eficiente, especialmente em entradas de tamanho considerável, onde a complexidade assintótica $O(n \log n)$ do Heap permite uma manutenção ordenada dos dados com um custo computacional reduzido por operação, em contraste com métodos de ordenação menos eficientes, como o BubbleSort.

Além disso, a necessidade de realizar o procedimento de *heapify* apenas uma vez para a construção inicial da heap e, posteriormente, apenas para ajustes incrementais após remoções, contribui para um desempenho otimizado ao ordenar grandes volumes de dados. Portanto, a utilização do Heap para seleção se destaca como uma escolha eficaz em aplicações práticas onde o tempo de execução e a eficiência são críticos, mantendo-se competitiva até mesmo contra algoritmos de ordenação avançados, como QuickSort e MergeSort, que compartilham da mesma complexidade assintótica, mas podem não apresentar o mesmo comportamento consistente em todos os casos de uso.

6 Considerações finais

Ao analisarmos o desempenho do método de seleção utilizando a estrutura de Heap, percebemos que ele se destaca especialmente em cenários onde a eficiência para grandes volumes de dados é crucial. A estrutura de Heap, que é uma árvore binária completa, permite que as operações de inserção, remoção e acesso ao elemento de maior ou menor valor sejam realizadas de forma eficiente, com complexidade de tempo de $O(\log n)$. Esta propriedade se deve ao fato de que, após a construção inicial do Heap —

que tem complexidade $O(n)$ —, a manutenção de sua propriedade de ordenação requer apenas ajustes incrementais, o que é significativamente menos custoso comparado a outros algoritmos de ordenação.

Se compararmos o Heap com algoritmos de ordenação mais simples, como o BubbleSort, que tem uma complexidade de $O(n^2)$ no pior caso, a diferença de eficiência torna-se evidente. O BubbleSort realiza um grande número de trocas e comparações, o que o torna inviável para grandes conjuntos de dados. Em contraste, o Heap, após a construção inicial, requer apenas operações logarítmicas para manutenção, o que reduz consideravelmente o tempo total necessário para ordenar os elementos.

Uma das principais vantagens do método de seleção por Heap é que, ao contrário de algoritmos como QuickSort, ele é um algoritmo de ordenação em tempo constante (in-place sorting), o que significa que ele não requer espaço adicional além do necessário para o array original. Isso é crucial em situações de restrições de memória, onde o uso de espaço extra pode ser proibitivo. Além disso, uma vez que o Heap é construído, cada remoção ou ajuste subsequente requer apenas um custo logarítmico, mantendo a eficiência ao longo do processo.

Em aplicações reais, o método de Heap é especialmente útil para tarefas que envolvem seleção repetida de elementos máximos ou mínimos, como em algoritmos de busca de mínimos caminhos (por exemplo, Dijkstra), em sistemas de prioridades (como em filas de prioridade), e em tarefas que envolvem grandes quantidades de dados, como em sistemas de banco de dados e sistemas de agendamento de tarefas. Nestes contextos, o Heap não apenas proporciona uma ordenação eficiente, mas também permite operações rápidas de inserção e remoção de elementos, o que é fundamental para o desempenho geral do sistema.

Em suma, a utilização do método de seleção por Heap se mostra altamente eficiente para grandes conjuntos de dados, principalmente devido à sua complexidade logarítmica em operações de manutenção e à necessidade de apenas uma construção inicial da heap. Enquanto outros algoritmos podem oferecer vantagens em casos específicos, o Heap oferece um desempenho consistentemente bom e é menos suscetível a variações de desempenho com base em características particulares dos dados de entrada. Por essas razões, ele continua a ser uma escolha robusta em uma ampla gama de aplicações computacionais.

7 Referências

1. CORMEN, Thomas H. et al. Introdução a Algoritmos. Rio de Janeiro: Campus, 2002.

2. `https://pessoal.dainf.ct.utfpr.edu.br/maurofonseca/lib/exe/fetch.php?media=cursos:if63c:if63ced_08_ordenacao.pdf`

3. `https://sca.profmat-sbm.org.br/profmat_tcc.php?id1=5511&id2=171053345`

4. Princeton Algorithms. Aplicações Comerciais e de Pesquisa Operacional. Disponível em: `https://www.cs.princeton.edu/algs4/`.

5. Codefinity. Impacto da Evolução dos Algoritmos de Ordenação. Disponível em: `https://codefinity.com/blog/algorithm/`.

6. Feofiloff, Paulo. "Heap: Estrutura de Dados de Heap." Departamento de Ciência da Computação, Instituto de Matemática e Estatística da Universidade de São Paulo. Disponível em: `https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/heap.html`. Acesso em: 9 de setembro de 2024.

7. Github. Repositório dos algoritmos.
`https://github.com/CesarTHD/Variacoes-Metodo-da-Selecao-por-Raiz-Quadrada`