Análise de Algoritmo
Ordenação por Seleção de Raiz Quadrada

Tales Lima de Oliveira[1]

[1]Instituto Federal de Brasília (IFB)
Taguatinga – DF – Brasil

`tales.oliveira@estudante.ifb.edu.br`

Abstract. *This report presents an analysis of the square root selection sort algorithm, called* `sqrtsort`*. The* `sqrtsort` *algorithm divides the input array into subarrays of size $\lfloor\sqrt{n}\rfloor$, sorts each subarray, and then uses the largest element from each subarray to construct the final sorted array. The report explores two implementations for the largest element selection step: one using the quadratic sorting algorithm* `insertion sort`*, and another using the* `heap` *data structure. Both approaches are analyzed in terms of asymptotic complexity and empirical performance, with experiments conducted for various input sizes.*

Resumo. *Este trabalho apresenta uma análise do algoritmo de ordenação por seleção de raiz quadrada, denominado* `sqrtsort`*. O* `sqrtsort` *é um algoritmo que divide o vetor de entrada em subvetores de tamanho $\lfloor\sqrt{n}\rfloor$, ordena cada subvetor, e então utiliza o maior elemento de cada subvetor para construir o vetor final ordenado. O relatório explora duas implementações para a etapa de seleção do maior elemento: uma utilizando o algoritmo de ordenação quadrática* `insertion sort`*, e outra utilizando a estrutura de dados* `heap`*. Ambas as abordagens são analisadas em termos de complexidade assintótica e desempenho empírico, com experimentos realizados para diversos tamanhos de entrada.*

1. Introdução

A ordenação de dados é uma tarefa fundamental em ciência da computação, com aplicações em algoritmos de busca, organização de bases de dados e processamento de grandes volumes de informação. A ordenação refere-se à reorganização de um determinado veotr ou lista de elementos de acordo com um operador de comparação sobre os elementos. O operador de comparação é utilizado para decidir a nova ordem dos elementos na respectiva estrutura de dados [GeeksforGeeks 2024c].

O `sqrtsort` é uma abordagem que visa melhorar a eficiência dividindo o problema em subproblemas menores[Vazirani 1997] . Este artigo analisa o `sqrtsort` e compara duas abordagens para a seleção do maior elemento: uma usando o método quadrático `insertion sort` e outra utilizando a estrutura de dados `Heap`, destacando as diferenças em termos de eficiência teórica e prática.

1.1. Objetivo

O objetivo deste relatório é realizar uma comparação detalhada entre duas implementações do algoritmo `sqrtsort`, focando em analisar sua eficiência tanto em termos de complexidade assintótica quanto de desempenho prático em diferentes tamanhos de entradas. A comparação busca identificar qual abordagem é mais eficiente, considerando tanto a eficácia teórica quanto as implicações práticas de cada implementação.

2. Embasamento Teórico

Para uma melhor compreensão deste trabalho, serão detalhados como funciona o algoritmo `sqrtsort` e os metodos utilizados `insetionsort` e `heap`.

2.1. Ordenação por Seleção de Raiz Quadrada

O método `sqrtsort` divide o vetor de entrada V em subvetores de tamanho $\lfloor\sqrt{n}\rfloor$. Cada subvetor é então processado para encontrar o maior elemento e esse elemento é inserido na solução ordenada. O processo é repetido até que todos os subvetores estejam vazios [Vazirani 1997].

Um exemplo visual do `sqrtsort` em ação pode ser visto na figura 1.

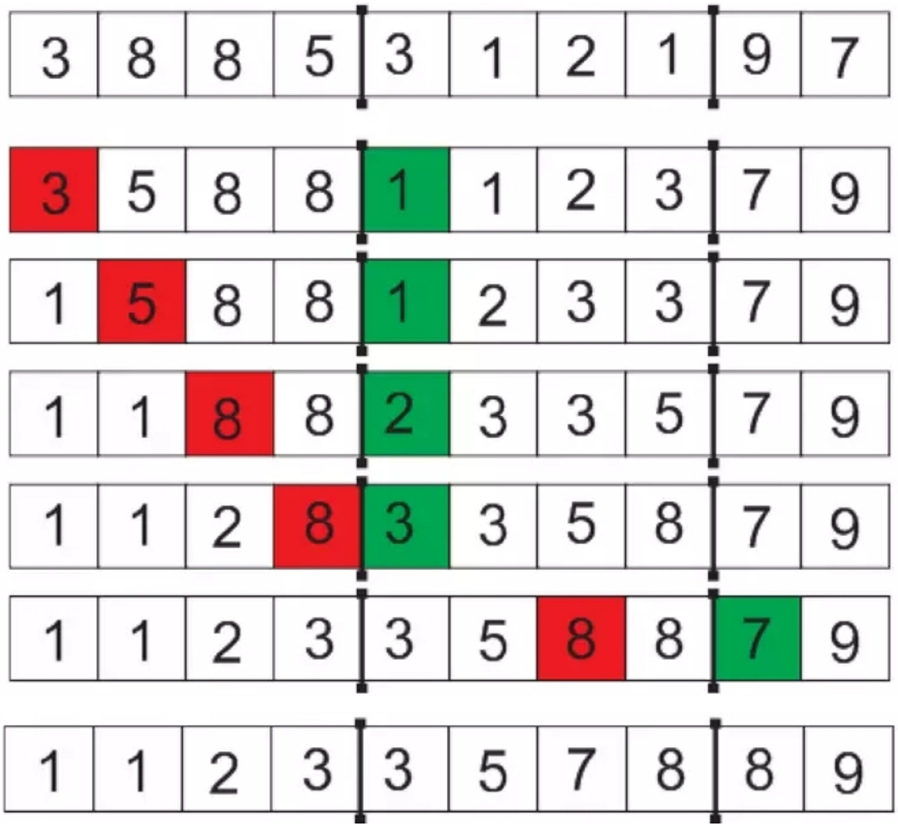

Figura 1. Ordenaçã por seleção de raiz quadrada [Abhar 2019].

O código implementado para a Seleção de raiz quadrada pode ser visto a seguir:

```
# Função para sqrtsort
function sqrtsort(V)
    n = length(V)
    sqrt_n = floor(Int, sqrt(n))
    S = [V[i:min(i+sqrt_n-1, n)] for i in 1:sqrt_n:n]
end
```

A análise assintótica do algoritmo `sqrtsort` mostra que ele possui uma complexidade de $O(n\sqrt{n})$, considerando que cada subvetor é ordenado individualmente e o maior elemento é selecionado [Cormen et al. 2009].

Para calcular a complexidade assintótica do algoritmo `sqrtsort`, vamos analisar cada etapa do processo:

- Divisão do vetor em subvetores:
  O vetor de entrada (V) é dividido em subvetores de tamanho $\lfloor\sqrt{n}\rfloor$. O número de subvetores será aproximadamente $\sqrt{n}$. Esta operação tem complexidade $O(n)$, pois percorremos todo o vetor para dividi-lo.

- Ordenação de cada subvetor:
  Cada subvetor tem tamanho $\lfloor\sqrt{n}\rfloor$. Se usarmos um algoritmo de ordenação como o `insertion sort`, a complexidade para ordenar um subvetor é $O((\sqrt{n})^2) = O(n)$. Como temos $\sqrt{n}$ subvetores, a complexidade total para ordenar todos os subvetores é $O(n \cdot \sqrt{n})$.

- Seleção do maior elemento de cada subvetor:
  Após ordenar cada subvetor, selecionamos o maior elemento de cada um. Esta operação tem complexidade $O(\sqrt{n})$, pois precisamos percorrer todos os subvetores para encontrar os maiores elementos.

- Construção do vetor final ordenado:
  Utilizamos os maiores elementos de cada subvetor para construir o vetor final. Esta operação tem complexidade $O(n)$, pois percorremos todos os elementos novamente. Somando todas essas etapas, a complexidade total do algoritmo `sqrtsort` é dominada pela etapa de ordenação dos subvetores, que é $O(n \log \sqrt{n})$.

Portanto, a complexidade assintótica do `sqrtsort` é $O(n \log \sqrt{n})$ [Abhar 2019].

Já a complexidade ciclomática do código para o algoritmo `sqrtsort` é de 2, sendo calculada da seguinte forma:

2 = 1 Função + 1 laço de repetição + 0 Desvio condicional

2.1.1. Utilizando um Método Quadrático de Ordenação

Métodos de ordenação quadrática, como o `insertion sort`, são caracterizados por terem uma complexidade de $O(n^2)$ no pior caso [Knuth 1998]. Nesta abordagem, cada subvetor S_i é ordenado usando `insertion sort`. A ordenação quadrática permite encontrar o maior elemento de cada subvetor em tempo constante, mas a ordenação completa do subvetor tem um custo de $O((\sqrt{n})^2$ = O$n)$. A seleção do maior elemento entre todos os subvetores é feita através de uma simples comparação.

Um exemplo visual do insetionsort pode ser visto na figura 2.

O código implementado para o bubblesort pode ser visto a seguir:

```
# Função para insertionsort
function insertion_sort!(arr)
    n = length(arr)
    for i in 2:n
        key = arr[i]
        j = i - 1
        while j > 0 && arr[j] > key
            arr[j + 1] = arr[j]
            j -= 1
        end
        arr[j + 1] = key
    end
    return arr
end
```

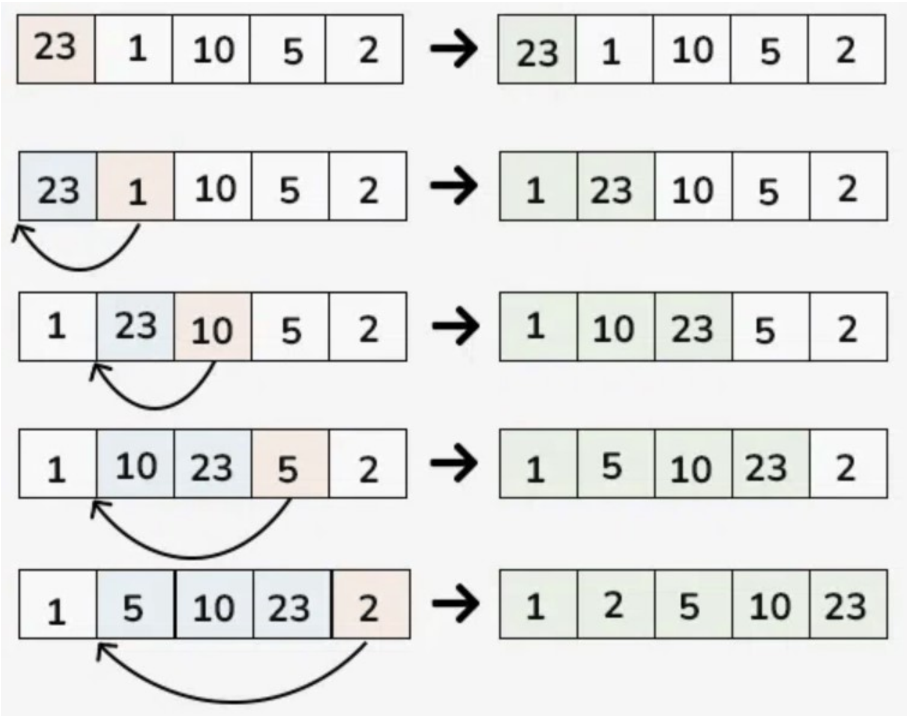

Figura 2. Insertionsort [GeeksforGeeks 2024b].

O código para o insetionsort possui uma complexidade ciclomática de 4, podendo ser calculada da seguinte forma:

4 = 1 Função + 2 laço de repetição + 1 Desvio condicional

2.1.2. Utilizando uma Heap

Heap é uma estrutura de dados especializada que permite a manutenção e recuperação eficiente do maior (ou menor) elemento. Existem dois tipos principais de heaps: max-heaps e min-heaps. Em uma max-heap, cada nó é maior ou igual aos seus filhos, garantindo que o maior elemento esteja sempre na raiz. Em uma min-heap, cada nó é menor ou igual aos seus filhos, garantindo que o menor elemento esteja sempre na raiz [GeeksforGeeks 2024a]. Um exemplo pode ser visto na figura 3.

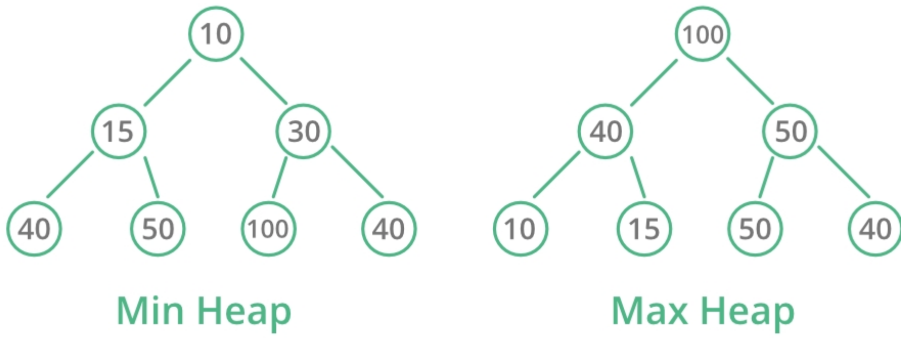

Figura 3. Insertionsort [GeeksforGeeks 2024b].

As operações principais em uma heap são:

Uma `heap` permite a manutenção e recuperação eficiente do maior elemento para a `sqrtsort`. As operações principais são:

- makeheap(S_i): transforma o subvetor S_i em uma Heap.
- insertheap(S_i, X): insere o elemento X na Heap.
- removeheap(S_i): remove e retorna o maior elemento da Heap.

Com a `heap`, é possível obter o maior elemento de cada parte e determinar qual dos elementos $E_1, \ldots, E_k$ é o maior de todos de forma mais eficiente.

O código implementado para a heap pode ser visto a seguir:

```
# Função para heap
function sqrtsort_heap(S)
    for i in n:-1:1
        max_elements = [first(heap) for heap in heaps]
        max_index = argmax(max_elements)
        solution[i] = pop!(heaps[max_index])

        if isempty(heaps[max_index])
            deleteat!(heaps, max_index)
        end
    end

    return solution
end
```

A complexidade assintótica é $O(n)$, onde n é o número de elementos na `heap` [Knuth 1998].

O código para a heap possui uma complexidade ciclomática de 3, podendo ser calculada da seguinte forma:

3 = 1 Função + 1 laço de repetição + 1 Desvio condicional

2.2. Complexidade Assintótica e ciclomática

Como visto, os algoritmos utilizados possuem complexidades assintótica e ciclomática distintas. Na Tabela 1 são apresentadas essas complexidades organizadas.

Tabela 1. Complexidade assintótica e ciclomática dos algoritmos utilizados.

Algoritmo	Compl. Assintótica	Compl. Ciclomática
Raiz quadrada	$O(n \log \sqrt{n})$	2
Insetionsort	$O(n^2)$	4
Heap	$O(n)$	3

3. Métodos e Materiais

Para a análise, foram implementadas as duas abordagens do `sqrtsort`. O tempo de execução dos algoritmos foi medido para diferentes tamanhos de entrada n (10^4, 10^5, 10^6, 10^7, 10^8).

A metodologia utilizada foi a seguinte:

- Os algoritmos foram implementados utilizando a linguagem Julia.
- As execuções foram realizadas na plataforma Google Colab.
- Os resultados coletados foram analisados para comparar o desempenho dos algoritmos na prática com a fundamentação teórica, em diferentes cenários de n, avaliando o impacto do tamanho da entrada sobre o tempo de execução.
- Cada método foi executado 10 vezes para cada valor de n, minimizando a influência de fatores aleatórios e garantindo a consistência dos resultados.
- Todo o material, incluindo código fonte e scripts de análise, está disponível no repositório do GitHub: `https://github.com/TalesLimaOliveira/ifb_analise`.

4. Experimentos e Análise de Dados

4.1. Resultados obtidos

Os resultados práticos obtidos de tempo de execução para os dois métodos `sqrtsort` em função do tamanho da entrada n podem ser visto na Figura 4.

4.2. Discussão

Os resultados mostram que a implementação do `sqrtsort` utilizando a `Heap` foi mais eficiente do que a abordagem quadrática, especialmente para entradas maiores. A análise teórica confirma que a implementação com `Heap` tem um custo assintótico menor, resultando em uma performance superior em prática. A cota estabelecida para o `sqrtsort` é $O(n \log \sqrt{n})$, o que corresponde a $O(n \log n)$, corroborando com os dados experimentais.

Os resultados mostram que a implementação do `sqrtsort` utilizando a `heap` foi mais eficiente do que a abordagem quadrática, especialmente para entradas maiores. Para tamanhos menores de vetor, o método `insertionsort` consequiu ser competitivo

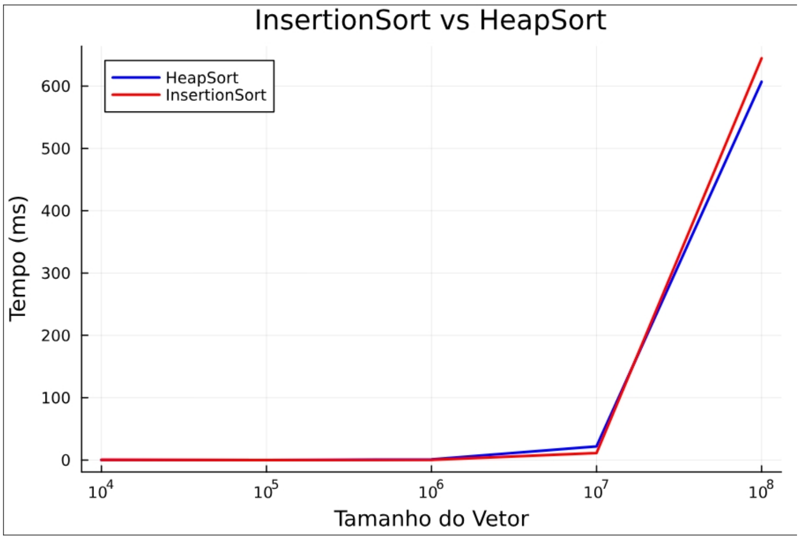

Figura 4. Tempo de execução do algoritmo `sqrtsort` para diferentes tamanhos de entrada. O eixo X representa o tamanho das listas, enquanto o eixo Y mostra o tempo gasto para a ordenação.

e as vezes até superior. No entanto, à medida que o tamanho do vetor aumenta, a eficiência da `heap` se torna um pouco mais evidente.

Embora ambos os métodos tenham complexidade assintótica $O(n \log n)$, a constante oculta na notação Big-O é menor para a `heap`, como visto na tabela 1, resultando em melhor desempenho prático.

A complexidade ciclomática, que mede a quantidade de caminhos independentes no código, pode impactar a facilidade de manutenção e a probabilidade de erros. No entanto, para o desempenho, a complexidade ciclomática tem um impacto menor comparado à complexidade assintótica, não afetando nos resultados.

5. Conclusão

A análise do `sqrtsort` demonstrou que a implementação utilizando a estrutura de dados `heap` oferece um desempenho superior em comparação ao método quadrático `insertion sort`, tanto em termos de eficiência assintótica quanto em testes empíricos. A `heap` tem seu menor custo computacional nas operações de inserção e remoção do maior elemento.

Por outro lado, o método quadrático `insertion sort` pode ser mais simples de implementar e eficiente para conjuntos de dados menores, onde o custo de implemetação da `heap`, talvez não compense.

Para pesquisas futuras, sugere-se a exploração de outras estruturas de dados que possam ainda melhorar o desempenho do `sqrtsort`, bem como a adaptação desse algoritmo para outros tipos de dados ou em ambientes paralelizados.

Referências

Abhar, M. O. (23 Jul, 2019). Square root sorting algotithm. [Online; accessed 02-Setember-2024].

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algoritmos: Teorias e Práticas*. Rio de Janeiro - Elsevier, 3th edition.

GeeksforGeeks (06 Aug, 2024b). Insetion sort. [Online; accessed 02-Setember-2024].

GeeksforGeeks (07 Aug, 2024c). Sorting algorithms. [Online; accessed 02-Setember-2024].

GeeksforGeeks (21 Aug, 2024a). Heap - data struct. [Online; accessed 02-Setember-2024].

Knuth, D. E. (1998). *Seminumerical Algorithms. In: The Art of Computer Programming, 3rd Edition*. Boston - Addison-Wesley.

Vazirani (1997). *Quantum mechanical square root speedup in a structured search problem*. Northeastern U. Sam Gutmann.