CENTRO DE ENSINO UNIFICADO DE BRASÍLIA (UniCEUB)

Henrique de Souza Sogayar

Introdução a algoritmos de busca aplicado: um estudo comparativo

Águas Claras – DF

2024

Sumário

1. Introdução

A base da computação moderna é composta por dados e armazenamento de dados, tornando crucial a utilização de algoritmos eficientes para a busca desses dados. Neste estudo, analisaremos alguns dos principais algoritmos de busca, destacando suas implementações, testes e comparações. Além disso, exploraremos a complexidade de tempo e de espaço desses algoritmos para entender sua eficiência em diferentes contextos.

A complexidade de espaço refere-se à quantidade de memória que um algoritmo requer para sua execução, enquanto a complexidade de tempo se concentra no tempo necessário para executar o algoritmo em relação ao tamanho da entrada. Compreender esses conceitos é essencial para a escolha de algoritmos adequados para diferentes aplicações, especialmente à medida que os conjuntos de dados continuam crescendo em tamanho e complexidade.

Esse estudo se propôs a aprofundarmos a análise desses algoritmos, examinando suas características, aplicações com o objetivo de explicar de maneira didática e democrática o assunto em questão para auxiliar a compreensão sobre o assunto. Até hoje, as pesquisas sobre o tema costumam utilizar termos técnicos e necessitam de muito conhecimento prévio e por isso essa pesquisa se justifica. Por meio deste estudo é possível concluir que, algoritmos de ordenação mais complexos, conseguem ordenar grandes entradas em muito menos tempo, mas que, em pequenas entradas, algoritmos de menor complexidade

conseguem ordenar listas com praticamente o mesmo tempo.

2. Complexidade de espaço

A complexidade de espaço de um algoritmo é uma função que descreve a quantidade de memória que um algoritmo usa em relação ao tamanho da entrada. Ela inclui tanto o espaço necessário para armazenar a entrada quanto o espaço adicional necessário para a execução do algoritmo (por exemplo, variáveis temporárias, pilhas de recursão, etc.).

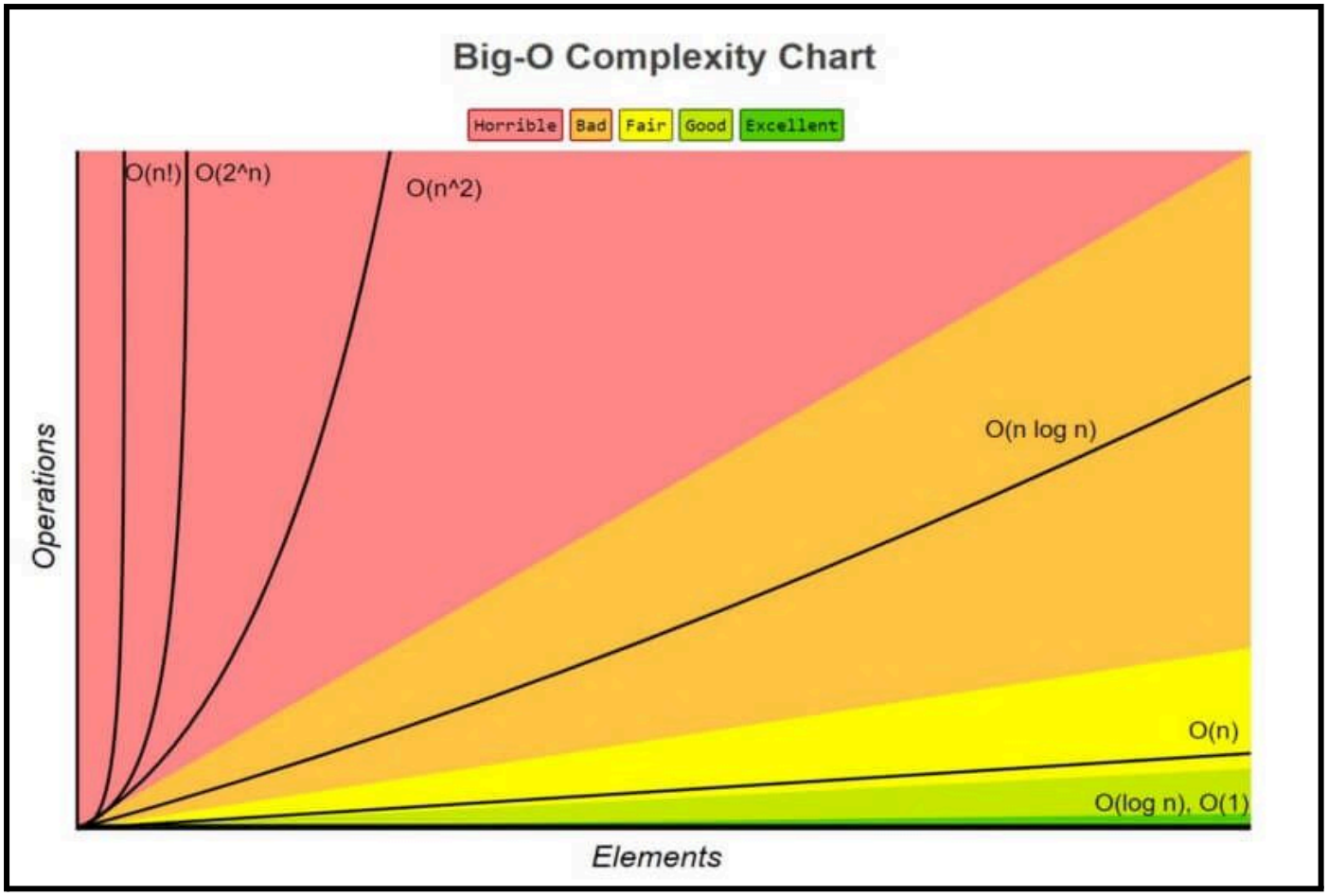
Por hora, é necessário apenas entender que, todo algoritmo possui seu espaço alocado na memória, e a escolha do melhor algoritmo varia de acordo com o *espaço disponível* x o *tempo esperado para a execução* do mesmo. A seguir, entenderemos a **Complexidade do Tempo.**

3. Complexidade de tempo

A determinação exata do tempo de execução de um algoritmo geralmente é complexa, sendo apenas estimada. A análise busca avaliar o tempo de execução de um algoritmo quando ele é executado sobre uma entrada (n) grande. Por exemplo, para uma função $6n^3 + 10n^2 + 3$, pode-se dizer que $f(n) = O(n^3)$.

As notações assintóticas (linguagem que permite analisar o tempo de execução de um algoritmo) são aplicadas a funções, como representações do tipo $O(n^2)$, para simplificar e eliminar detalhes menos relevantes dessas funções.

Usando a notação O, é possível descrever o tempo de execução de um algoritmo apenas inspecionando sua estrutura. Tecnicamente, afirmar que o tempo de execução de um algoritmo de ordenação por inserção é $O(n^2)$ para um dado 'n' pode ser considerado impreciso, pois o tempo de execução varia dependendo do tamanho da entrada 'n'. Em resumo, isso indica que o "*worst-case*" ou pior caso do tempo de execução é $O(n^2)$."



- Imagem 5.1

- Imagem importante para entender a qualidade dos algoritmos por meio da notação assintótica Big-O.

4. Definição de Algoritmos Computacionais

A definição de algoritmos segundo o livro “Algoritmos lógica para desenvolvimento de programação de computadores” o autor Manoel Manzano define da seguinte maneira “Algoritmos são conjuntos de passos finitos e organizados que, quando executados, resolvem um determinado problema”.

O livro “*Introduction to Algorithms*” (Introdução a Algoritmos) dos autores (Cormen, Leiserson, Rivest, Stein), coloca algoritmos como “Informalmente, um algoritmo é qualquer procedimento computacional bem definido que recebe algum valor, ou conjunto de valores, como entrada e produz algum valor, ou conjunto de valores, como saída. Assim, um algoritmo é uma sequência de passos computacionais que transformam a entrada na saída.” Ou seja, basicamente, o algoritmo dita exatamente como o computador deve agir para resolver um problema.

É importante a ênfase em “algoritmos computacionais” pois, algoritmos podem ser utilizados fora da computação, mas nosso estudo estará focado somente nos algoritmos computacionais.

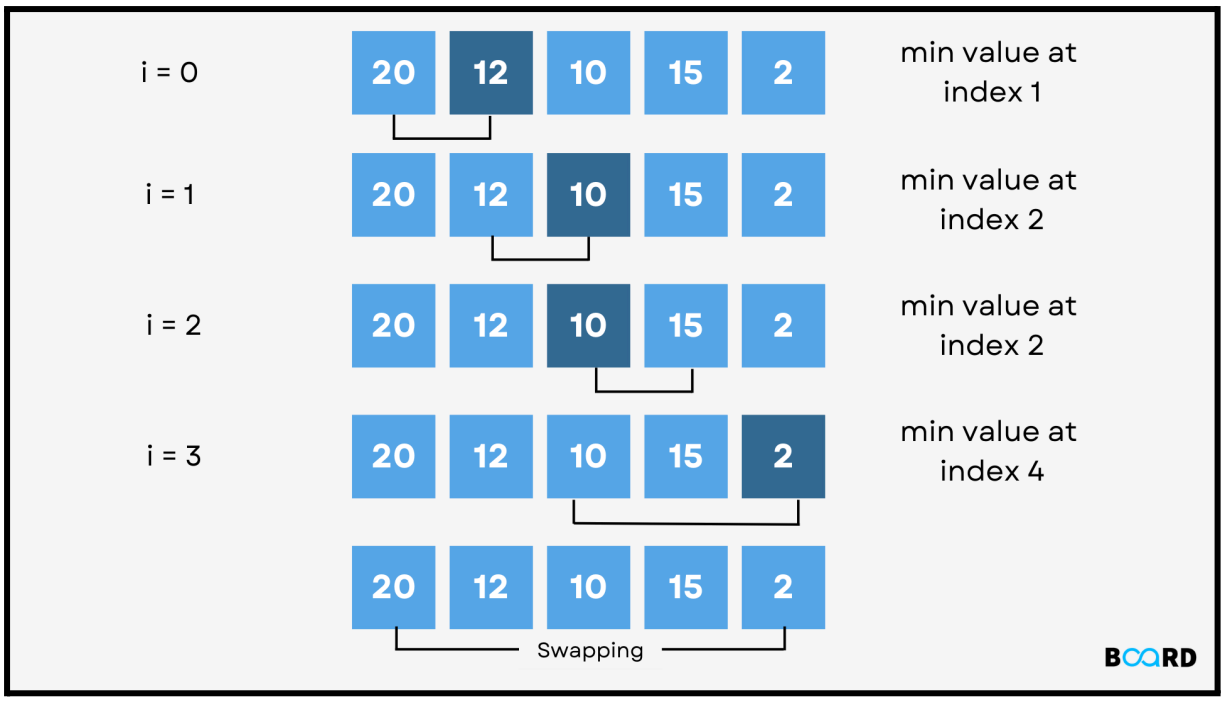
5. Algoritmos de ordenação

Algoritmos de ordenação são métodos usados para reorganizar elementos de uma *array* (lista) em uma ordem específica, como crescente ou decrescente. Eles são fundamentais na ciência da computação, pois a ordenação eficiente melhora a performance de outras operações, como busca e fusão de dados. Existem diversos algoritmos de ordenação, cada um com suas próprias vantagens e desvantagens em termos de complexidade de tempo e espaço.

5.1. Selection Sort

Selection Sort (Ordenação por seleção) é um algoritmo de ordenação que organiza uma lista dividindo-a em duas partes: uma sub-lista de itens já ordenados e uma sub-lista de itens não ordenados. Ele funciona encontrando o menor elemento da sub-lista não ordenada e trocando-o com o primeiro elemento dessa sub lista.

Este processo é repetido, movendo a fronteira entre as sub listas ordenada e não ordenada uma posição à direita, até que toda a lista esteja ordenada. É um algoritmo simples, mas não é eficiente para grandes listas, pois possui complexidade de tempo $O(n^2)$ onde n é o número de elementos.



- Imagem 6.1

Basicamente ele vai percorrer todas as casas de um array, comparando com o valor mínimo, caso seja encontrado um novo valor mínimo, ele substitui o índice com o mesmo.

Algoritmo de Selection Sort em Python:

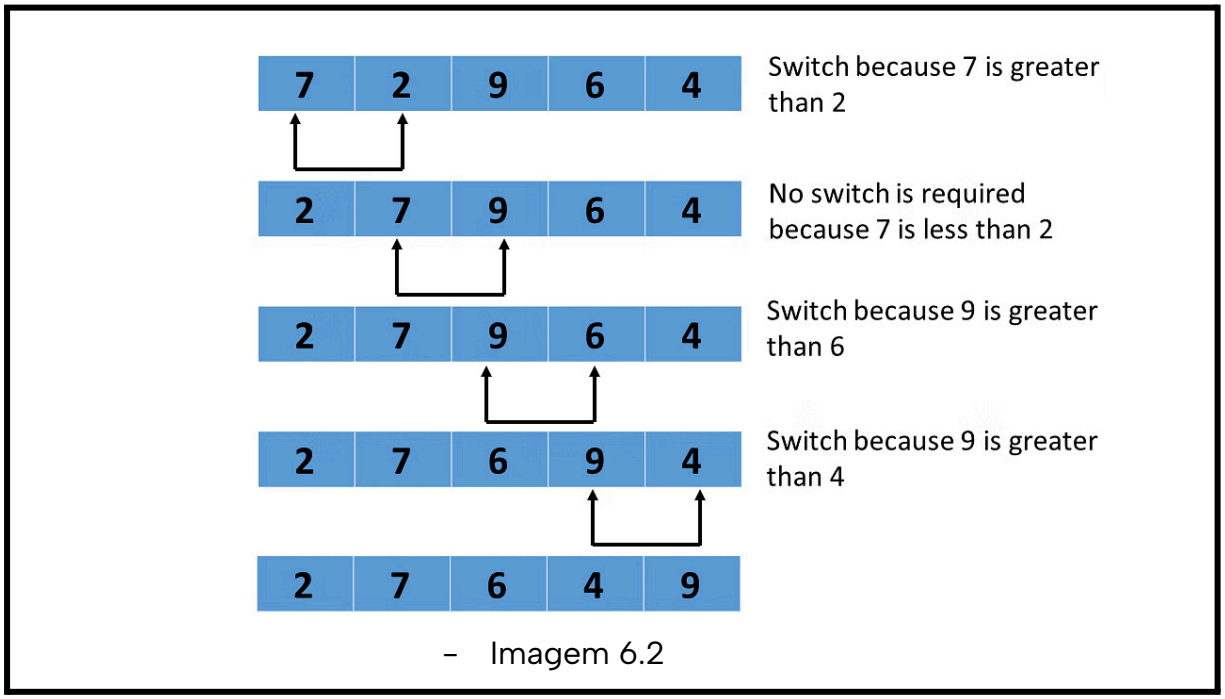
```python
# Selection Sort
def selection_sort(lista):
    tam_lista = len(lista)
    for j in range(tam_lista - 1):
        min_index = j
        for indice in range(j, tam_lista):
            if lista[indice] < lista[min_index]:
                min_index = indice
        if lista[j] > lista[min_index]: # Verfica se o item é menor que o mínimo
#           lista[j], lista[min_index] = lista[min_index], lista[j] # possível em python
            aux = lista[j] # Cria uma variável auxiliar para trocar os valores
            lista[j] = lista[min_index]
            lista[min_index] = aux
```

- Imagem 6.1.2

- Observando a função “for” dentro de outro “for” é possível identificar a notação O(n²) na complexidade do tempo, pois o pior caso possível seria ‘n’ sendo repetido ‘n’ vezes.

5.2. Bubble Sort

Bubble Sort (Ordenação por bolha) é um algoritmo de ordenação que percorre repetidamente a lista a ser ordenada, compara elementos juntando os itens em bolhas e os troca se necessário. Esse processo é repetido até que a lista esteja ordenada. A cada passagem pela lista, diminui o número de elementos a serem verificados nas passagens subsequentes. Embora seja fácil de entender e implementar, o Bubble Sort é ineficiente para listas grandes, pois possui uma complexidade de tempo O(n²) onde n é o número de elementos.



- Imagem 6.2

Algoritmo de Bubble Sort em Python:
foto

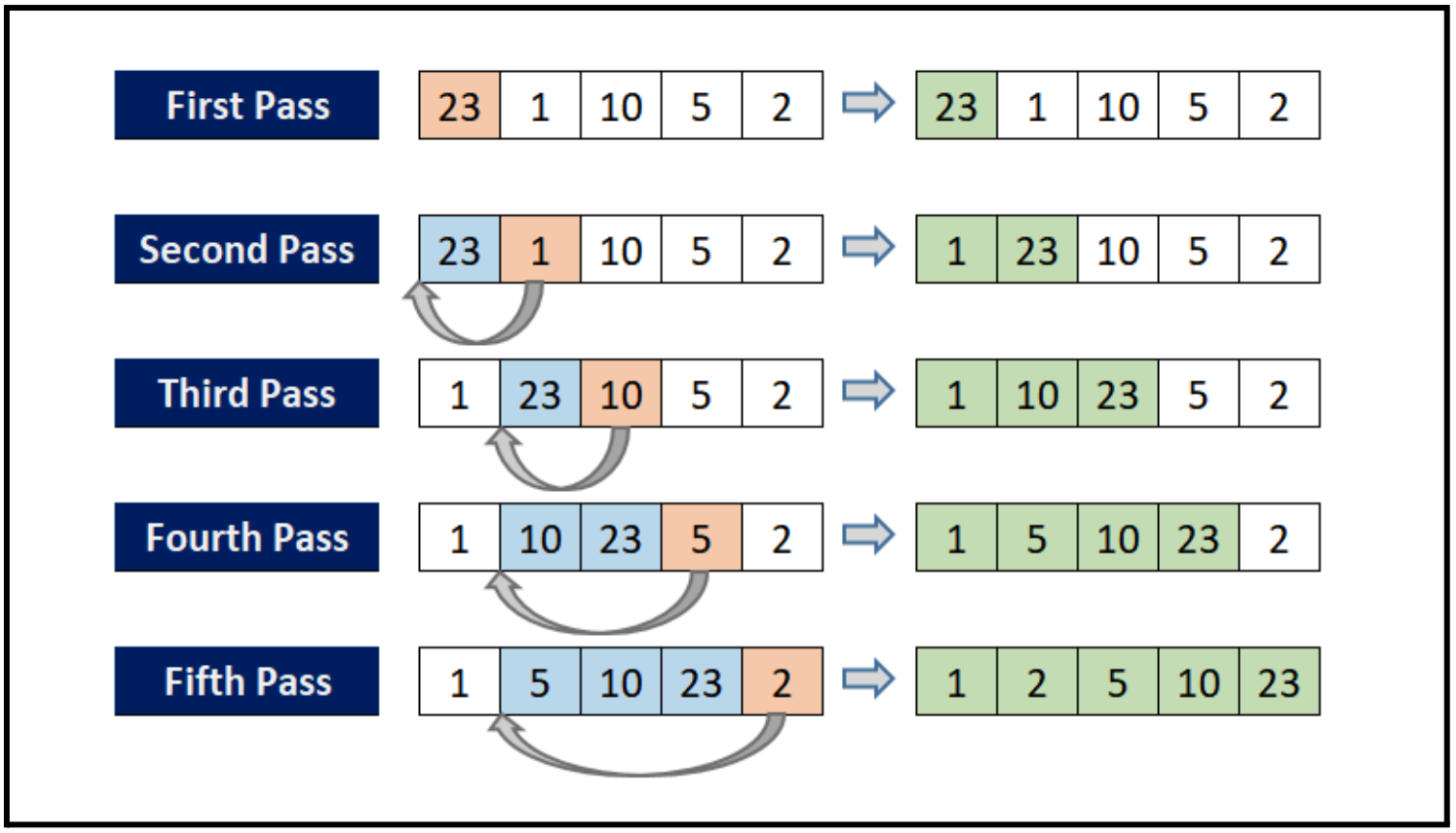
```python
# Bubble Sort
def bubble_sort(lista):
    tam_lista = len(lista)
    for j in range(tam_lista - 1):
        for indice in range (tam_lista - 1):
            if lista[indice] > lista[indice + 1]: # Verifica se o item a seguir é maior ou não que ele
                aux = lista[indice] # Cria uma variável auxiliar para trocar os valores
                lista[indice] = lista[indice + 1]
                lista[indice + 1] = aux
```

- Imagem 6.2.2

- Observando a função “for” dentro de outro “for” é possível identificar a notação O(n²) na complexidade do tempo, pois o pior caso possível seria ‘n’ sendo repetido ‘n’ vezes.

5.3. Insertion Sort

Insertion Sort (Ordenação por inserção) é um algoritmo de ordenação simples e eficiente para pequenos conjuntos de dados, que funciona construindo a lista ordenada um elemento de cada vez. Ele percorre a lista de elementos, e para cada elemento, insere-o na posição correta em relação aos elementos já ordenados. Isso é feito movendo os elementos maiores para a direita para abrir espaço para o elemento sendo inserido. A complexidade de tempo de O(n²), onde n é o número de elementos.



- Imagem 6.3

Algoritmo de Insertion Sort em Python:

```python
# Insertion Sort
def insertion_sort(lista):
    tam_lista = len(lista)
    for indice in range (1, tam_lista):
        chave = lista[indice]
        j = indice - 1 # Representação da parte ja ordenada da lista
        while j >= 0 and lista[j] > chave:
            lista[j+1] = lista[j] # Avança quem esta na sub-lista para a lista
            j = j - 1
        lista[j+1] = chave
```

- Imagem 6.3.2

- Observando a função “for” dentro de outro “for” é possível identificar a notação $O(n^2)$ na complexidade do tempo, pois o pior caso possível seria ‘n’ sendo repetido ‘n’ vezes.

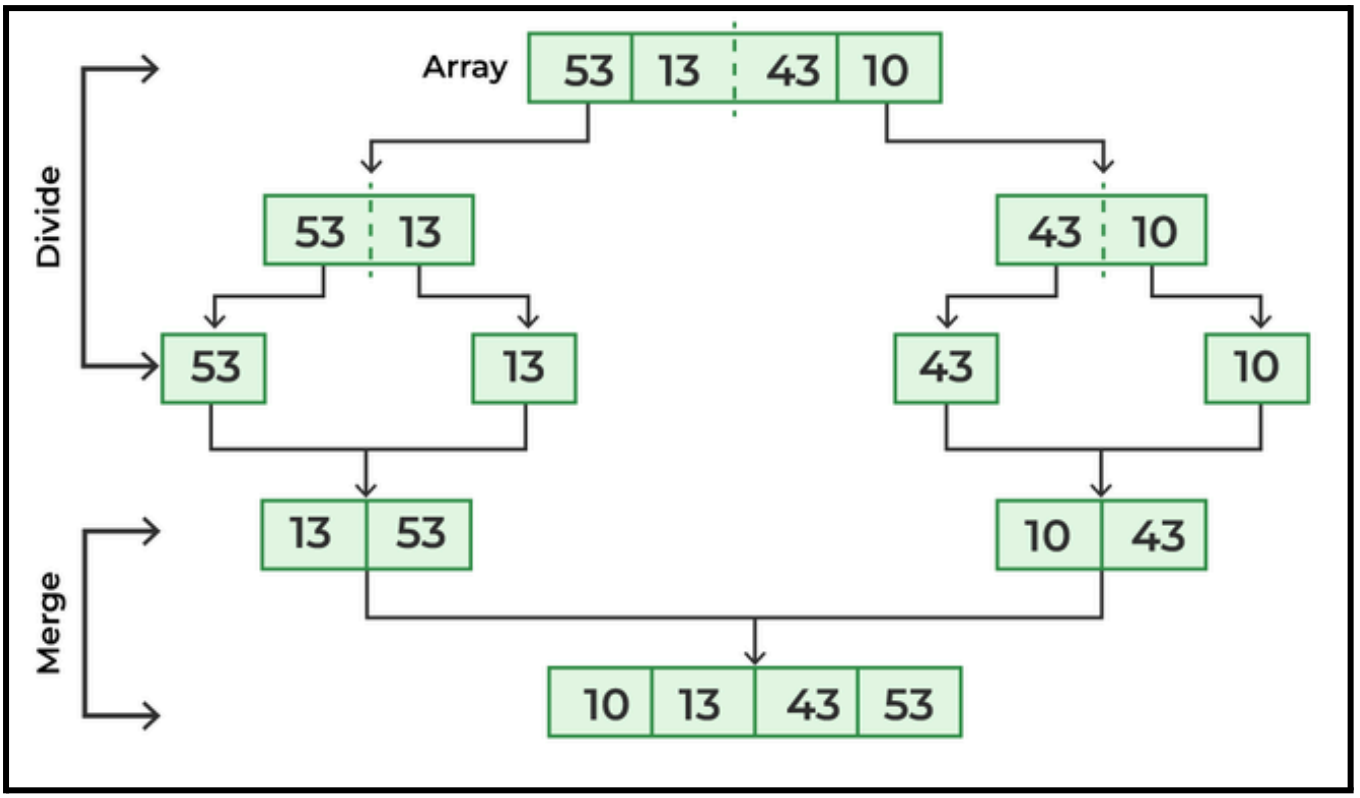
5.4. Divisor de Águas

Se você conseguiu entender e observar os algoritmos apresentados até agora, perceberá que todos seguem uma lógica muito semelhante e são considerados ineficientes para entradas grandes, ou seja, possuem baixa eficiência em certas situações. Mas, afinal, por que é importante conhecer as limitações desses algoritmos? A otimização de algoritmos é uma prática comum e fundamental para compreender os limites físicos do software e hardware. Por exemplo, em hardwares embarcados, onde a complexidade de espaço é limitada, utilizar algoritmos simples como o Insertion Sort pode ser uma boa solução devido à sua simplicidade e baixa demanda de memória.

A seguir vamos introduzir algoritmos de organização que possuem melhor complexidade de tempo, entretanto, exigem maior alocamento de memória.

5.5. Merge Sort

Merge Sort (Ordenação por junção) é um algoritmo de ordenação eficiente que utiliza a técnica de divisão e conquista (divide-and-conquer). Ele divide a lista de elementos em duas metades, ordena cada metade recursivamente e, em seguida, combina as duas metades ordenadas para produzir a lista final ordenada. A fase de combinação é feita de forma a garantir que a lista resultante esteja em ordem crescente. A complexidade de tempo do Merge Sort é O(n log n) em todos os casos, tornando-o mais eficiente que algoritmos simples como Bubble Sort e Insertion Sort, especialmente para listas grandes. No entanto, ele requer espaço adicional proporcional ao tamanho da lista para armazenar as sublistas temporárias. (Volte a imagem 5.1 e compare a complexidade de tempo entre Merge Sort e os outros algoritmos apresentados anteriormente).



- Imagem 6.5

Algoritmo de Merge Sort em Python:

```python
# Merge Sort
def merge_sort(lista, inicio = 0, fim = None):
    if fim is None:
        fim = len(lista)
    if fim - inicio > 1:
        meio = (inicio + fim)//2 # coloca um "meio" exato
        merge_sort(lista, inicio, meio) # faz um Merge Sort na lista da esquerda
        merge_sort(lista, meio, fim) # faz um Merge Sort na lista da direita
        merge(lista, inicio, meio, fim) # Junta as listas (esq e dir).
```

- Imagem 6.5.1

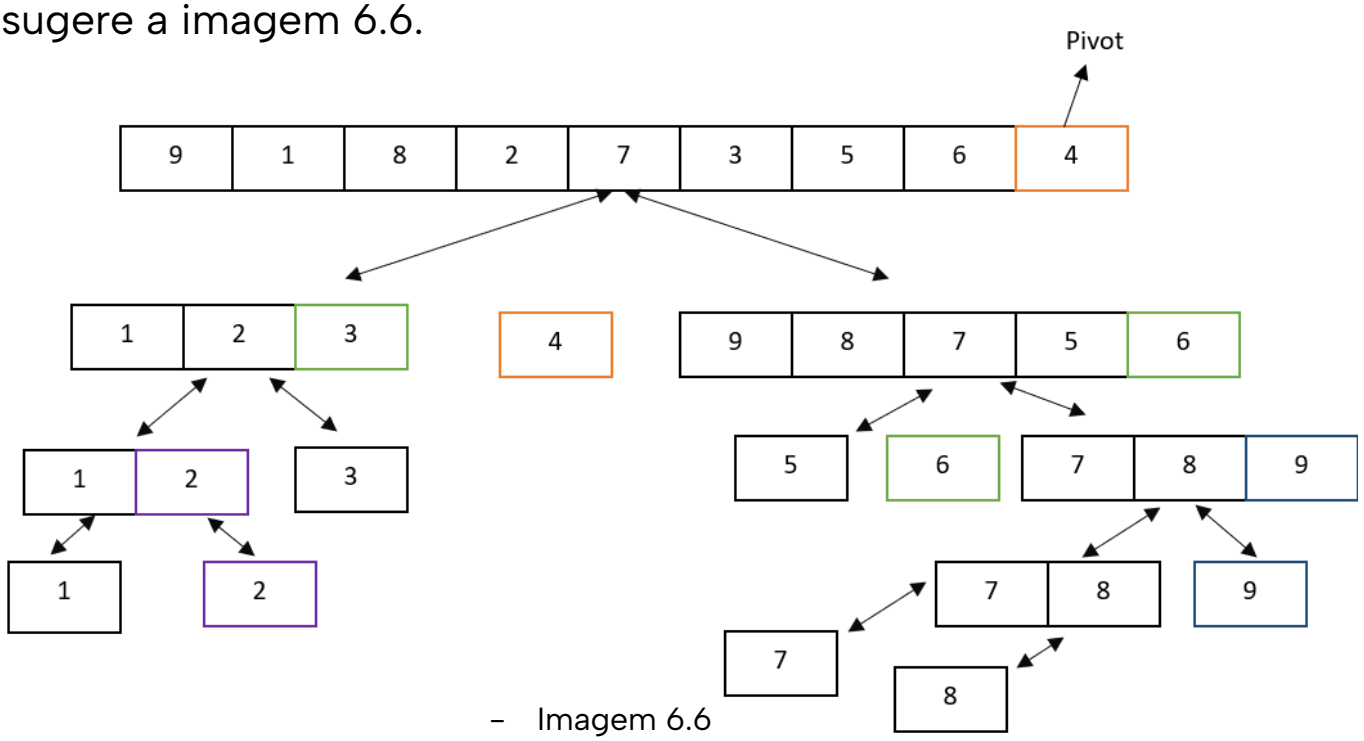
```python
def merge(lista,inicio,meio,fim):
    esquerda = lista[inicio:meio] # Notação do python para selecionar os itens da lista especificados
    direita = lista[meio:fim]
    top_left, top_right = 0, 0 # Organizam os topos das respectivas sub-listas
    for k in range (inicio,fim): # o indice 'k' gerencia o indice do topo da lista a ser organizada
        if top_left >= len(esquerda):
            lista[k] = direita[top_right]
            top_right = top_right + 1
        elif top_right >= len(direita):
            lista[k] = esquerda[top_left]
            top_left = top_left + 1
        elif esquerda[top_left] < direita[top_right]:
            lista[k] = esquerda[top_left]
            top_left = top_left + 1
        else:
            lista[k] = direita[top_right]
            top_right = top_right + 1
```

- Imagem 6.5.2

5.6. Quick Sort

Quick Sort (Ordenação rápida), indica ser um método rápido e eficiente. Ele escolhe um elemento como pivô e particiona (divide) a lista de elementos em duas sub-listas: uma contendo elementos menores que o pivô e outra com elementos maiores. Em seguida, ordena cada sub-lista recursivamente e combina todas para produzir a lista final ordenada. A fase de divisão é feita de forma que garanta que o pivô esteja na posição correta, com todos os elementos menores à sua esquerda e todos os maiores à sua direita. A complexidade de tempo do Quick Sort é O(n log n) na média, tornando-o mais eficiente que algoritmos simples como Bubble Sort e Insertion Sort, especialmente para listas grandes. No entanto, no pior caso, sua complexidade pode ser $O(n^2)$, embora este possa ser evitado com técnicas de escolha de pivô.

Nesta implementação, vamos colocar o pivot como sempre sendo o último item da lista a ser ordenada, como sugere a imagem 6.6.



– Imagem 6.6

Algoritmo de Quick Sort em Python:

```python
# Quick Sort
def quick_sort(lista, inicio = 0, fim = None):
    if fim is None:
        fim = len(lista) - 1
    if inicio < fim:
        p = partition(lista, inicio, fim) # 'p' é o pivô
        quick_sort(lista, inicio, p - 1) # Lista da esquerda, dos menores
        quick_sort(lista, p + 1, fim) # Lista da direita, dos maiores
```

- Imagem 6.6.1

```python
def partition(lista, inicio, fim):
    pivo = lista[fim]
    i = inicio
    for j in range(inicio, fim): # Passa por cada item da lista
        if lista[j] <= pivo:
            lista[j], lista[i] = lista[i], lista[j] # Troca de posições
            i = i + 1 # Incrementa o limite de menor que o pivô
    lista[i], lista[fim] = lista[fim], lista[i]
    return i
```

- Imagem 6.6.2

6. Resultados

Para alcançar os melhores resultados, foi necessário a utilização de um script que gere palavras e listas de palavras aleatórias, podendo ser alterado tanto o tamanho da palavra

quanto a quantidade de palavras contidas na lista. Para isso, foi utilizado o seguinte código.

```python
# Gera uma palavra aleatória
def randomWord(length = 10): # Tamanho 10 da palavra aleatória
    letra = string.ascii_lowercase
    return ''.join(random.choice(letra) for _ in range(length))

# Função para gerar uma lista de palavras aleatórias
def randomList(size = 1000): #Lista gera mil itens
    return [randomWord(random.randint(3, 1000)) for _ in range(size)]
```

- Imagem 7.1

E para a contagem e comparação dos algoritmos, foi necessário a utilização de um "cronômetro" computacional, ou seja, foi criado um código para que contasse quanto tempo cada algoritmo levou para ordenar a mesma lista de palavras.

```python
# Função para cronometrar o tempo de execução dos algoritmos de ordenação
def cronometro(sortFunction, array):
    inicio = time.time() # Inicio da contagem
    sortFunction(array.copy())
    fim = time.time() # Fim da contagem
    return fim - inicio # Diferença entre Fim x Inicio
```

- Imagem 7.2

A tabela a seguir mostra os resultados alcançados pelos algoritmos de ordenação com uma lista de 100 (cem) palavras, cada palavra com 10 letras.

ALGORITMO	TEMPO ESTIMADO
Selection Sort	0.00095 segundos
Bubble Sort	0.00283 segundos
Insertion Sort	0.00084 segundos
Merge Sort	0.00057 segundos
Quick Sort	0.00028 segundos

- Imagem 7.3

A tabela a seguir mostra o tempo de execução com uma entrada mediana, agora a entrada imposta pelo código será uma entrada de 1.000 (mil) de palavras, cada palavra com 10 letras.

ALGORITMO	TEMPO ESTIMADO
Selection Sort	0.08833 segundos
Bubble Sort	0.29778 segundos
Insertion Sort	0.09038 segundos
Merge Sort	0.00754 segundos
Quick Sort	0.00405 segundos

- Imagem 7.4

A tabela a seguir mostra o tempo de execução com uma entrada superior, agora a entrada imposta pelo código será uma entrada de 10.000 (dez mil) de palavras, cada palavra com 10 letras.

ALGORITMO	TEMPO ESTIMADO
Selection Sort	8.441 segundos
Bubble Sort	30.180 segundos
Insertion Sort	9.218 segundos
Merge Sort	0.090 segundos
Quick Sort	0.053 segundos

- Imagem 7.5

7. Conclusão

Este estudo analisou alguns algoritmos de ordenação, avaliando suas complexidades de tempo e espaço, bem como suas aplicações práticas. Os resultados obtidos demonstram claramente a importância de escolher o algoritmo correto dependendo do tamanho e da natureza dos dados a serem processados.

É evidente que, algoritmos de maior simplicidade na hora de implementar, são os mesmos algoritmos que levaram muito mais tempo para concluir suas tarefas quando a quantidade de palavras era elevada. Mas é importante ressaltar que não é porque são mais demorados, que são ineficazes. Ao observar a imagem 7.3, é possível visualizar que o maior tempo de ordenação foi de quase 3 (três) milésimos de segundo, ou seja,

se o fluxo de entrada for baixo, uma implementação simples consegue resolver de maneira eficiente.

Mas é importante visualizar a grande discrepância conforme a entrada do algoritmo aumenta. Ao observar a imagem 7.5 é possível ver que, os algoritmos em verde claro possuem tempo total muito inferior aos algoritmos em amarelo, sendo o tempo (dos de amarelo) mais de 100 (cem) vezes maior se comparado com os algoritmos em verde.

Por fim, espero ter ajudado a compreender melhor os algoritmos de ordenação, o que são, como são implementados e qual a maior diferença entre todos os cinco algoritmos abordados neste estudo. Ainda há uma infinidade de outros algoritmos, mas esses são os mais conhecidos, tanto pela sua facilidade em entendimento, quanto pela facilidade ao implementar.

8. Links

- https://www.kaggle.com/code/henriquesogayar/algoritmos

- https://github.com/Sogayar/algoritmos

9. Bibliografia

[1] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

[2] MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. *Algoritmos: lógica para desenvolvimento de programação de computadores*. São Paulo: Érica, 2016.

[3] Szwarcfiter, Jayme Luiz; Markenzon, Lilian. Estruturas de Dados e Seus Algoritmos. 3ª ed. Rio de Janeiro: LTC, 2010.