

Leonardo R. Orabona

_Algoritmos e Estrutura de Dados

UNIDADE II



Sumário

1. Algoritmos de ordenação	3
2. Selection Sort.....	4
3. Insertion Sort	7
4. Bubble Sort.....	10
5. Merge Sort.....	13
6. Quick Sort.....	16

Algoritmos de ordenação



Algoritmos de ordenação são usados para organizar um conjunto de elementos em determinada ordem, geralmente em ordem crescente ou decrescente.

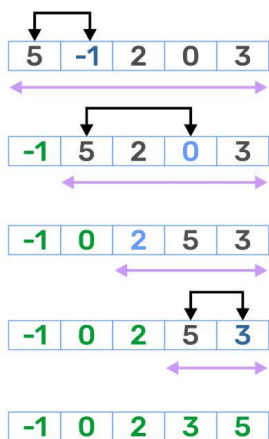
- **Selection Sort:** Seleciona repetidamente o menor (ou maior) elemento e o coloca na posição correta.
- **Insertion Sort:** insere elementos em uma posição ordenada iterativamente.
- **Bubble Sort:** Simples, mas ineficiente para grandes conjuntos de dados.
- **Merge Sort:** Baseado na técnica de dividir para conquistar, eficiente e estável.
- **Quick Sort:** Escolhe um pivô e divide os elementos em menores e maiores que ele, sendo um dos mais eficientes.

Selection Sort



O algoritmo de seleção encontra o menor elemento e o move para a posição correta, repetindo esse processo para todos os elementos restantes. Tem uma complexidade de tempo médio de $O(n^2)$.

Selection Sort



Green = Sorted

Blue = Current minimum

Find minimum elements in unsorted array and swap if required (element not at correct location already).

É um algoritmo de ordenação simples, mas ineficiente para grandes conjuntos de dados. Ele funciona encontrando o menor elemento de um *array* e trocando-o com o elemento na primeira posição, depois encontra o segundo menor e troca com a segunda posição, e assim por diante, até que o *array* esteja ordenado.

Funcionamento do Selection Sort

- Percorre o *array* em busca do menor elemento.
- Troca esse menor elemento com o primeiro elemento do *array* (se necessário).
- Repete o processo para os próximos elementos, considerando a parte não ordenada do *array*.
- Continua até que todos os elementos estejam ordenados.

Exemplo de funcionamento

Dado o *array*:

[5, 3, 8, 4, 2]

Passo 1: encontra o menor elemento (2) e troca com o primeiro elemento:

[2, 3, 8, 4, 5]

Passo 2: encontra o menor elemento na parte restante (3), já está na posição correta.

Passo 3: encontra o menor entre [8, 4, 5] (é 4) e troca com 8:

[2, 3, 4, 8, 5]

Passo 4: encontra o menor entre [8, 5] (é 5) e troca com 8:

[2, 3, 4, 5, 8]

Agora, o *array* está ordenado!

Complexidade de tempo

O Selection Sort sempre percorre o *array* inteiro para encontrar o menor elemento, independentemente da ordenação inicial. Assim, ele tem uma complexidade de tempo de:

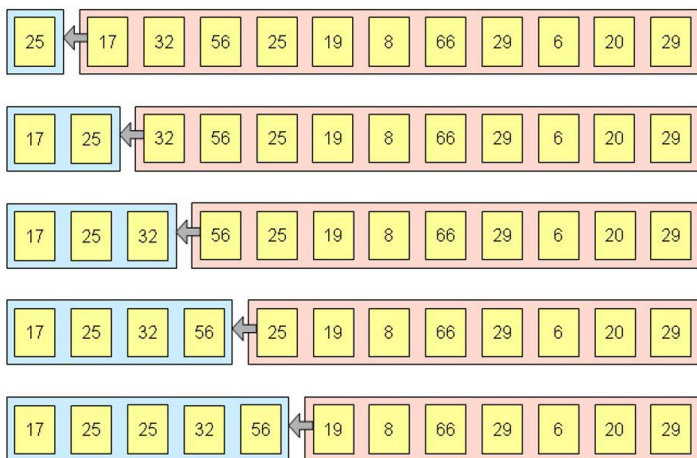
- **Melhor caso:** $O(n^2)$ (quando o *array* já está ordenado).
- **Caso médio:** $O(n^2)$.
- **Pior caso:** $O(n^2)$ (quando o *array* está ordenado de forma inversa).

3

Insertion Sort



Esse algoritmo constrói a lista ordenada inserindo cada elemento, um de cada vez, na posição correta dessa lista. Sua complexidade de tempo médio é $O(n^2)$.



Fonte: elaborada pelo autor.

Funcionamento do Insertion Sort

- Considera o primeiro elemento como ordenado.
- Pega o próximo elemento e o compara com os elementos da parte ordenada, deslocando os maiores para a direita.
- Insere o elemento na posição correta.
- Repete o processo para todos os elementos do *array*.

Exemplo de funcionamento

Dado o *array*:

[5, 3, 8, 4, 2]

Passo 1: o primeiro elemento 5 já está ordenado.

Passo 2: o 3 é comparado com 5 e inserido antes dele.

[3, 5, 8, 4, 2]

Passo 3: o 8 já está na posição correta.

Passo 4: o 4 é comparado com 8 e 5, e inserido na posição correta.

[3, 4, 5, 8, 2]

Passo 5: o 2 é comparado com todos os elementos anteriores e inserido na posição correta.

[2, 3, 4, 5, 8]

Agora, o *array* está ordenado!

Complexidade de tempo

O Insertion Sort tem um desempenho que depende do estado inicial dos dados:

- **Melhor caso:** $O(n^2)$ (quando o *array* já está ordenado)
- **Caso médio:** $O(n^2)$
- **Pior caso:** $O(n^2)$ (quando o *array* está em ordem reversa)

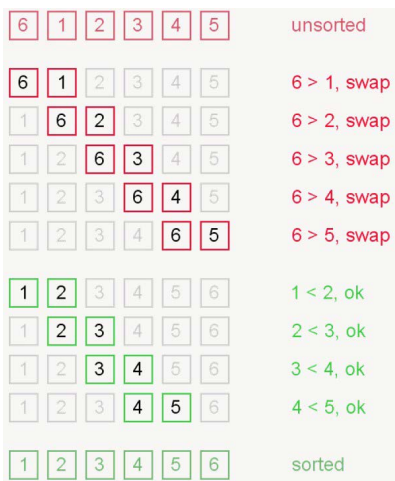
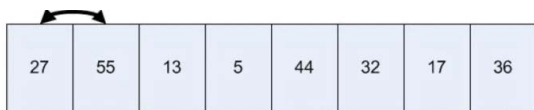
Como ele insere elementos um a um, pode ser mais eficiente que Selection Sort para *arrays* **parcialmente ordenados**.

4

Bubble Sort



Esse é um algoritmo simples que compara elementos adjacentes e os troca se estiverem fora de ordem. É eficaz para conjuntos pequenos, mas não é eficiente para conjuntos grandes, com uma complexidade de tempo médio de $O(n^2)$.



Fonte: elaborada pelo autor.

É um algoritmo de ordenação simples que compara elementos adjacentes e os troca se estiverem fora de ordem. Esse processo se repete até que toda a lista esteja ordenada. O nome "Bubble" vem do fato de que os elementos maiores "flutuam" para o topo da lista a cada iteração.

Funcionamento do Bubble Sort

- Percorre o *array*, comparando pares de elementos adjacentes.
- Se os elementos estiverem fora de ordem, eles são trocados.
- No final de cada passagem, o maior elemento estará na sua posição correta.
- O processo se repete para os elementos restantes até que não haja mais trocas.

Exemplo de funcionamento

Dado o *array*:

[5, 3, 8, 4, 2]

Passo 1: comparar e trocar onde necessário:

[3, 5, 8, 4, 2] → [3, 5, 8, 4, 2] → [3, 5, 4, 8, 2] → [3, 5, 4, 2, 8]

Passo 2: repetir para os elementos restantes:

[3, 5, 4, 2, 8] → [3, 4, 5, 2, 8] → [3, 4, 2, 5, 8]

Passo 3: continuar o processo até que o *array* esteja ordenado:

[3, 2, 4, 5, 8] → [2, 3, 4, 5, 8]

Complexidade de tempo

O Bubble Sort **sempre percorre o *array* múltiplas vezes**, mesmo que já esteja ordenado.

- **Melhor caso:** $O(n^2)$ (quando o *array* já está ordenado e usa uma versão otimizada com "*flag*" para detectar ordenação).
- **Caso médio:** $O(n^2)$.
- **Pior caso:** $O(n^2)$ (quando o *array* está em ordem inversa).

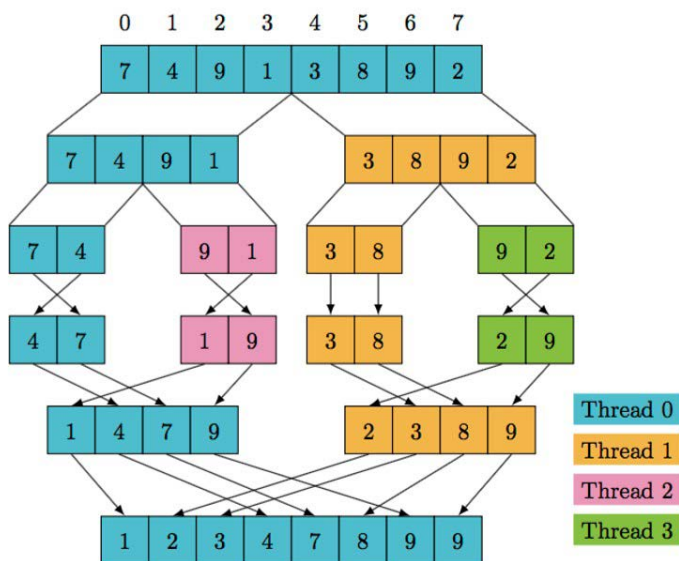
Como o número de comparações e trocas é muito alto, esse algoritmo não é eficiente para grandes conjuntos de dados.

5

Merge Sort






Um algoritmo de ordenação do tipo dividir para conquistar, que separa o conjunto em duas metades, ordena cada uma e, em seguida, as mescla para obter a lista final ordenada. Sua complexidade de tempo é $O(n \log n)$.



Fonte: elaborada pelo autor.

Um algoritmo de ordenação dividir e conquistar que divide o conjunto em duas metades, ordena cada metade e, em seguida, mescla as duas metades ordenadas para obter a lista ordenada final. Possui uma complexidade de tempo de $O(n \log n)$.

Funcionamento do Merge Sort

-  **Dividir:** o *array* é dividido recursivamente ao meio até que cada sublista contenha um único elemento (ou esteja vazia).
-  **Conquistar:** cada par de sublistas é mesclado em ordem crescente.
-  **Combinar:** as sublistas ordenadas são unidas até formar a lista final ordenada.

Exemplo de funcionamento

Dado o *array*:

[5, 3, 8, 4, 2, 7, 6, 1]

-  Divisão recursiva

[5, 3, 8, 4] | [2, 7, 6, 1]

[5, 3] [8, 4] | [2, 7] [6, 1]

[5] [3] [8] [4] | [2] [7] [6] [1]

Ordenação e mesclagem

[3, 5] [4, 8] | [2, 7] [1, 6]

[3, 4, 5, 8] | [1, 2, 6, 7]

[1, 2, 3, 4, 5, 6, 7, 8]

Agora, o *array* está ordenado!

Complexidade de Tempo

O Merge Sort **sempre** executa a mesma quantidade de divisões e mesclagens, independentemente da ordem inicial dos elementos.

➤ **Melhor caso:** $O(n \log n)$.

➤ **Caso médio:** $O(n \log n)$.

➤ **Pior caso:** $O(n \log n)$.

Essa eficiência faz com que o Merge Sort seja **mais rápido que algoritmos $O(n^2)$ como Bubble Sort, Insertion Sort e Selection Sort** para grandes conjuntos de dados.

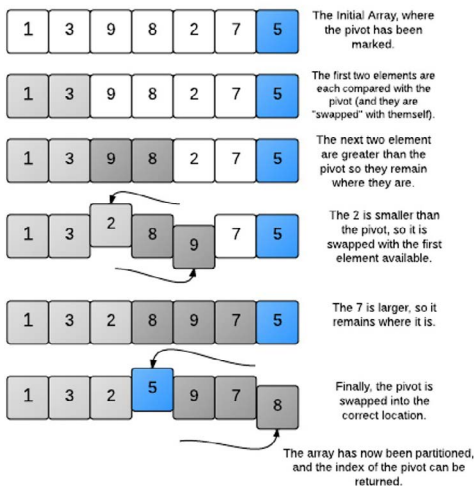
6

Quick Sort



Outro algoritmo de ordenação dividir e conquistar que seleciona um “pivô” e rearranja os elementos de modo que os elementos menores que o pivô estejam à esquerda e os elementos maiores estejam à direita. Ele é muito eficiente em média, com uma complexidade de tempo médio de $O(n \log n)$.

Partitioning an array



Fonte: elaborada pelo autor.

É um algoritmo de ordenação baseado na estratégia Dividir para Conquistar (*Divide and Conquer*). Ele seleciona um pivô, reorganiza os elementos em torno desse pivô e aplica recursivamente o mesmo processo às sublistas resultantes.

Funcionamento do Quick Sort

- **Escolha do pivô:** um elemento do *array* é escolhido como pivô (pode ser o primeiro, o último, um aleatório ou o mediano).
- **Particionamento:** os elementos menores que o pivô são movidos para a esquerda e os maiores para a direita.
- **Recursão:** o processo é repetido para as sublistas à esquerda e à direita do pivô.
- **Combinação:** como a ordenação acontece *in-place*, não há necessidade de mesclar os *subarrays* como no Merge Sort.

Exemplo de funcionamento

Dado o *array*:

[5, 3, 8, 4, 2, 7, 6, 1]

- Escolha do pivô (exemplo: último elemento, 1)
 - Reorganizamos os elementos: [1, 3, 8, 4, 2, 7, 6, 5]
 - O pivô 1 está na posição correta.
- Recursão nas sublistas:
 - Sublista à esquerda: [] (vazia)
 - Sublista à direita: [3, 8, 4, 2, 7, 6, 5]
- Escolha do pivô (exemplo: 5)
 - Reorganizamos os elementos: [3, 4, 2, 5, 8, 7, 6]
- Repetição até ordenação completa:
 - [2, 3, 4, 5, 6, 7, 8]

Complexidade de tempo

- O desempenho do Quick Sort depende da escolha do pivô:
- Melhor caso:** $O(n \log n)$ (quando o pivô divide o *array* de forma equilibrada).
- Caso médio:** $O(n \log n)$ (ocorre na maioria dos casos).

Pior caso: $O(n^2)$ (quando o pivô escolhido é sempre o menor ou maior elemento, como em um *array* já ordenado).

Para evitar o pior caso, técnicas como **pivô aleatório** ou **mediana de três** são usadas.

