UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM COMPUTAÇÃO

ALUNA: NATÁLIA MARIA DE ARAÚJO LIMA

Relatório Comparativo:
Algoritmos: Bubble Sort; Selection Sort; Insertion Sort;

Algoritmos de ordenação são fundamentais na ciência da computação, e entre os diversos algoritmos existentes, o *Bubble Sort*, o *Selection Sort* e o *Insertion Sort* se destacam por sua simplicidade e eficácia em conjuntos de dados pequenos ou condições específicas. E Embora cada um desses algoritmos tenha sua própria mecânica e características, eles compartilham algumas propriedades, como operar *in-place* (ou seja, sem necessidade de memória adicional significativa) e ter *complexidade quadrática* em casos médios e piores *(O(n2))*. No entanto, diferenças cruciais em seu desempenho surgem no número de trocas e comparações necessárias, bem como na eficiência geral, dependendo do estado inicial dos dados a serem ordenados.

Bubble Sort:

Características:

- *Algoritmo Simples:* Considerado um dos algoritmos de ordenação mais simples de entender e implementar.
- *Percorre a Lista Várias Vezes:* Examina repetidamente a lista a ser ordenada, compara cada par de elementos adjacentes e os troca se estiverem na ordem errada.
- *"Flutua" os Elementos Maiores:* A cada passagem pelo array, o maior elemento "flutua" até sua posição correta no final da lista.
- *Comparação de Elementos Adjacentes:* Compara elementos dois a dois, fazendo ajustes (trocas) conforme necessário para ordenar a lista.
- *Eficiência:* Devido à sua natureza O(n^2) para complexidade de tempo no pior caso, não é adequado para grandes datasets.
- *In place*: Não precisa de espaço extra.
- *Algoritmo Stable*: Preserva a ordem.
- *Tempo de execução:* Sua complexidade de tempo médio e pior caso é $O(n^2)$, onde n é o número de elementos no array. É um dos algoritmos de ordenação mais lentos para listas grandes devido ao número de comparações e trocas.

Exemplo:

array: [8,4,7,1]
trocas: [8,4,7,1] → [8,4,7,1] → [4,8,7,1] → [4,8,7,1] → [4,7,8,1] → [4,7,8,1] → [4,7,1,8] → [4,7,1,8] → [4,7,1,8] → [4,7,1,8] → [4,1,7,8] → [4,1,7,8] → [4,1,7,8] → [1,4,7,8]

- ➔ O algoritmo começa comparando o primeiro número com os demais.
- ➔ A cada dois itens, se o primeiro é maior que o segundo, o algoritmo inverte suas posições
- ➔ A operação de “flutuação” segue até que o maior fique no final.
- ➔ Assim que o maior chega no último índice, a comparação recomeça do primeiro elemento, dispensando o último.
- ➔ Novas “flutuações” acontecem, sem que o último participe da iteração.
- ➔ Não é necessário comparar com o último, pois o maior já estará posicionado no final.
- ➔ Assim que o maior chega no último índice, a comparação recomeça do primeiro

elemento, dispensando o último.

➔ Quando restam apenas dois, e eles estiverem em ordem, a ordenação está concluída.

Pseudo-código:

```
   n ← tamanho(lista)
   Para i de 0 até n - 2 faça:
      Para j de 0 até n - i - 2 faça:
         Se lista[j] > lista[j + 1] então:
            // Troca lista[j] com lista[j + 1]
            temp ← lista[j]
            lista[j] ← lista[j + 1]
            lista[j + 1] ← temp
         fim-se
      fim-para
   fim-para
fim-procedimento

// Código principal
lista ← [8, 4, 7, 1] // Array exemplo
Chama BubbleSort(lista)
Imprime lista
```

Selection Sort:

Características:

- *Seleciona o Menor Elemento:* Em cada iteração, seleciona o menor elemento do array não ordenado e o coloca na sua posição correta na parte já ordenada.
- *Divisão em Partes Ordenada e Não Ordenada:* Divide a lista em duas partes: uma com elementos já ordenados e outra com os restantes a serem ordenados.
- *Eficiência Consistente:* Tem complexidade de tempo O(n^2) em todos os casos (melhor, médio e pior), tornando-o menos eficiente para listas grandes.
- *In-place e Unstable:* Opera diretamente sobre o array de entrada sem utilizar espaço extra significativo, mas não é considerado estável, pois pode alterar a ordem de registros com chaves iguais.
- *Simples de Implementar:* Assim como o Bubble Sort, é fácil de entender e implementar, embora não seja o mais eficiente para listas grandes.
- *Tempo de execução:* Também tem complexidade de tempo O(n²) para todos os casos (melhor, médio e pior), similar ao Bubble Sort, mas geralmente realiza menos trocas.

Exemplo:

array: [8,4,7,1]
trocas: [8,4,7,1] → [8,4,7,1] → [1,4,7,8] → [1,4,7,8] → [1,4,7,8] → [1,4,7,8] → [1,4,7,8] →

[1,4,7,8] → [1,4,7,8]

- ➔ O algoritmo inicia procurando o menor número
- ➔ O menor número é trocado com o primeiro, que não será mais removido.
- ➔ O algoritmo continua procurando o menor número da subsequência restante.
- ➔ Se ele já é o primeiro da subsequência restante, ele já está na posição correta.
- ➔ Assim é feito para todos os demais, até chegar no último, e a lista estará ordenada.

Pseudo-código:

```
    Para i de 0 até tamanho(lista) - 2 faça:
        menor ← i
        Para j de i + 1 até tamanho(lista) - 1 faça:
            Se lista[j] < lista[menor] então:
                menor ← j
            fim-se
        fim-para
        // Troca lista[i] com lista[menor]
        temp ← lista[i]
        lista[i] ← lista[menor]
        lista[menor] ← temp
    fim-para
fim-procedimento

// Código principal
lista ← [8, 4, 7, 1] // Array exemplo
Chama SelectionSort(lista)
Imprime lista
```

Insertion Sort:

Características:

- *Simples e Intuitivo:* Assim como organizar cartas em mãos, o Insertion Sort constrói a lista ordenada um elemento por vez.
- *Constrói Sequência Final Gradualmente:* Divide a lista em duas partes: uma ordenada e outra não ordenada. Insere um elemento da parte não ordenada na posição correta da parte ordenada em cada iteração.
- *Eficiência para Listas Pequenas*: Muito eficiente para listas pequenas ou quase ordenadas, embora tenha complexidade O(n^2) no pior caso.
- *Algoritmo Stable:* Mantém a ordem relativa de registros com chaves iguais, tornando-se um método de ordenação estável.
- *In-place:* Opera diretamente no array de entrada, utilizando um pequeno espaço adicional fixo.
- *Adaptativo:* Melhora sua eficiência para listas que já estão parcialmente ordenadas.
- *Tempo de execução:* A complexidade de tempo é O(n²) no pior caso, mas tende a

ser mais rápido que Bubble e Selection Sort para pequenas listas ou listas que já estão parcialmente ordenadas, devido à sua natureza adaptativa.

Exemplo:

array: [8,4,7,1]
trocas: [8,4,7,1] → [8,4,7,1] → [4,8,7,1] → [4,8,7,1] → [4,7,8,1] → [4,7,8,1] → [1,4,7,8]

- ➔ O algoritmo compara o segundo elemento (4) com o primeiro (8). Como 4 é menor que 8, eles são trocados de lugar.
- ➔ O algoritmo considera o terceiro elemento (7) e o compara com os elementos anteriores (8 e 4). Como 7 é menor que 8 e maior que 4, 7 é inserido entre 4 e 8.
- ➔ O algoritmo olha para o último elemento (1) e o compara com todos os elementos anteriores (8, 7, e 4). Como 1 é menor que todos eles, ele é movido para a posição inicial do array.
- ➔ O array é ordenado de forma crescente como [1, 4, 7, 8]. Em cada passo, o algoritmo "insere" o elemento em consideração na posição correta entre os elementos já analisados, garantindo que a parte do array até o ponto atual esteja sempre ordenada.

Pseudo código:

```
   Para i de 1 até tamanho(lista) - 1 faça:
      chave ← lista[i]
      j ← i
      Enquanto j > 0 e lista[j-1] > chave faça:
         lista[j] ← lista[j-1]
         j ← j - 1
      fim-enquanto
      lista[j] ← chave
   fim-para
fim-procedimento

// Código principal
lista ← [8, 4, 7, 1] // Array exemplo
Chama InsertionSort(lista)
Imprime lista
```