Análise Comparativa de Métodos de Ordenação

Heitor C.V. Moreira

[1]Instituto de Informatica – Pontifícia Universidade Catolica de Minas Gerais
Belo Horizonte – MG – Brasil

Abstract. *This work aims to analyze and compare 4 sorting methods based on the number of movements, comparisons and time in milliseconds. The four algorithms used are* ***Selection Sort***, ***Insertion Sort***, ***Bubble Sort*** *and* ***QuickSort****.*

Resumo. *Este trabalho tem como objetivo analisar e comparar 4 métodos de ordenação utilizando como base a quantidade de movimentações, de comparações e tempo em milissegundos. Os quatro algoritmos usados são* ***Selection Sort***, ***Insertion Sort***, ***Bubble Sort*** *e* ***QuickSort****.*

1. Informações do Ambiente

1.1. Informações do computador

```
Macbook Air
Chip - Apple M1
Memory - 8gb
macOs - 15.1.1 (24B91)
```

1.2. Java Version

```
$java -version
openjdk version "21.0.3" 2024-04-16 LTS
OpenJDK Runtime Environment Corretto-21.0.3.9.1 (build 21.0.3+9-LTS)
OpenJDK 64-Bit Server VM Corretto-21.0.3.9.1 (build 21.0.3+9-LTS, mixed mode, sharing)
```

2. Métodos de Ordenação

Funções auxiliares

```
private static boolean isSmaller(int a, int b){
    comp += 1;
    return a < b;
}
private static void swap(int array[], int i, int j) {
    mov+=3;
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

2.1. Selection Sort

Comportamento: O algoritmo encontra o menor elemento da sublista não ordenada e o troca com o primeiro elemento não ordenado à esquerda. Após a troca, os limites da sublista ordenada são movidos para a direita.

Caso	Comparações	Movimentações	Espaço
Melhor	$O(n^2)$	$O(1)$	$O(1)$
Pior	$O(n^2)$	$O(n)$	$O(1)$

Table 1. Tabela de complexidade

Chegando em seu pior caso quando nenhum elemento já está previamente na posição correta

Implementação Selection Sort

```
1    private static void selectionSort(int array[]){
2        for(int i = 0; i<array.length-1; i++){
3            int min = i;
4            for(int j = i+1; j<array.length; j++){
5                if(isSmaller(array[j], array[min])){
6                    min = j;
7                }
8            }
9            swap(array, min, i);
10       }
11   }
```

2.2. Insertion Sort

Comportamento: O algoritmo insere cada elemento na posição correta à medida que percorre a lista, movendo os elementos maiores para a direita até encontrar o local adequado para o novo elemento.

Caso	Comparações	Movimentações	Espaço
Melhor	$O(n)$	$O(1)$	$O(1)$
Pior	$O(n^2)$	$O(n^2)$	$O(1)$

Table 2. Tabela de complexidade

O pior caso ocorre quando a lista está em ordem decrescente (inverso à ordem desejada).

Implementação Insertion Sort

```
    private static void insertionSort(int array[]){
        for (int i = 1; i < array.length; i++) {
            int key = array[i];
            mov++;
            int j = i - 1;
            while (j >= 0 && isSmaller(key, array[j])) {
                array[j + 1] = array[j];
                mov++;
                j--;
            }
            array[j + 1] = key;
            mov++;
        }
    }
```

2.3. Bubble Sort

Comportamento: O algoritmo compara elementos adjacentes e os troca de lugar caso estejam na ordem errada. Esse processo é repetido até que a lista esteja ordenada, fazendo com que os maiores elementos ”subam” para o final da lista em cada iteração.

Caso	Comparações	Movimentações	Espaço
Melhor	$O(n)$	$O(1)$	$O(1)$
Pior	$O(n^2)$	$O(n^2)$	$O(1)$

Table 3. Tabela de complexidade

Implementação Bubble Sort

```
    private static void bubbleSort(int array[]){
        boolean swaped = true;
        for(int i = 0; i<array.length-1 && swaped; i++){
            swaped = false;
            for(int j = 0; j<array.length-1-i; j++){
                if(isSmaller(array[j+1], array[j])){
                    swap(array, j, j+1);
                    swaped=true;
                }
            }
        }
    }
```

2.4. Quick Sort

Comportamento: O algoritmo escolhe um pivô e particiona a lista em duas sublistas, uma com elementos menores que o pivô e outra com elementos maiores. Esse processo é recursivamente repetido nas sublistas até que a lista esteja completamente ordenada.

Caso	Comparações	Movimentações	Espaço
Melhor	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Pior	$O(n^2)$	$O(n^2)$	$O(n)$

Table 4. Tabela de complexidade

Chegando em seu pior caso quando o pivô é sempre o menor ou o maior elemento da sublista

Implementação Quick Sort

```
    private static void quickSort(int array[], int left, int right){
        int middle = left+(right-left)/2;
        int i = left, j = right;
        while(i <= j){
            while (isSmaller(array[i], array[middle]))  i++;
            while (isSmaller(array[middle], array[j])) j--;
            if(i <= j){
                swap(array, i, j);
                i++;j--;
            }
        }
        if(left < j) quickSort(array, left, j);
        if(i < right) quickSort(array, i, right);
    }
    private static void quickSort(int array[]){
        quickSort(array, 0, array.length-1);
    }
```

3. Testes

3.1. Parâmetros de teste

Para os testes foram usados:

- Números randômicos de 0 a 100
- 4 tamanhos de entrada:
  - 10^2
  - 10^3
  - 10^4
  - 10^5
- 30 testes em cada tamanho
- Métricas avaliadas:
  - Media do tempo em milissegundos(ms)
  - Número de comparações
  - Número de movimentações

Função Geradora

```
    private static int[] generateRandomArray(int n) {
        Random random = new Random();
        int[] array = new int[n];
        for (int i = 0; i < n; i++) {
            array[i] = random.nextInt(100);
        }
        return array;
    }
```

3.2. Resultados

Algoritmo	Comparações Média	Movimentações Média	Tempo Médio(ms)	Tamanho
SelectionSort	4950	297	0	100
InsertionSort	2651	2754	0	100
BubbleSort	4949	7668	0	100
QuickSort	740	531	0	100
SelectionSort	499500	2997	0	1000
InsertionSort	245366	246371	0	1000
BubbleSort	499122	733119	0	1000
QuickSort	10771	7965	0	1000
SelectionSort	49995000	29997	138	10000
InsertionSort	24948742	24958747	9	10000
BubbleSort	49965597	74816247	48	10000
QuickSort	139320	108261	0	10000
SelectionSort	4999950000	299997	13439	100000
InsertionSort	2478286854	2478386858	917	100000
BubbleSort	4999048847	7434560580	11754	100000
QuickSort	1715960	1592460	4	100000

Figure 1. Resultados

A primeira observação que podemos perceber que em questão do tempo o **QuickSort** por ser de complexidade $O(n \log n)$ só começou a exibir um tempo de execução maior que 0ms a partir do tamanho 10^5

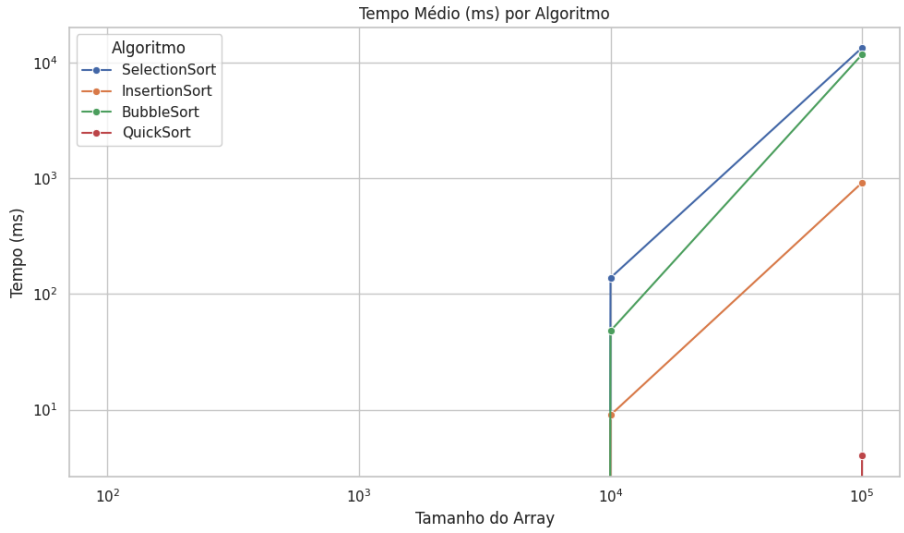

Figure 2. Tempo Geral

Sobre a visão de quantidade de comparações o **QuickSort** apresenta também a menor quantidade

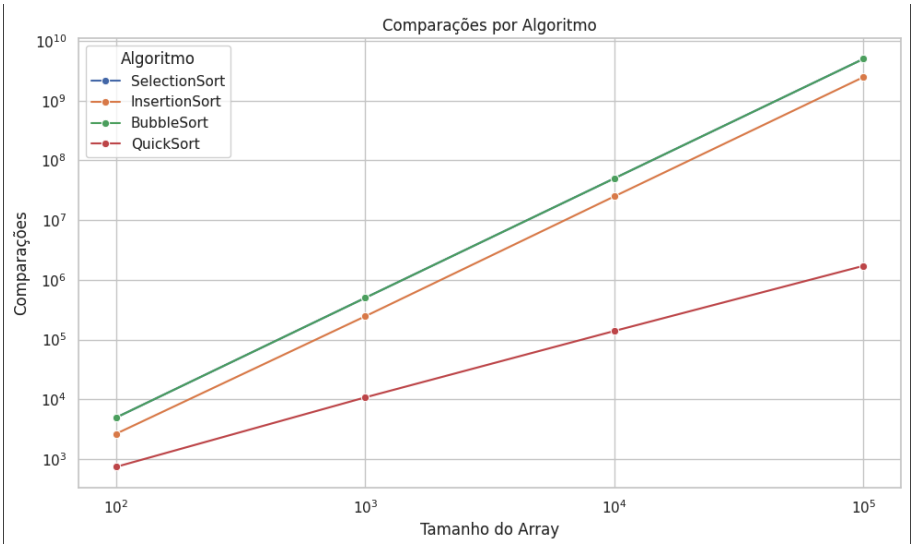

Figure 3. Comparações Gerais

Em quantidade de movimentações quem se destaca acaba sendo o **SelectionSort** e o **QuickSort**, pois tem respectivamente $O(n)$ e $O(n \log n)$

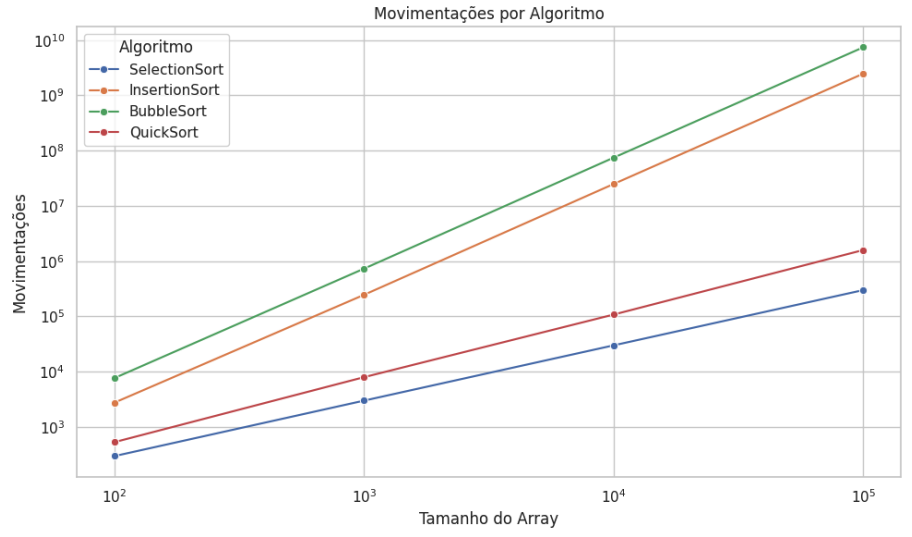

Figure 4. Movimentações Gerais

3.3. Conclusão

Podemos concluir, ao comparar a complexidade desses quatro algoritmos de ordenação, que quanto menor a complexidade assintótica, mais rápido e com menos recursos o algoritmo tende a executar. Neste estudo, por exemplo, o **QuickSort** se destaca — considerando apenas comparações, movimentações e tempo — por apresentar o melhor desempenho na maioria dos casos. No entanto, sob a perspectiva do uso de memória, os demais algoritmos possuem complexidade espacial de $O(1)$, enquanto o **QuickSort** pode atingir $O(n)$ no pior caso e $O(\log n)$ no melhor.

Assim, é fundamental escolher o algoritmo mais adequado ao seu caso de uso, a fim de obter os melhores resultados.

4. References

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1998.