# Algoritmos de Organização

**Camily Pereira Albres**[1]

[1]Pontíficia Universidade Católica do Paraná (PUCPR)
Escola Politécnica – Curitiba – PR – Brazil

https://github.com/Kamirii/Sort-Algorithms

***Abstract.*** *This article presents performance experiments comparing the Heap Sort, Insertion Sort, and Merge Sort algorithms across five vector sizes (50, 500, 1000, 5000, and 10000). Averages for runtime, number of swaps, and iterations were analyzed. These metrics provide a comprehensive view of algorithm performance under various conditions, offering an opportunity to understand each implemented sorting algorithm.*

***Resumo.*** *Este artigo apresenta experimentos de desempenho entre os algoritmos Heap Sort, Insertion Sort e Merge Sort, usando cinco tamanhos de vetores (50, 500, 1000, 5000 e 10000).Foram analisados as médias de: tempo de execução, número de trocas e iterações. Essas métricas oferecem uma visão completa do desempenho dos algoritmos em diversas condições, proporcionando assim a oportunidade de entender cada algoritmo de organização implementado.*

## 1. Insertion Sort

Insertion sort é um dos algoritmos de organização mais fáceis para organizar uma lista ou array de valores. O algoritmo em consiste em percorrer a lista, item por item, e posicionar cada elemento na ordem correta em relação aos elementos já ordenados.

Para cada $A[i]$, se $A[i] > A[i+1]$, troque o elemento até $A[i] \leq A[i+1]$

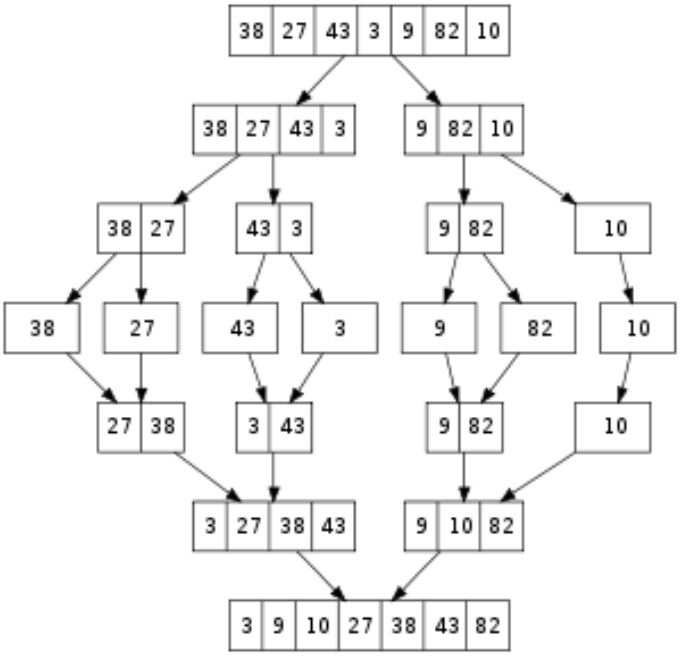
### 1.1. Implementação em Java

```
for (int i = 1; i < tam; i++) {
    int x = vetor[i];
    int j = i - 1;
    while (j >= 0 && vetor[j] > x) {
        vetor[j + 1] = vetor[j];
        j = j - 1;
    }
    vetor[j + 1] = x;
}
```

## 2. Merge Sort

O Merge sort é um algoritmo bastante interessante porque ele utiliza a abordagem *divide and conquer*. Essa abordagem consiste em quebrar o problema inicial em problemas menores,até que a solução do problema inicial seja dada pela combinação dos resultados dos problemas menores. O Merge Sort funciona exatamente assim. Nós recebemos

uma lista de *n* valores, que será dividida em duas partes iguais, ou proximas de iguais. A partir disso, podemos continuar dividindo essas sub-listas em listas menores. O objetivo é ordenar essas listas menores e realizar a junção de todas as sub-listas ordenadas para ordenar a lista do tamanho original.




### 2.1. Implementação em Java

O Merge Sort pode ser implementado de maneira interativa e também de maneira recursiva.

```
private void mergeSort(int min, int max) {
    if (min < max) {
        int mid = (min + max) / 2;
        mergeSort(min, mid);
        mergeSort(mid + 1, max);
        merge(min, mid, max);
    }
 }


private void merge(int min, int mid, int max) {
    int[] temp = new int[this.tam];

    for (int i = min; i <= max; i++) {
        temp[i] = vetor[i];
    }

    int i = min;
    int j = mid + 1;
```

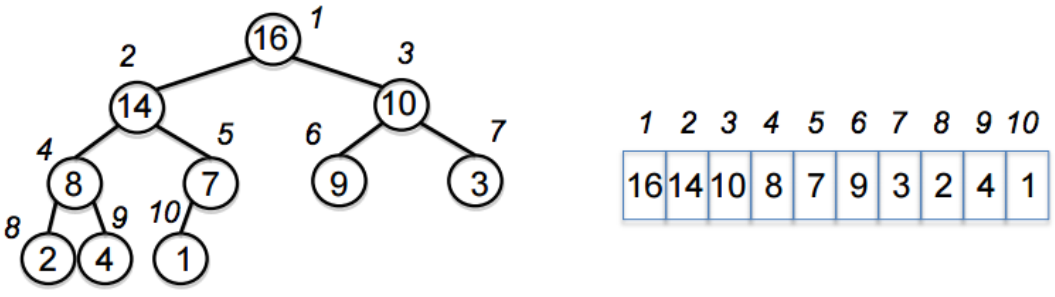
```
int k = min;

while (i <= mid && j <= max) {
    if (temp[i] <= temp[j]) {
    vetor[k] = temp[i];
    i++;
} else {
    vetor[k] = temp[j];
    j++;
}
k++;
}

while (i <= mid) {
    this.vetor[k] = temp[i];
    k++;
    i++;
    }
}
```

## 3. Heap Sort

O Heapsort é um algoritmo de ordenação baseado em comparações que utiliza uma estrutura de dados chamada heap binário. O heap binário é uma árvore binária mantida na forma de um vetor. Assim como o mergesort, o heapsort possui uma complexidade de tempo de O(n log n), e, assim como o insertion sort, o heapsort realiza a ordenação no local (in-place), o que significa que nenhum espaço adicional é necessário durante o processo de ordenação.




### 3.1. Implementação em Java

```
public int[] sort() {

    int mid = tam / 2 - 1;
```

```
        for (int i = mid; i >= 0; i--) {
            constroiHeap(tam, i);
        }

        for (int i = tam - 1; i > 0; i--) {
            int temp = vetor[0];
            vetor[0] = vetor[i];
            vetor[i] = temp;
            constroiHeap(i, 0);
        }

        return vetor;
    }


    private void constroiHeap(int tamHeap, int i) {

        int paiMaior = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < tamHeap && vetor[left] > vetor[paiMaior]) {
            paiMaior = left;
        }

        if (right < tamHeap && vetor[right] > vetor[paiMaior]) {
            System.out.println(right);
            paiMaior = right;
        }

        if (paiMaior != i) {
            int swap = vetor[i];
            vetor[i] = vetor[paiMaior];
            vetor[paiMaior] = swap;
            numTrocas++;
            numIteracoes++;

            constroiHeap(tamHeap, paiMaior);
        }
    }
```

## 4. Resultados

Foram utilizado vetores inteiros de números aleatórios de 5 tamanhos diferentes: 50,500,1000,5000 e 1000.

| INSERTION SORT | | | | | |
|---|---|---|---|---|---|
| Resultados | 50 | 500 | 1000 | 5000 | 10000 |
| Tempo de execução médio | 0 ms | 3 ms | 2 ms | 14 ms | 31 ms |
| Número Médio de Trocas | 684 | 62495 | 237385 | 6232204 | 24851999 |
| Número Médio de Iterações | 49 | 499 | 999 | 4999 | 9999 |

| MERGE SORT | | | | | |
|---|---|---|---|---|---|
| Resultados | 50 | 500 | 1000 | 5000 | 10000 |
| Tempo de execução médio | 0 ms | 1 ms | 3 ms | 53 ms | 174 ms |
| Número Médio de Trocas | 256 | 4171 | 9363 | 59080 | 128207 |
| Número Médio de Iterações | 256 | 4171 | 9363 | 59080 | 128207 |

| HEAP SORT | | | | | |
|---|---|---|---|---|---|
| Resultados | 50 | 500 | 1000 | 5000 | 10000 |
| Tempo de execução médio | 0 ms | 0 ms | 0 ms | 1 ms | 3 ms |
| Número Médio de Trocas | 238 | 4041 | 9087 | 57157 | 124262 |
| Número Médio de Iterações | 238 | 4041 | 9087 | 57157 | 124262 |

## 5. Discussão

Ao realizar os testes, podemos perceber que o Insertion Sort, foi perdendo desempenho comforme o tamanho de vetores aumentavam. Além disso, o Merge Sort também perdeu desempenho, isso porque esse algoritmo cria uma estrutura temporária para poder organizar os dados. O heap Sort manteve seu desempenho com todos os tamanhos de vetores, ele é in-place e possui uma complexidade de tempo de execução interessante.