UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO” (UNESP)
ANÁLISE DE ALGORITMOS

GUSTAVO YUJII SILVA KADOOKA

# ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO APLICADOS A DIFERENTES CONJUNTOS DE DADOS

BAURU/SP
2025

UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO” (UNESP)
GUSTAVO YUJII SILVA KADOOKA

# ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO APLICADOS A DIFERENTES CONJUNTOS DE DADOS

Relatório apresentado como requisito à obtenção de uma das notas da disciplina de Análise de Algoritmos do curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”.

Prof. Dr. João Paulo Papa

BAURU/SP
2025

# Lista de Figuras

# Sumário

**Páginas**

# 1 Introdução

Os algoritmos de ordenação representam uma das áreas mais fundamentais da ciência da computação. Eles estão presentes em uma vasta gama de aplicações que vão desde a organização de dados em bancos de dados até otimizações em sistemas de busca e compressão de dados. Compreender o funcionamento, as complexidades e os comportamentos dos principais algoritmos de ordenação é essencial para que se possa escolher a abordagem mais eficiente para cada situação. Este relatório apresenta uma análise comparativa entre seis algoritmos de ordenação — Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort e Radix Sort — aplicados a diferentes tipos de conjuntos de dados.

# 2 Objetivos

## 2.1 Objetivo geral

Analisar e comparar o desempenho de diferentes algoritmos de ordenação em conjuntos de dados com diferentes características, por meio de simulações computacionais.

## 2.2 Objetivos específicos

- Implementar seis algoritmos clássicos de ordenação;
- Realizar simulações utilizando três tipos distintos de conjuntos de dados: ordenados de forma crescente, ordenados de forma decrescente e aleatórios;
- Avaliar o desempenho dos algoritmos considerando diferentes tamanhos de entrada;
- Medir e registrar o tempo de execução de cada algoritmo em cada cenário;
- Discutir os resultados obtidos com base na complexidade teórica e no comportamento observado.

# 3 Justificativas

A escolha por analisar algoritmos de ordenação justifica-se pela relevância desses métodos em diversas aplicações práticas e teóricas da computação. Conhecer o comportamento de diferentes algoritmos frente a diferentes tipos de entrada é essencial para a escolha adequada em contextos reais, nos quais o desempenho e a eficiência são fatores críticos. A comparação experimental permite observar na prática os impactos das complexidades algorítmicas e fornece uma base concreta para o entendimento dos limites e vantagens de cada abordagem.

# 4 Metodologia

Os experimentos foram realizados por meio da implementação dos algoritmos Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort e Radix Sort em linguagem C. Para cada algoritmo, foram gerados vetores de diferentes tamanhos (1.000, 7.600, 14.200, 20.800, 27.400, 34.000, 40.600, 47.200, 53.800, 60.400, 67.000, 73.600, 80.200, 86.800, 93.400 e 100.000 elementos), organizados em três tipos de ordenação:

- Crescente: vetor previamente ordenado em ordem crescente;
- Decrescente: vetor ordenado em ordem decrescente;
- Aleatória: vetor com elementos gerados de forma randômica.

Cada experimento foi executado cinco vezes, sendo registrado o tempo de execução médio, em milissegundos, para garantir maior confiabilidade nos resultados. Os tempos de execução foram obtidos por meio da biblioteca time.h da linguagem de programação C. Os dados coletados foram organizados em uma tabela csv e representados graficamente para facilitar a análise comparativa.

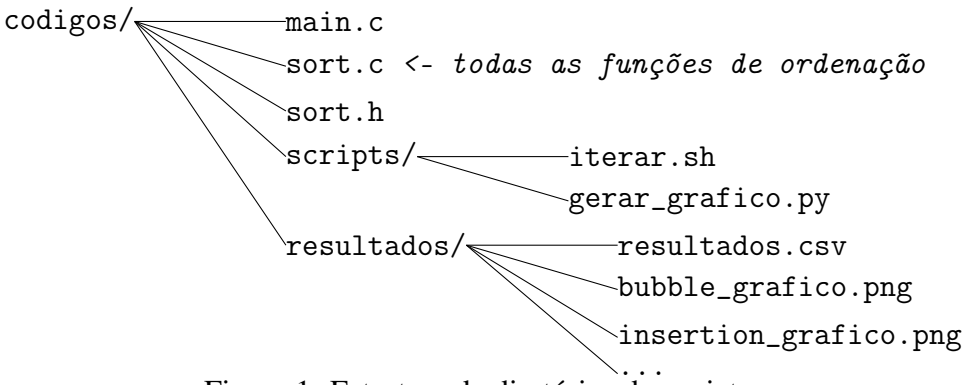

Figura 1: Estrutura de diretórios do projeto.

A estrutura do projeto foi organizada de forma modular, conforme ilustrado na Figura 1, com o objetivo de facilitar a manutenção, testes e análise de desempenho dos algoritmos de ordenação. A seguir, são descritas as funcionalidades dos principais componentes do projeto:

- `main.c`: Arquivo principal que interpreta os argumentos passados via linha de comando, gera os vetores de entrada conforme o tipo solicitado (crescente, decrescente ou aleatório) e executa o algoritmo de ordenação correspondente.
- `sort.c` e `sort.h`: Contêm, respectivamente, a implementação e os protótipos das funções dos algoritmos de ordenação utilizados no projeto: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort e Radix Sort. O algoritmo a ser utilizado é selecionado de acordo com o parâmetro passado na execução.
- `scripts/`: Diretório com scripts auxiliares para automação dos testes e geração de gráficos:

- `iterar.sh`: Script em Bash responsável por automatizar a execução de testes para diferentes tamanhos de vetores, tipos de ordenação e algoritmos. Os resultados são armazenados em um arquivo `.csv`.
- `gerar_grafico.py`: Script em Python que utiliza as bibliotecas `seaborn, numpy, pandas e matplotlib` para gerar gráficos de desempenho com base nos resultados obtidos.

- `resultados/`: Diretório onde são armazenados os resultados dos testes e os gráficos gerados:
  - `resultados.csv`: Arquivo contendo os dados brutos dos testes, incluindo o algoritmo utilizado, o tamanho do vetor, o tipo de ordenação e o tempo de execução.
  - Arquivos `.png`: Gráficos de desempenho para cada algoritmo, gerados automaticamente a partir do script Python.

Essa organização modular permitiu realizar testes sistemáticos e automatizados, além de facilitar a coleta e visualização dos dados, contribuindo para uma análise comparativa eficiente do desempenho dos algoritmos de ordenação.

## 4.1 Principais Trechos de Código

Nesta seção, são apresentados os principais trechos de código que compõem a base da execução dos experimentos, com foco na automação e análise dos resultados.

### 4.1.1 Função Principal (`main.c`)

A função `main` é responsável por interpretar os parâmetros de entrada, gerar o vetor com base no tipo escolhido e executar o algoritmo de ordenação correspondente.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "sort.h"

void preencher_vetor(int *v, int n, int tipo) {
    if (tipo == 0)
        for (int i = 0; i < n; i++) v[i] = i;
    else if (tipo == 1)
        for (int i = 0; i < n; i++) v[i] = n - i;
    else {
        srand(42);
        for (int i = 0; i < n; i++) v[i] = rand()%100000;
    }
```

```c
}

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Uso: %s <algoritmo> <tamanho> <tipo>\n", argv[0]);
        printf("Algoritmos: bubble, insertion, merge, quick, heap, radix
            \n");
        return 1;
    }

    char *alg = argv[1];
    int n = atoi(argv[2]);
    int tipo = atoi(argv[3]);

    int *v = malloc(n * sizeof(int));
    preencher_vetor(v, n, tipo);

    clock_t ini = clock();

    if (strcmp(alg, "bubble") == 0)
        bubble_sort(v, n);
    else if (strcmp(alg, "insertion") == 0)
        insertion_sort(v, n);
    else if (strcmp(alg, "merge") == 0)
        merge_sort(v, 0, n-1);
    else if (strcmp(alg, "quick") == 0)
        quick_sort(v, 0, n-1);
    else if (strcmp(alg, "heap") == 0)
        heap_sort(v, n);
    else if (strcmp(alg, "radix") == 0)
        radix_sort(v, n);
    else {
        fprintf(stderr, "Algoritmo inválido: %s\n", alg);
        free(v);
        return 1;
    }

    clock_t fim = clock();
    double tempo = (double)(fim - ini) / CLOCKS_PER_SEC;

    printf("%s,%d,%d,%.6f\n", alg, n, tipo, tempo);

    free(v);
    return 0;
}
```

Listing 1: Função principal com medição de tempo

### 4.1.2 Script de Execução Automática (`iterar.sh`)

Este script em Bash automatiza a execução dos testes, iterando sobre todos os algoritmos, tamanhos de vetor e tipos de ordenação. Os resultados são salvos em formato CSV para posterior análise.

```
#!/bin/bash

ALGOS=("bubble" "insertion" "merge" "quick" "heap" "radix")
TAMANHOS=(1000 7600 14200 20800 27400 34000 40600 47200 53800 60400
    67000 73600 80200 86800 93400 100000)
TIPOS=(0 1 2) # 0=crescente, 1=decrescente, 2=aleatório

mkdir -p ../resultados
echo "algoritmo,tamanho,tipo,tempo" > ../resultados/resultados.csv

gcc -O2 -o ../sorter ../main.c ../sort.c

for algo in "${ALGOS[@]}"; do
    echo "Executando testes para $algo..."
    for tipo in "${TIPOS[@]}"; do
        for n in "${TAMANHOS[@]}"; do
            soma=0
            for i in {1..5}; do
                resultado=$(../sorter $algo $n $tipo)
                tempo=$(echo "$resultado" | awk -F',' '{print $4}')
                soma=$(echo "$soma + $tempo" | bc -l)
            done
            media=$(echo "$soma / 5" | bc -l)
            echo "$algo,$n,$tipo,$media" >> ../resultados/resultados.csv
        done
    done
done
```

Listing 2: Script para automação dos testes

### 4.1.3 Script de Geração de Gráficos (`gerar_grafico.py`)

O script em Python utiliza a biblioteca `pandas` para ler os resultados e `matplotlib/seaborn` para criar gráficos que comparam o desempenho dos algoritmos em diferentes cenários.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Carregar os dados
df = pd.read_csv("../resultados/resultados.csv")
```

```
8
9  # Mapear tipos para nomes legíveis
10 mapa_tipos = {0: "Crescente", 1: "Decrescente", 2: "Aleatório"}
11 df["tipo"] = df["tipo"].map(mapa_tipos)
12
13 # Estilo bonito com seaborn
14 sns.set(style="whitegrid")
15
16 # Lista de algoritmos únicos
17 algoritmos = df["algoritmo"].unique()
18 print(algoritmos)
19
20 # Gerar um gráfico para cada algoritmo
21 for alg in algoritmos:
22     dados_alg = df[df["algoritmo"] == alg]
23
24     plt.figure(figsize=(8,5))
25
26     # Plotar curvas para cada tipo de vetor
27     for tipo in dados_alg["tipo"].unique():
28         dados_tipo = dados_alg[dados_alg["tipo"] == tipo]
29         plt.plot(dados_tipo["tamanho"], dados_tipo["tempo"], marker="o",
               label=tipo)
30
31     plt.title(f"Desempenho do {alg.capitalize()} Sort")
32     plt.xlabel("Tamanho do Vetor")
33     plt.ylabel("Tempo (s)")
34     plt.legend()
35     plt.tight_layout()
36     plt.savefig(f"../resultados/{alg}_grafico.png")
37     plt.close()  # Fechar para não sobrepor gráficos
```

Listing 3: Geração de gráficos com base nos dados

Esses trechos representam o núcleo da automação do projeto, permitindo a execução sistemática dos algoritmos e a análise gráfica dos seus desempenhos.

# 5 Resultados e discussões

Nesta seção são apresentados os resultados obtidos para cada algoritmo de ordenação implementado. Para cada um, é discutida sua complexidade teórica, apresentado o código-fonte correspondente e analisado o desempenho por meio de gráficos gerados com os tempos de execução coletados nas simulações.

## 5.1 Funções e macros utilizadas

```
#define ARRAY_SIZE(v) sizeof(v)/sizeof(v[0])
static inline void swap(int *a, int *b) { if(a!=b) { (*a^=*b), (*b^=*a),
    (*a^=*b); } }
```

Listing 4: Funções e macros

## 5.2 Bubble Sort

```
void bubble_sort(int *v, int length)
{
    register int i, j;
    for(i=0; i<length-1; i++)
    {
        int is_sorted = 1;
        for(j=0; j<length-i-1; j++)
        {
            if(v[j+1] < v[j])
            {
                swap(&v[j+1], &v[j]);
                is_sorted = 0;
            }
        }
        if(is_sorted)
            break;
    }
}
```

Listing 5: Implementação do Bubble Sort em C com otimização

### 5.2.1 Complexidade Teórica

O algoritmo Bubble Sort realiza múltiplas passagens pelo vetor, comparando pares de elementos adjacentes e trocando-os de posição se estiverem fora de ordem. O laço mais externo é executado até que o vetor esteja ordenado ou até `n - 1` iterações. Dentro dele, o laço interno percorre `n - i - 1` elementos, onde `i` é o índice atual da iteração externa.

No pior caso, o vetor está ordenado em ordem decrescente. Nesse cenário, cada elemento precisa ser trocado com todos os seguintes até atingir sua posição correta, totalizando aproximadamente:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

No melhor caso, quando o vetor já está ordenado, a flag `is_sorted` evita iterações desnecessárias, encerrando o laço externo após a primeira passada. Isso resulta em uma complexidade de tempo $O(n)$, considerando apenas um percurso linear sem trocas.

O caso médio também resulta em $O(n^2)$, já que na média são necessárias várias passagens e comparações antes da ordenação completa.

Em termos de espaço, o Bubble Sort é um algoritmo in-place, com complexidade de espaço $O(1)$, pois não utiliza estruturas auxiliares significativas. (**??**)

### 5.2.2 Resultados Experimentais

A Figura 2 apresenta o desempenho empírico do algoritmo *Bubble Sort* para diferentes tamanhos de vetores e tipos de ordenação inicial. Como esperado, o tempo de execução cresce quadraticamente com o tamanho do vetor, especialmente no caso de vetores em ordem decrescente e aleatório.

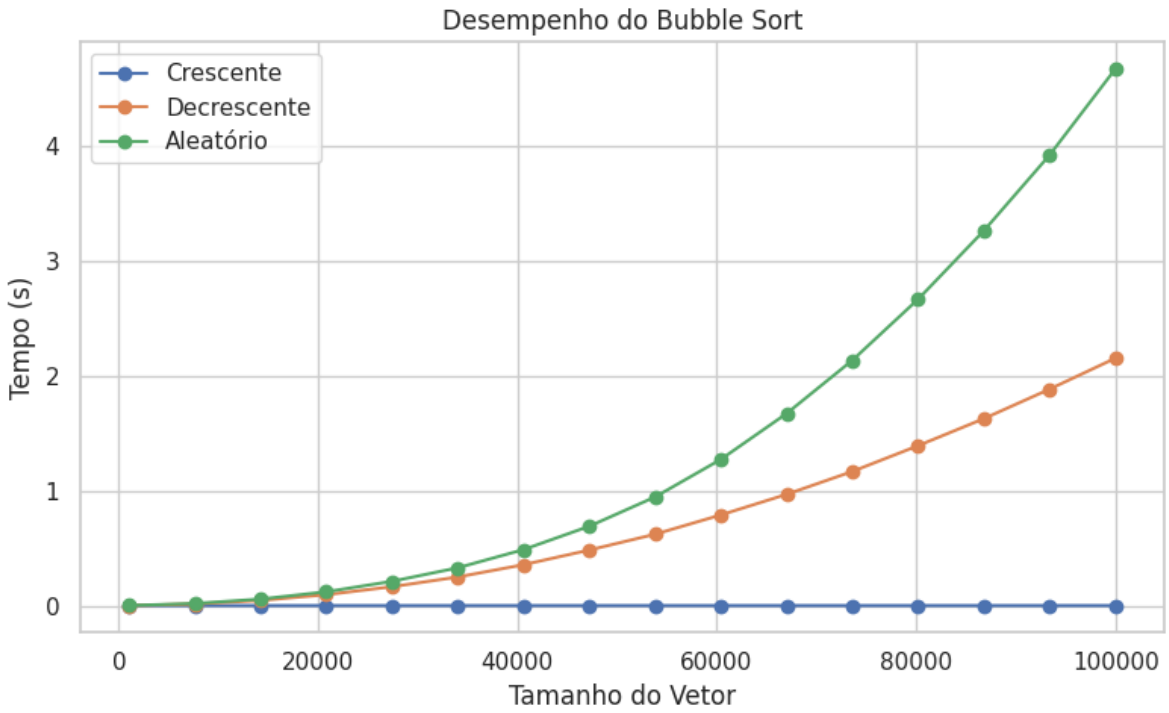


Figura 2: Tempo de execução do Bubble Sort para diferentes entradas

## 5.3 Insertion Sort

```
1 void insertion_sort(int *v, int len)
2 {
3     register int i, j;
4     for(i=1; i<len; i++)
5     {
6         int key = v[i];
```

```
7           for(j=i-1; j>=0 && v[j]>key; j--)
8               v[j+1] = v[j];
9           v[j+1] = key;
10      }
11  }
```

Listing 6: Implementação do Insertion Sort

### 5.3.1 Complexidade Teórica

O algoritmo *Insertion Sort* percorre o vetor da esquerda para a direita, inserindo cada elemento na posição correta em relação aos anteriores. Ele é eficiente para vetores pequenos ou quase ordenados.

No pior caso, o vetor está ordenado em ordem decrescente. A cada nova iteração, o elemento atual precisa ser comparado com todos os anteriores, deslocando-os para a direita. Isso leva a:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

No melhor caso, quando o vetor já está em ordem crescente, nenhuma troca é feita — apenas uma comparação por iteração. Isso resulta em uma complexidade de tempo linear:

$$O(n)$$

No caso médio, assumindo entradas aleatórias, espera-se que cada elemento percorra metade dos anteriores, o que leva novamente a:

$$O(n^2)$$

A complexidade de espaço é $O(1)$, pois o algoritmo é in-place e não utiliza estruturas auxiliares adicionais. (**??**)

### 5.3.2 Resultados Experimentais

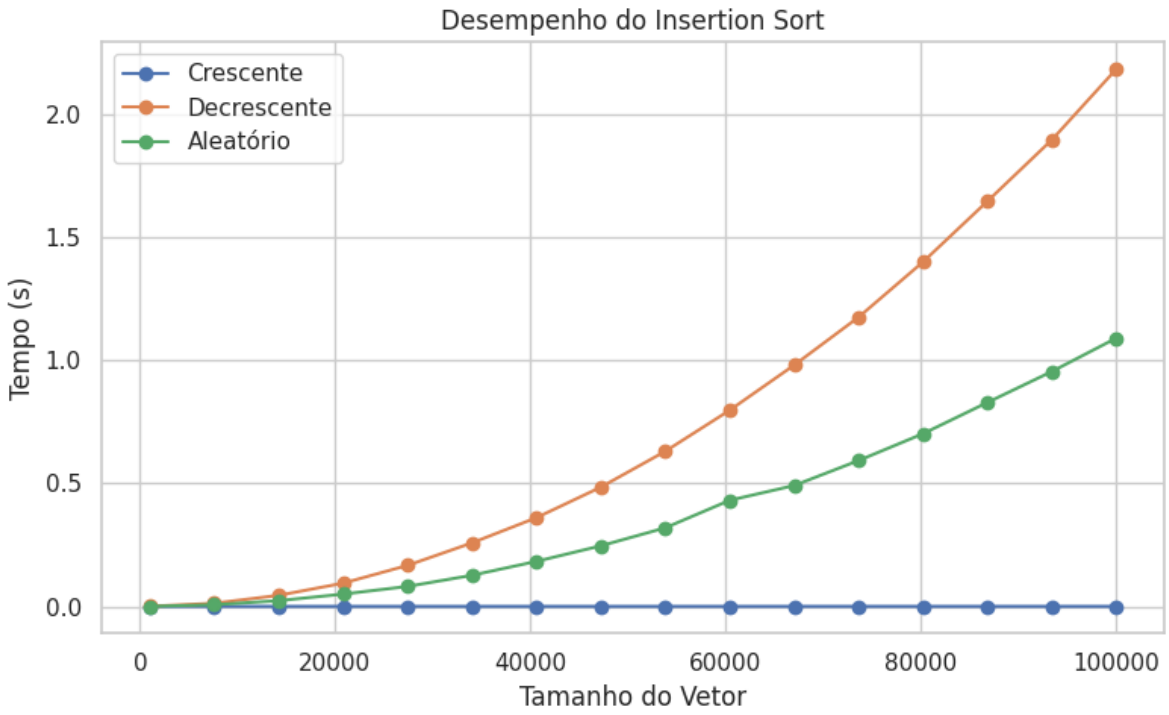


Figura 3: Tempo de execução do Insertion Sort para diferentes entradas

## 5.4 Quick Sort

```
int lomuto(int *v, int low, int high)
{
    int *pivot = &v[high];
    int boundary = low;
    for(int i=low; i<high; i++) {
        if(v[i] <= *pivot)
            swap(&v[i], &v[boundary++]);
    }
    swap(pivot, &v[boundary]);

    return boundary;
}

int hoare(int *v, int low, int high)
{
    int i=low-1, j=high+1;
    int pivot = v[low];
    while(1) {
        do {
            i++;
        } while(v[i] < pivot);
```

```
22          do {
23              j--;
24          } while(v[j] > pivot);
25          if(i>=j)
26              return j;
27          swap(&v[i], &v[j]);
28      }
29  }
30
31  void quick_sort(int *v, int low, int high)
32  {
33      if(low < high) {
34          int boundary = hoare(v, low, high);
35          quick_sort(v, low, boundary);
36          quick_sort(v, boundary+1, high);
37      }
38  }
```

Listing 7: Implementação do Quick Sort

### 5.4.1 Complexidade Teórica

O *Quick Sort* é um algoritmo baseado na estratégia de divisão e conquista. Ele escolhe um elemento pivô e reorganiza o vetor de forma que todos os elementos menores que o pivô fiquem à sua esquerda e os maiores à direita. Esse processo é repetido recursivamente em cada subvetor.

Existem duas estratégias principais de particionamento:

- **Lomuto:** escolhe normalmente o último elemento como pivô. O particionamento é mais simples, mas menos eficiente em alguns casos devido ao maior número de trocas.
- **Hoare:** utiliza dois ponteiros que percorrem o vetor a partir das extremidades, trocando elementos fora de lugar. Costuma realizar menos trocas e é mais eficiente na prática.

A complexidade do Quick Sort depende da escolha do pivô:

- **Melhor caso:** o pivô divide o vetor em duas partes iguais a cada recursão. Isso resulta em:

$$O(n \log n)$$

- **Pior caso:** o pivô é sempre o menor ou o maior elemento, criando partições de tamanho $n-1$ e 0. Isso leva a:

$$O(n^2)$$

- **Caso médio:** em geral, o particionamento divide o vetor razoavelmente bem, levando a:

$$O(n \log n)$$

O Quick Sort é um algoritmo in-place (com complexidade de espaço $O(\log n)$ devido à recursão), e sua versão padrão não é estável. Porém, por sua eficiência prática e uso frequente da memória cache, é amplamente utilizado. (**??**)

### 5.4.2 Resultados Experimentais

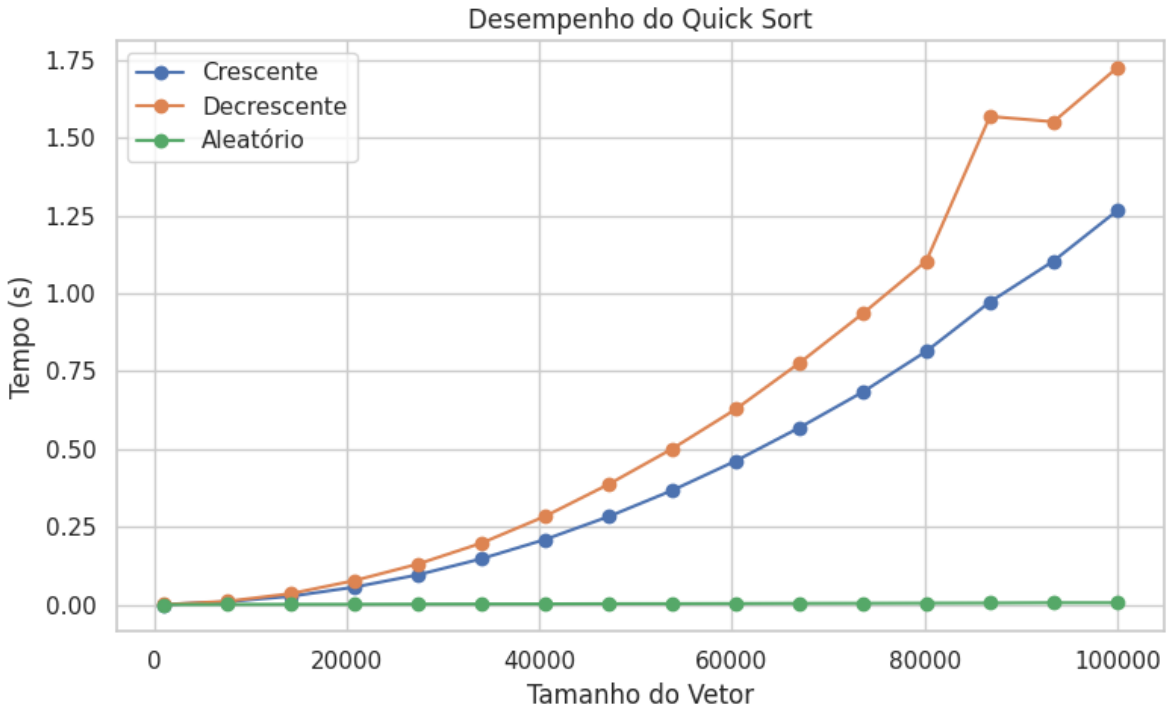


Figura 4: Tempo de execução do Quick Sort com método de Hoare para diferentes entradas

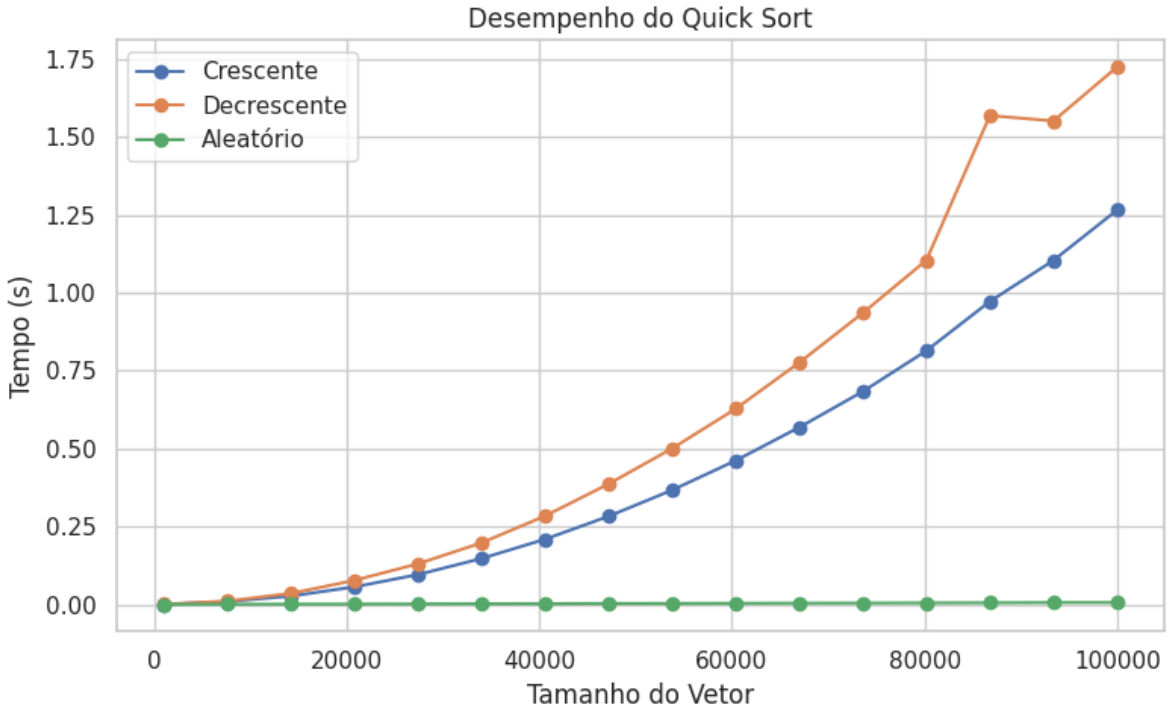


Figura 5: Tempo de execução do Quick Sort com método de Lomuto para diferentes entradas

## 5.5 Merge Sort

```
void merge(int v[], int low, int mid, int high)
{
    int n1 = mid-low+1;
    int n2 = high-mid;
    int v1[n1], v2[n2];

    for(int i=0; i<n1; i++)
        v1[i] = v[low+i];
    for(int i=0; i<n2; i++)
        v2[i] = v[mid+1+i];

    int i=0,j=0,k=low;
    while(i<n1 && j<n2)
    {
        if(v1[i] < v2[j])
            v[k++] = v1[i++];
        else
            v[k++] = v2[j++];
    }
    while(i<n1)
        v[k++] = v1[i++];
    while(j<n2)
        v[k++] = v2[j++];
```

```
24 }
25
26 void merge_sort(int *v, int low, int high)
27 {
28     if(low < high) {
29         int mid = (low+high)/2;
30         merge_sort(v, low, mid);
31         merge_sort(v, mid+1, high);
32
33         merge(v, low, mid, high);
34     }
35 }
```

Listing 8: Implementação do Merge Sort

### 5.5.1 Complexidade Teórica

O algoritmo *Merge Sort* segue a estratégia de *divisão e conquista*. Ele divide o vetor recursivamente ao meio até que cada subvetor tenha tamanho 1 e, em seguida, realiza a fusão (*merge*) ordenada desses subvetores.

Em cada nível da recursão, o vetor é dividido pela metade, resultando em $\log_2 n$ níveis. A cada nível, ocorre a fusão de todos os elementos, com custo proporcional a $n$. Portanto, a complexidade no pior, melhor e caso médio é:

$$O(n \log n)$$

O desempenho do Merge Sort é estável, independentemente da ordem inicial dos dados, e ele sempre realiza o mesmo número de comparações e fusões.

Entretanto, como a fusão exige a criação de vetores auxiliares para armazenar temporariamente os dados, o Merge Sort não é in-place. Isso leva a uma complexidade de espaço adicional de:

$$O(n)$$

Apesar disso, sua eficiência e previsibilidade o tornam uma boa escolha para dados grandes e em aplicações que exigem estabilidade na ordenação. (**??**)

### 5.5.2 Resultados Experimentais

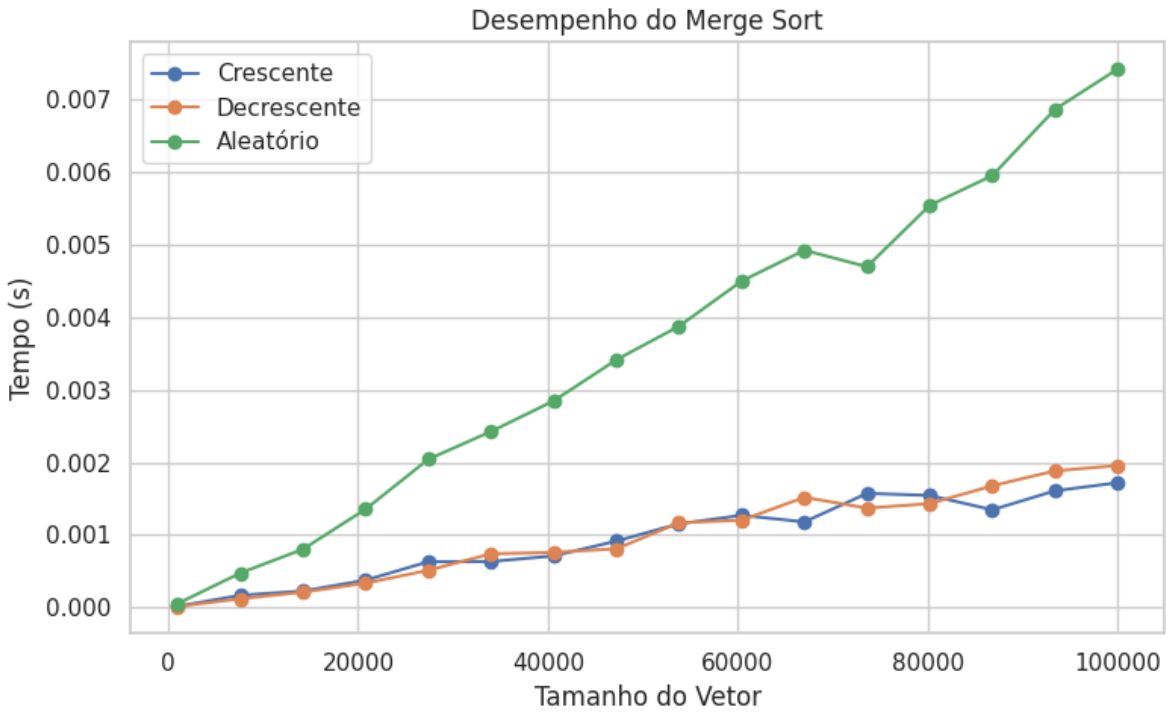


Figura 6: Tempo de execução do Merge Sort para diferentes entradas

## 5.6 Heap Sort

```
1  void max_heapify(int v[], int n, int i)
2  {
3      int l = 2*i+1;
4      int r = 2*i+2;
5      int larg = i;
6      if(l < n && v[l] > v[larg])
7          larg = l;
8      if(r < n && v[r] > v[larg])
9          larg = r;
10     if(larg != i) {
11         swap(&v[larg], &v[i]);
12         max_heapify(v, n, larg);
13     }
14 }
15
16 void heap_sort(int *v, int n)
17 {
18     for(int i=n/2-1; i>=0; i--)
19         max_heapify(v, n, i);
20
21     n--;
```

```
22      while(n) {
23          swap(&v[0], &v[n--]);
24          max_heapify(v, n, 0);
25      }
26  }
```

Listing 9: Implementação do Heap Sort

### 5.6.1 Complexidade Teórica

O *Heap Sort* é um algoritmo de ordenação baseado em uma estrutura de dados chamada *heap*, que é uma árvore binária completa com a propriedade de heap (no caso do max-heap, cada nó é maior que seus filhos).

O algoritmo funciona em duas fases:

1. Construção do heap a partir do vetor desordenado (fase de construção).
2. Repetidamente extrair o maior elemento (raiz do heap) e reestruturar o heap restante (fase de ordenação).

A análise da complexidade é dividida da seguinte forma:

- **Construção do heap:** cada elemento é movido para manter a propriedade do heap, o que leva tempo proporcional à sua altura. No total, essa fase tem complexidade:

$$O(n)$$

- **Remoção e reestruturação:** cada uma das $n$ remoções exige uma operação de *heapify*, com custo $O(\log n)$, resultando em:

$$O(n \log n)$$

- **Complexidade total:**

$$O(n \log n)$$

O Heap Sort é um algoritmo in-place, pois utiliza apenas uma quantidade constante de memória adicional ($O(1)$). No entanto, ele não é estável, e, embora sua complexidade seja semelhante à do Quick Sort no caso médio, tende a ser mais lento em prática por causa do maior custo constante das operações de heap. (**??**)

### 5.6.2 Resultados Experimentais

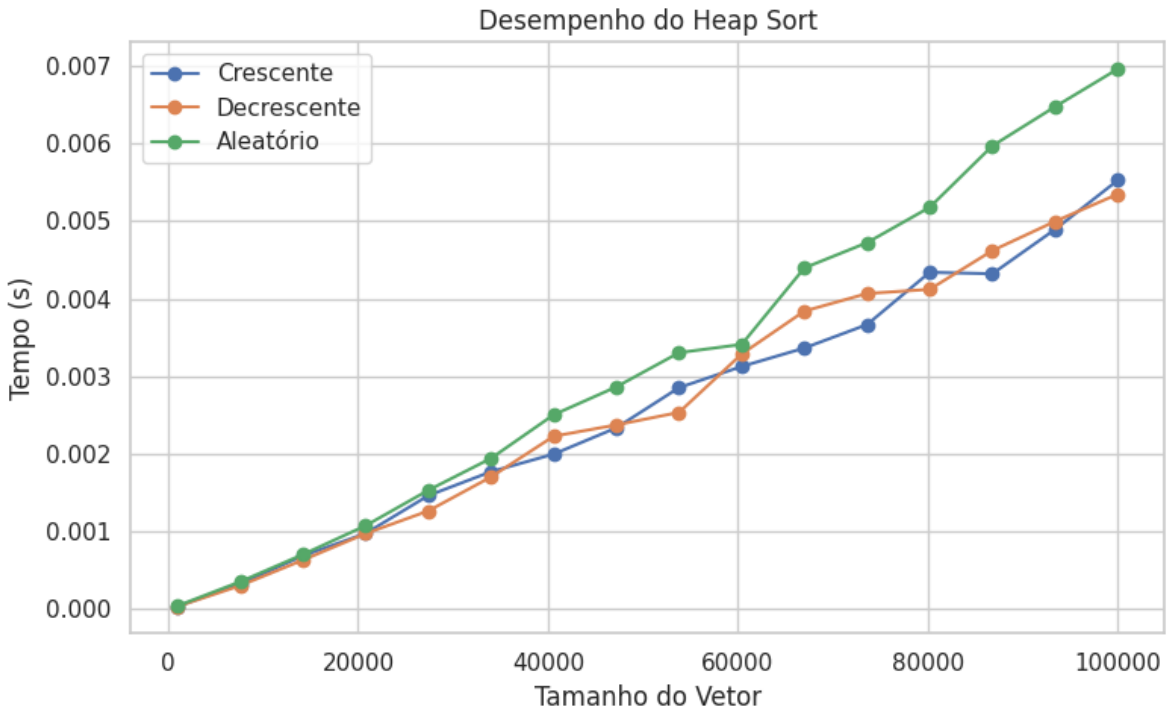


Figura 7: Tempo de execução do Heap Sort para diferentes entradas

## 5.7 Radix Sort

```
int getmax(int v[], int len)
{
    int max = v[0];
    for(int i=1; i<len; i++)
        if(v[i]>max) max = v[i];
    return max;
}

void counting_sort(int v[], int len, int exp)
{
    int *answer = malloc(len * sizeof(int));
    int count[10] = {0};

    int i;
    for(i=0; i<len; i++)
        count[(v[i]/exp)%10]++;
    for(i=1; i<10; i++)
        count[i] += count[i-1];
    for(i=len-1; i>=0; i--) {
        answer[count[(v[i]/exp)%10]-1] = v[i];
        count[(v[i]/exp)%10]--;
```

```
22      }
23      for(i=0; i<len; i++)
24          v[i] = answer[i];
25
26      free(answer);
27  }
28
29  void radix_sort(int *v, int len)
30  {
31      int max = getmax(v, len);
32
33      for(int i=1; max/i>0; i*=10)
34          counting_sort(v, len, i);
35  }
```

Listing 10: Implementação do Radix Sort

### 5.7.1 Complexidade Teórica

O *Radix Sort* é um algoritmo de ordenação não-comparativo que ordena os elementos analisando seus dígitos, da menor posição (menos significativa) até a maior (mais significativa), utilizando um algoritmo de ordenação estável como sub-rotina (geralmente o *Counting Sort*).

A ideia é ordenar os números várias vezes, uma para cada posição decimal, garantindo que ao final os números estejam completamente ordenados. (**??**)

- Sejam:
  - $n$: o número de elementos a serem ordenados,
  - $k$: o maior valor presente no vetor,
  - $d$: o número de dígitos do maior elemento (em base 10).
- Para cada dígito, o *Counting Sort* é aplicado com tempo $O(n+b)$, onde $b$ é o tamanho da base (geralmente 10).
- Como o algoritmo realiza $d$ passagens, a complexidade total será:

$$O(d \cdot (n+b))$$

- Como $d = \log_b(k)$, podemos reescrever como:

$$O(n \cdot \log k)$$

- Assumindo que $k$ é limitado por um inteiro de tamanho fixo (como 32 bits), o algoritmo pode ser considerado linear:

$$O(n)$$

**Espaço:** O Radix Sort não é in-place, pois utiliza memória auxiliar proporcional ao tamanho da entrada e à base.

**Estabilidade:** O algoritmo é estável, desde que a sub-rotina de ordenação utilizada também o seja (como o Counting Sort).

### 5.7.2 Resultados Experimentais

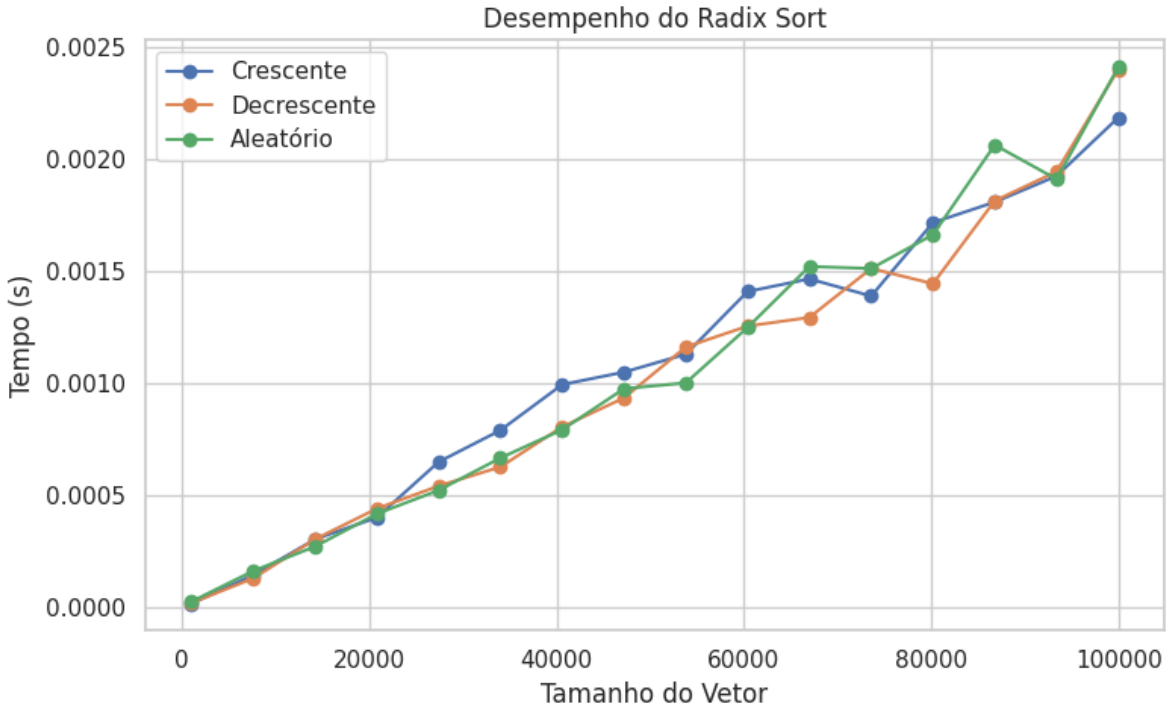


Figura 8: Tempo de execução do Radix Sort para diferentes entradas

# 6 Conclusão

Neste trabalho, foi realizada uma análise teórica e prática de seis algoritmos de ordenação: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort (com particionamentos de Hoare e Lomuto), Heap Sort e Radix Sort. Cada algoritmo foi implementado em linguagem C e avaliado quanto ao seu desempenho utilizando vetores de diferentes tamanhos e tipos de ordenação inicial (crescente, decrescente e aleatória).

## Considerações Teóricas

A análise teórica permitiu classificar os algoritmos em diferentes categorias:

- **Quadráticos:** Bubble Sort e Insertion Sort possuem complexidade $O(n^2)$ no pior caso, sendo mais adequados para vetores pequenos ou quase ordenados.

- **Divisão e Conquista:** Merge Sort e Quick Sort oferecem melhor desempenho assintótico ($O(n \log n)$), sendo amplamente utilizados em aplicações reais.
- **Baseados em Heap e Dígitos:** Heap Sort também apresenta $O(n \log n)$ no pior caso, enquanto Radix Sort pode atingir complexidade linear $O(n)$, desde que os dados sejam inteiros e com tamanho limitado.

### Resultados Experimentais

Os testes práticos confirmaram as expectativas teóricas:

- O **Bubble Sort** foi consistentemente o mais lento, especialmente para vetores grandes.
- O **Insertion Sort** apresentou bom desempenho em vetores quase ordenados, mas sofreu em vetores aleatórios ou decrescentes.
- **Merge Sort** e **Quick Sort** (em ambas as variações) apresentaram desempenho estável e eficiente, com Quick Sort geralmente sendo ligeiramente mais rápido, embora sensível à escolha do pivô.
- **Heap Sort** teve desempenho competitivo, mas com overhead de estruturação do heap.
- **Radix Sort** superou os demais em vetores grandes com números inteiros, destacando-se por sua abordagem não-comparativa e linear em muitos casos.

### Conclusão Final

A escolha do algoritmo ideal depende fortemente do contexto da aplicação. Para entradas pequenas ou quase ordenadas, algoritmos simples como o Insertion Sort são suficientes. Já para aplicações gerais e de grande escala, algoritmos como Merge Sort, Quick Sort ou Radix Sort se mostram mais apropriados. A análise experimental complementa a teoria e evidencia como fatores como a natureza dos dados e o custo de operações impactam diretamente na eficiência dos algoritmos na prática.

Trabalhos futuros poderiam incluir o estudo de algoritmos híbridos (como Timsort), variações paralelas ou o impacto da cache e otimizações de hardware no desempenho.

## Referências

GEEKSFORGEEKS. **Heap Sort**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/heap-sort/>.

GEEKSFORGEEKS. **Time and Space Complexity Analysis of Bubble Sort**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/>.

GEEKSFORGEEKS. **Time and Space Complexity Analysis of Merge Sort**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/>.

GEEKSFORGEEKS. **Time and Space Complexity Analysis of Quick Sort**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/>.

GEEKSFORGEEKS. **Time and Space Complexity of Insertion Sort Algorithm**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/>.

GEEKSFORGEEKS. **Time and Space Complexity of Radix Sort Algorithm**. 2021. Accessed: 2025-05-25. Disponível em: <https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/>.