UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

MARINA ESTER ONOFRE LOPES

ALGORITMOS DE ORDENAÇÃO

Natal – RN
2025

Sumário

1 INTRODUÇÃO

O objetivo deste trabalho é concluir com êxito a terceira unidade de Estrutura de Dados Básica 1 e garantir o aprendizado sobre os algoritmos de ordenação, que são fundamentais na organização e processamento de dados na programação.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Bubble Sort

```cpp
#include <vector>
#include "bubble.hpp"
using namespace std;

void bubble(vector<int>& v, int n){
    for(int j = 0; j < n - 1; j++){
        for(int i = 0; i < n - j; i++){
            if(v[i] > v[i+1]){
                swap(v[i], v[i+1]);
            }
        }
    }
}
```

O Bubble Sort compara elementos vizinhos e os troca de lugar se estiverem fora de ordem. O melhor caso ocorre quando o vetor já está ordenado (O(n)). O pior e o caso médio têm complexidade $O(n^2)$. A complexidade de espaço é O(1).

2.2 Insertion Sort

```cpp
#include <vector>
#include "insertion.hpp"
using namespace std;

void insertion(vector<int>& v, int n){
    for(int i = 1; i < n; i++){
        int novo = v[i];
        int fim = i - 1;
        while(novo <= v[fim] && fim >= 0){
            v[fim + 1] = v[fim];
            fim = fim -1;
        }
```

```
13          v[fim + 1] = novo;
14      }
15  }
```

Percorre o vetor e insere cada novo elemento na posição correta. Melhor caso: O(n); pior e médio: O(n^2). Espaço: O(1).

2.3 Selection Sort

```
1  #include "selection.hpp"
2  #include <vector>
3  using namespace std;
4
5  void selection(vector<int>& v, int n){
6      int menor;
7      for(int j = 0; j < n; j++){
8          for(int i = j; i < n; i++){
9              if(i == j){
10                 menor = i;
11             } else {
12                 if(v[i] < v[menor]){
13                     menor = i;
14                 }
15             }
16         }
17         if(v[menor] != v[j]){
18             swap(v[j], v[menor]);
19         }
20     }
21 }
```

Procura o menor elemento e troca com a posição correta. A complexidade é sempre O(n^2). Espaço: O(1).

2.4 Merge Sort

```
1  #include "merge.hpp"
2  #include <vector>
3  using namespace std;
4
5  void inter(vector<int>& v, int i, int meio, int f, int c, int t) {
6      vector<int> esquerda(v.begin() + i, v.begin() + meio + 1);
```

```
    vector<int> direita(v.begin() + meio + 1, v.begin() + f + 1);
    int i = 0, j = 0, k = i;

    while (i < esquerda.size() && j < direita.size()) {
        c++;
        if (esquerda[i] <= direita[j]) {
            v[k++] = esquerda[i++];
        } else {
            v[k++] = direita[j++];
            t++;
        }
    }
    while (i < esquerda.size()) v[k++] = esquerda[i++];
    while (j < direita.size()) v[k++] = direita[j++];
}

void merge(vector<int>& v, int i, int f, int c, int t) {
    if (i < f) {
        int meio = (i + f) / 2;
        merge(v, i, meio, c, t);
        merge(v, meio + 1, f, c, t);
        inter(v, i, meio, f, c, t);
    }
}
```

Baseado em dividir e conquistar. Complexidade: O(n log n) em todos os casos. Espaço: O(n).

2.5 Quick Sort

```
#include "quick.hpp"
#include <cstdlib>
#include <ctime>

static bool seed = false;

int particiona(vector<int>& v, int i, int f, int c, int t) {
    if (!seed) {
        srand(time(nullptr));
        seed = true;
    }

```

```
13      int ind = i + rand() % (f - i + 1);
14      swap(v[ind], v[f]);
15
16      int pivo = v[f];
17      int i = i - 1;
18
19      for (int j = i; j < f; ++j) {
20          c++;
21          if (v[j] <= pivo) {
22              i++;
23              swap(v[i], v[j]);
24              t++;
25          }
26      }
27
28      swap(v[i + 1], v[f]);
29      t++;
30      return i + 1;
31  }
32
33  void quick(vector<int>& v, int i, int f, int c, int t) {
34      if (i < f) {
35          int p = particiona(v, i, f, c, t);
36          quick(v, i, p - 1, c, t);
37          quick(v, p + 1, f, c, t);
38      }
39  }
```

Baseado em dividir e conquistar. Melhor e médio caso: O(n log n), pior caso: $O(n^2)$. Espaço: O(log n).

3 METODOLOGIA

Para gerar vetores aleatórios, quase ordenados ou invertidos, usou-se funções com `iota` e `shuffle`. As métricas de desempenho (tempo, comparações, trocas) foram medidas usando a biblioteca `chrono` e variáveis contadoras.

4 RESULTADOS

Após a execução dos algoritmos com vetores de diferentes tamanhos e distribuições (aleatórios, quase ordenados e inversos), foram obtidas as seguintes observações práticas:

- **Quick Sort**: apresentou o menor tempo de execução em praticamente todos os testes, especialmente com vetores grandes e aleatórios. O uso de pivô aleatório contribuiu para evitar o pior caso.
- **Merge Sort**: manteve desempenho constante com tempo um pouco acima do Quick Sort, porém com uso adicional de memória devido aos vetores auxiliares.
- **Insertion Sort**: teve bom desempenho com vetores quase ordenados, mas mostrou lentidão com dados aleatórios ou invertidos.
- **Selection Sort**: apresentou o mesmo número de comparações independentemente da ordem dos dados. O tempo de execução foi alto em todos os casos.
- **Bubble Sort**: foi o mais lento, principalmente com vetores grandes. Mesmo em vetores já ordenados, sua eficiência é limitada.

Além disso, foram coletadas as seguintes métricas para cada algoritmo:

- Número de comparações realizadas
- Número de trocas efetuadas
- Tempo de execução (em milissegundos)

Essas métricas foram fundamentais para validar, na prática, as análises teóricas de complexidade apresentadas na fundamentação teórica.

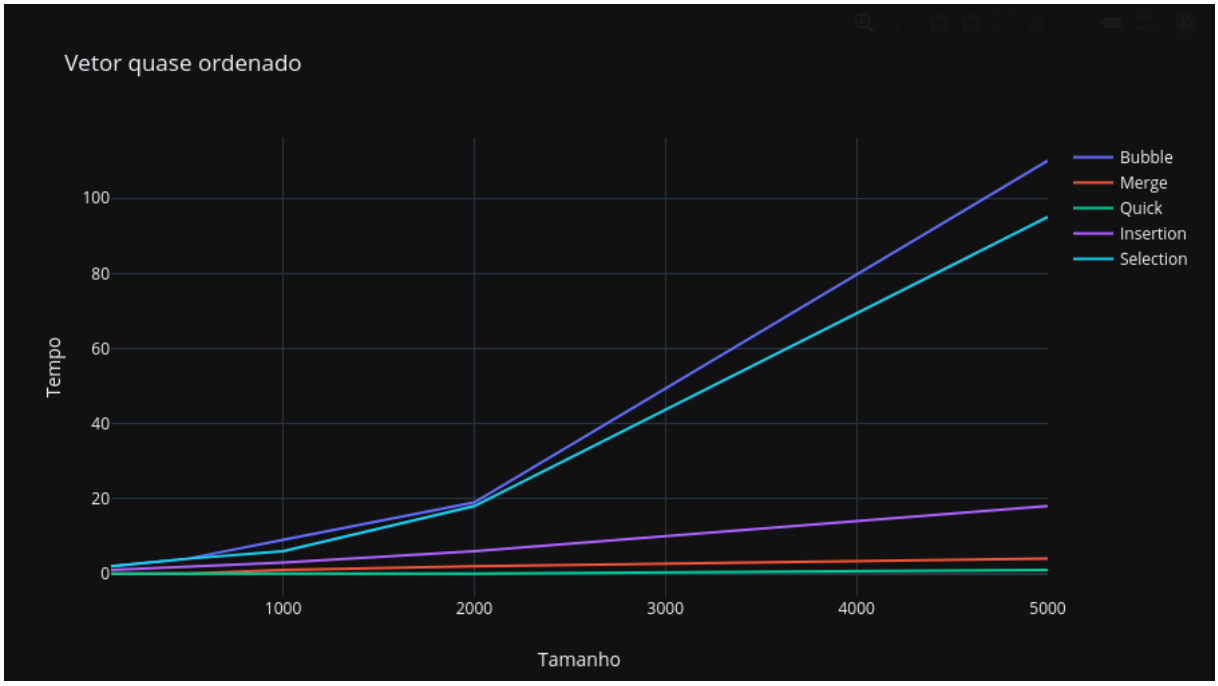

Figura 1: Desempenho dos algoritmos em vetor quase ordenado (tamanho 5000)

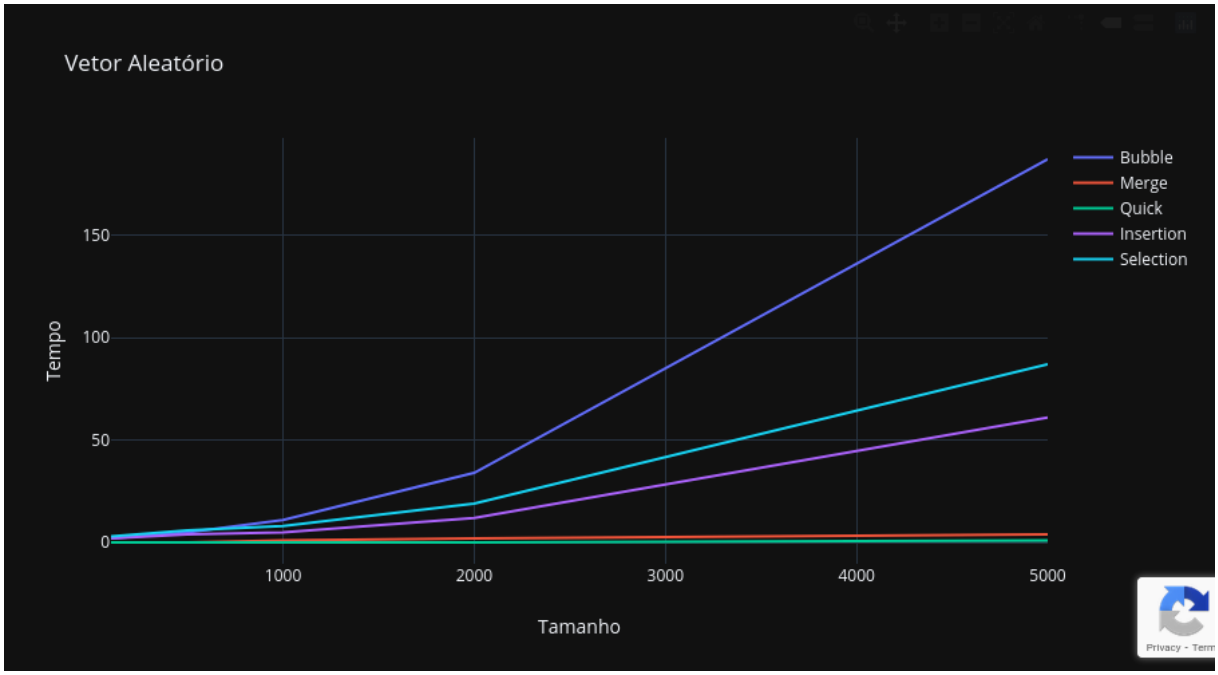

Figura 2: Desempenho dos algoritmos em vetor aleatório (tamanho 5000)

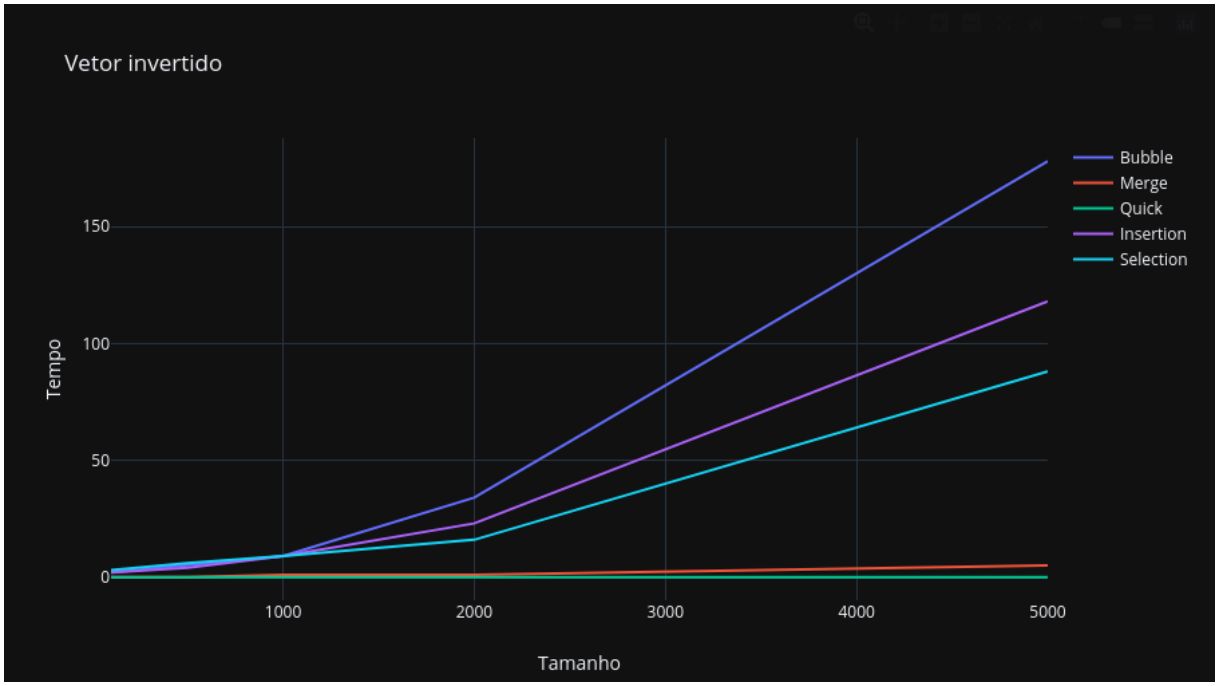

Figura 3: Desempenho dos algoritmos em vetor inverso (tamanho 5000)

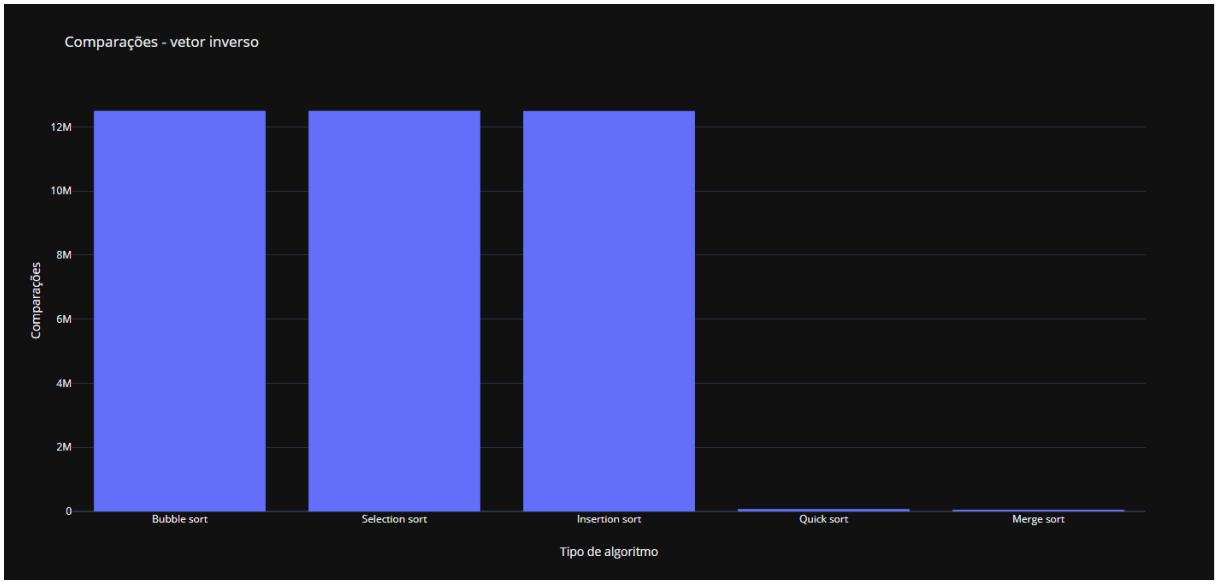

Figura 4: Comparações dos algoritmos em vetor aleatório

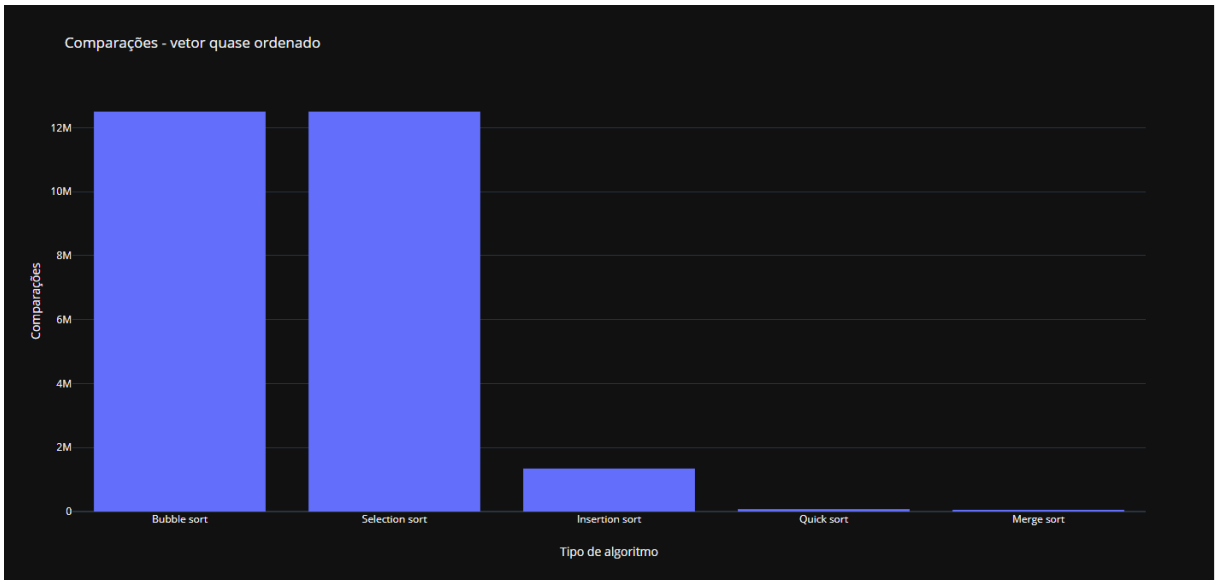

Figura 5: Comparações dos algoritmos em vetor quase ordenado

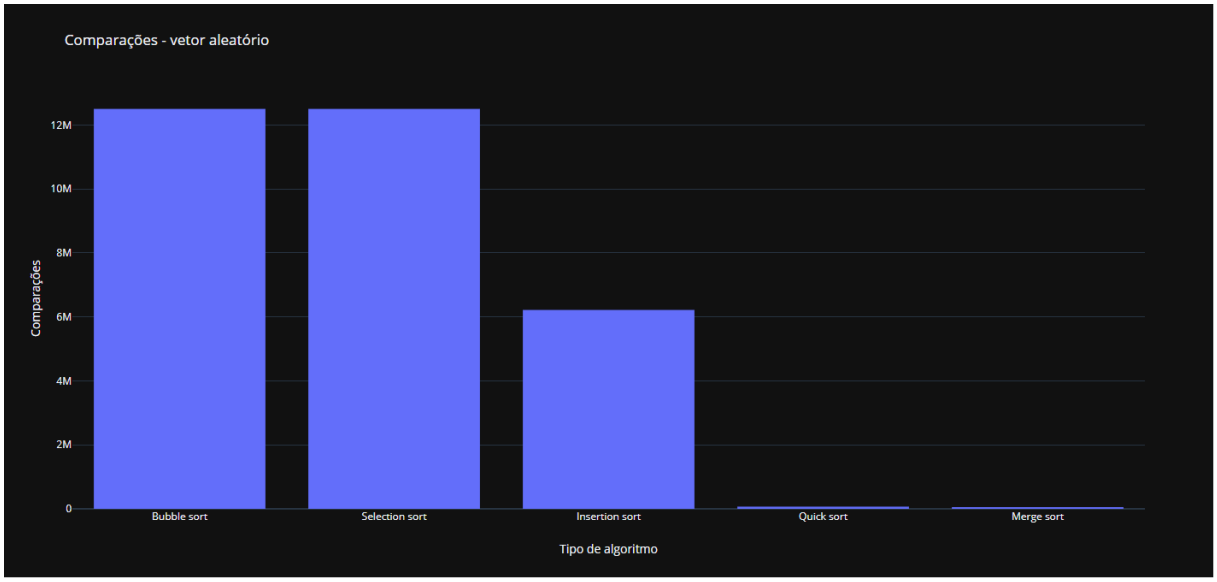

Figura 6: Comparações dos algoritmos em vetor inverso

5 RANKING DE DESEMPENHO

1. Quick Sort – mais eficiente na maioria dos casos
2. Merge Sort – estável e previsível
3. Insertion Sort – bom para dados quase ordenados
4. Selection Sort – simples, mas sempre $O(n^2)$
5. Bubble Sort – didático, mas ineficiente

6 CONCLUSÃO

O estudo comparativo dos algoritmos de ordenação demonstrou as diferenças de desempenho e aplicação de cada um, destacando a importância de saber escolher o algoritmo adequado conforme o cenário. O Quick Sort se mostrou o mais eficiente na maioria dos casos, enquanto o Bubble Sort é mais utilizado em contextos educacionais.

7 REFERÊNCIAS

- CORMEN, Thomas H. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Elsevier, 2013.
- GEEKSFORGEEKS. Bubble Sort. Disponível em: https://www.geeksforgeeks.org/bubble-sort/. Acesso em: 05 jul. 2025.