Algoritmos de Ordenação: Uma Visão Comparativa

Jhonatas Vinicius Neri dos Santos[1], **Caio Pessinni Dias**[1]

[1]Departamento de Computação – Universidade Federal do Espírito Santo (UFES)
Alegre – ES – Brasil

jhonatas.santos@edu.ufes.br, caio.p.dias@edu.ufes.br

Abstract. *This work aims to implement and compare different sorting algorithms, analyzing their performance and characteristics. The objective is to provide a detailed understanding of the main data sorting techniques, addressing aspects such as the number of comparisons, swaps performed, and execution time in different scenarios.*

Resumo. *Este trabalho tem como objetivo implementar e comparar diferentes algoritmos de ordenação, analisando seu desempenho e características. A proposta é fornecer uma compreensão detalhada das principais técnicas de classificação de dados, abordando aspectos como número de comparações, trocas realizadas e tempo de execução em diferentes cenários.*

1. Introdução

Este trabalho tem como objetivo principal avaliar e comparar, por meio de experimentação, o desempenho e a eficiência dos algorítimos: *Bolha, Bolha com critério de parada, Inserção direta, Inserção Binária, Seleção Direta, ShellSort, HeapSort, QuickSortIni, QuickSortCentro, QuickSortMediana, MergeSort, RadixSort, BucketSort*. A análise baseia-se na execução de testes com vetores de diferentes tamanhos, onde são coletados e comparados dados como tempos de execução, número de comparações realizadas e quantidade de movimentações necessárias para organizar os elementos. Embora os algoritmos de ordenação possam ser aplicados a qualquer estrutura linear, optou-se por utilizar exclusivamente vetores contendo números inteiros, de forma a facilitar a obtenção de dados quantitativos que subsidiem uma análise objetiva e detalhada.

Os testes iniciais foram realizados com um vetor contendo 100 elementos, e a análise contemplou três cenários distintos: (a) vetor já ordenado em ordem crescente; (b) vetor ordenado em ordem decrescente; e (c) vetor desordenado com valores aleatórios. Esse mesmo procedimento foi replicado utilizando vetores maiores, com 1.000 e 10.000 elementos. Com base nos resultados, busca-se identificar as principais vantagens e limitações de cada algoritmo, fornecendo dados para a escolha do mais adequado em diferentes contextos.

A estrutura deste trabalho está organizada da seguinte forma: na seção 2, são apresentados os algoritmos de ordenação abordados no estudo, acompanhados de suas respectivas implementações em Linguagem C. Na seção 3, são expostos os dados coletados durante os testes. Por fim, a seção 4 apresenta as conclusões obtidas, destacando os aspectos mais relevantes da comparação realizada.

2. Algoritmos de Ordenação

Algoritmos de ordenação são métodos utilizados para organizar ou rearranjar elementos de uma sequência, facilitando o acesso eficiente aos dados no futuro. Um dos principais objetivos desses algoritmos é ordenar vetores, pois uma única variável pode conter múltiplas posições, dependendo do tamanho do vetor definido.

2.1. *Bubble Sort*

De acordo com [Bissoli 2025] o algoritmo *Bubble Sort*percorre o vetor várias vezes, comparando elementos vizinhos e realizando trocas sempre que o primeiro for maior que o segundo. Esse processo continua até que todos os elementos estejam organizados em ordem crescente.

```
for (i = tamanho - 1; i >= 1; i--) {
    for (j = 0; j < i; j++) {
        if (dados[j] > dados[j + 1]) {
            aux = dados[j];
            dados[j] = dados[j + 1];
            dados[j + 1] = aux;
        }
    }
}
```

Algoritmo Bubble Sort [Bissoli 2025].

2.2. *Optimized Bubble Sort*

Segundo [Bissoli 2025] algoritmo *Optimized Bubble Sort* percorre o vetor diversas vezes, comparando elementos vizinhos e trocando-os caso estejam fora de ordem. No entanto, ele verifica se houve trocas durante a passagem; se nenhuma troca for feita, o processo é interrompido antecipadamente, pois o vetor já está ordenado.

```
while (mudou) {
    mudou = 0;
    for (j = 0; j < n; j++) {
        if (dados[j] > dados[j + 1]) {
            aux = dados[j];
            dados[j] = dados[j + 1];
            dados[j + 1] = aux;
            mudou = 1;
        }
    }
    n--;
}
```

Algoritmo Optimized Bubble Sort [Bissoli 2025].

2.3. *Insertion Sort*

Conforme [Geeksforgeeks 2025] o algoritmo de *Insertion Sort* organiza os elementos de um conjunto progressivamente. A cada nova etapa, ele seleciona um elemento e o compara com os que já foram ordenados, deslocando-os quando necessário.Esse processo

continua até que o elemento seja inserido na posição correta, garantindo a parte já analisada, ordenada.

```
// Funcao para realizar a ordenacao por insercao direta
void insertionSort(int *array, int tamanho) {
    // Percorre o vetor a partir do segundo elemento
    for (int i = 1; i < tamanho; i++) {
        int key = array[i];
        int j = i - 1;
        // Move os elementos maiores que a chave
        uma posicao para a direita
        while (j >= 0) {
            if (array[j] > key) {
                array[j + 1] = array[j];
                j--;
            } else {
                break;
            }
        }
        if (j != i - 1) {
            array[j + 1] = key;
        }
    }
}
```

Algoritmo Insertion Sort [Geeksforgeeks 2025].

2.4. *Binary Insertion Sort*

Conforme [Bissoli 2025] o *Binary Insertion Sort* é uma evolução do *Insertion Sort*. Ao invés de percorrer o vetor de forma sequencial, ele utiliza a busca binária para localizar rapidamente a posição onde o elemento deve ser inserido. Esse processo se repete até que todos os elementos estejam organizados no vetor.

```
// Funcao para realizar busca binaria
int binarySearch(int a[], int item, int low, int high){
        if (high <= low)
                return (item > a[low]) ? (low + 1) : low;
        int mid = (low + high) / 2;
        if (item == a[mid])
                return mid + 1;
        if (item > a[mid])
                return binarySearch(a, item, mid + 1, high);
        return binarySearch(a, item, low, mid - 1);
}
// Funcao para realizar ordenacao por insercao binaria
void binaryInsertionSort(int *array, int tamanho){
    int i, loc, j, selected;
    for (i = 1; i < tamanho; ++i){
        j = i - 1;
        selected = array[i];
        loc = binarySearch(array, selected, 0, j);
        while (j >= loc){
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = selected;
    }
}
```

Algoritmo Binary Insertion Sort [Geeksforgeeks 2025].

2.5. *Selection Sort*

Segundo [Bissoli 2025] o algoritmo de *Selection Sort* busca o menor elemento no vetor e o posiciona no início, trocando-o com o primeiro elemento. Em seguida, repete o processo para o restante do vetor, encontrando o menor valor e trocando-o com a próxima posição disponível, até que todos os elementos estejam ordenados.

```c
// Funcao para realizar a troca entre dois elementos no vetor
void swapSelection(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Funcao principal para realizar a ordenacao por selecao
void selectionSort(int *array, int tamanho) {
    for (int step = 0; step < tamanho - 1; step++) {
        int min_idx = step;
        for (int i = step + 1; i < tamanho; i++) {
            if (array[i] < array[min_idx]) {
                min_idx = i;
            }
        }
         if (min_idx != step) {
        swapSelection(&array[min_idx], &array[step]);
        }
    }
}
```

Algoritmo Selection Sort [Geeksforgeeks 2025].

2.6. *ShellSort*

Consoante a [Geeksforgeeks 2025] o *ShellsSort* é uma versão aprimorada do método de *Selection Sort*, na qual a lista é segmentada em subgrupos menores, e o algoritmo é aplicado a cada um deles. Com o tempo, o tamanho dos subgrupos diminui progressivamente até que todo o vetor esteja completamente ordenado.

```c
// Funcao para realizar a ordenacao por shell
void shellSort(int *dados, int tamanho) {
    // Ajusta o intervalo (gap) entre os elementos a
    serem comparados no vetor
    for (int gap = tamanho / 2; gap > 0; gap /= 2) {
        // Itera sobre os elementos no intervalo do "gap"
        for (int i = gap; i < tamanho; i++) {
            int temp = dados[i];
            int j;
            // Move elementos maiores que `temp` para
            frente no intervalo do "gap"
            for (j = i; j >= gap && dados[j - gap] > temp;
            j -= gap) {
                dados[j] = dados[j - gap];
            }
            dados[j] = temp;
        }
    }
}
```

Algoritmo ShellSort [Geeksforgeeks 2025].

2.7. *HeapSort*

De acordo com [Bissoli 2025] o algoritmo *HeapSort* é um método eficiente de ordenação que utiliza a estrutura de dados Heap, uma árvore binária completa que obedece à pro-

priedade do heap máximo (o valor de cada nó pai é maior ou igual ao de seus filhos). O processo começa transformando o vetor em um heap máximo, garantindo que o maior elemento fique na raiz. Em seguida, esse elemento é trocado com o último item do vetor e removido da estrutura de heap. O procedimento de reorganização do heap (heapify) é aplicado à raiz para restaurar a propriedade do heap, e essa extração do maior elemento se repete até que todos os itens estejam ordenados.

```
// Funcao para ajustar o heap para manter a propriedade
de heap
void heapify(int *array, int tamanho, int i) {
    int largest = i;
    int l = 2 * i + 1;          // Indice do filho a esquerda
    int r = 2 * i + 2;          // Indice do filho a direita
    if (l < tamanho) {
        if (array[l] > array[largest]) {
            largest = l;
        }
    }
    if (r < tamanho) {
        if (array[r] > array[largest]) {
            largest = r;
        }
    }
    // Se o maior nao for o no atual, troca os elementos
    e continua a ajustar
    if (largest != i) {
        int temp = array[i];
        array[i] = array[largest];
        array[largest] = temp;
        // Recursivamente ajusta a subarvore afetada
        heapify(array, tamanho, largest);
    }
}
// Funcao principal para realizar a ordenacao por heap
void heapSort(int *array, int tamanho) {
    // Constroi o heap maximizador (heapify em todos os
    nos nao-folha)
    for (int i = tamanho / 2 - 1; i >= 0; i--) {
        heapify(array, tamanho, i);
    }
    // Extrai elementos do heap um por um
    for (int i = tamanho - 1; i > 0; i--) {
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;
        heapify(array, i, 0);
    }
}
```

Algoritmo HeapSort [Geeksforgeeks 2025].

2.8. *QuickSort*

Em consonância com [Bissoli 2025] o algoritmo *QuickSort* seleciona um elemento como pivô e particiona o vetor em duas seções: os valores menores ficam à esquerda e os mai-

ores, à direita. Depois, repete esse procedimento de forma recursiva em cada subdivisão até que todo o vetor esteja ordenado.

2.8.1. *QuickSortIni*

O pivô é o primeiro elemento.

```
int particionar(int *array, int low, int high) {
    int pivot = array[low];
    int i = low;
    for (int j = low + 1; j <= high; j++) {
        if (array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    int temp = array[i];
    array[i] = array[low];
    array[low] = temp;
    return i;
}
void quicksortIni(int *array, int low, int high) {
    if (low < high) {
        int pivotIndex = particionar(array, low,
        high);
        quicksortIni(array, low, pivotIndex - 1);
        quicksortIni(array, pivotIndex + 1, high);
    }
}
void quicksortIniWrapper(int *array, int tamanho) {
    quicksortIni(array, 0, tamanho - 1);
}
```

Algoritmo QickSortIni [Geeksforgeeks 2025].

2.8.2. *QuickSortCentro*

O pivô é o elemento central.

```c
void quicksortCentro(int *array, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        int pivot = array[mid];
        int temp = array[low];
        array[low] = array[mid];
        array[mid] = temp;
        int i = low + 1;
        int j = high;
        while (i <= j) {
            while (i <= high && array[i] <= pivot) {
                i++;
            }
            while (array[j] > pivot) {
                j--;
            }
            if (i < j) {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        array[low] = array[j];
        array[j] = pivot;
        quicksortCentro(array, low, j - 1);
        quicksortCentro(array, j + 1, high);
    }
}
void quicksortCentroWrapper(int *array, int tamanho) {
    quicksortCentro(array, 0, tamanho - 1);
}
```

Algoritmo QickSortCentro [Geeksforgeeks 2025].

2.8.3. *QuickSortMediana*

O pivô é a mediana do primeiro elemento, elemento central e ultimo elemento.

```c
void quicksortMediana(int *array, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        int a = array[low], b = array[mid], c = array[high];
        int pivotIndex = (a < b) ?
        ((b < c) ? mid : ((a < c) ? high : low))
        : ((a < c) ? low : ((b < c) ? high : mid));
        int pivot = array[pivotIndex];
        int temp = array[low];
        array[low] = array[pivotIndex];
        array[pivotIndex] = temp;
        int i = low + 1;
        int j = high;
        while (i <= j) {
            while (i <= high && array[i] <= pivot) {
                i++;
            }
            while (array[j] > pivot) {
                j--;
            }
            if (i < j) {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        array[low] = array[j];
        array[j] = pivot;
        quicksortMediana(array, low, j - 1);
        quicksortMediana(array, j + 1, high);
    }
}
void quicksortMedianaWrapper(int *array, int tamanho) {
    quicksortMediana(array, 0, tamanho - 1);
}
```

Algoritmo QickSortMediana [Geeksforgeeks 2025].

2.9. *MergeSort*

Segundo [Bissoli 2025] o algoritimo *MergeSort* divide o vetor em partes menores, ordena cada uma separadamente e, em seguida, combina as partes ordenadas para formar um único vetor organizado.

```c
// Funcao para mesclar (merge) duas sublistas ordenadas
void merge(int *array, int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = array[left + i];
    for (j = 0; j < n2; j++)
        R[j] = array[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        array[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        array[k] = R[j];
        j++;
        k++;
    }
}
// Funcao recursiva para dividir o vetor e realizar a
mescla das listas
void mergeSortHelper(int *array, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortHelper(array, left, mid);
        mergeSortHelper(array, mid + 1, right);
        merge(array, left, mid, right);
    }
}
// Funcao principal para realizar a ordenacao por merge
void mergeSort(int *array, int tamanho) {
    mergeSortHelper(array, 0, tamanho - 1);
}
```

Algoritmo MergeSort [Geeksforgeeks 2025].

2.10. *RadixSort*

Consoante ao [Bissoli 2025] o algoritimo *RadixSort* organiza os elementos ordenando-os com base em seus dígitos individuais, do menos significativo, para o mais significativo. Esse processo é repetido para cada casa decimal até que todos os dígitos

tenham sido processados, garantindo a ordenação completa do conjunto de dados.

```
// Funcao para encontrar o maior elemento no vetor
int getMax(int *array, int tamanho) {
    int max = array[0];
    for (int i = 1; i < tamanho; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
// Funcao Counting Sort para um digito especifico
void countingSort(int *array, int tamanho, int place) {
    int output[tamanho];     // Vetor para armazenar
    os resultados ordenados
    int count[10] = {0};     // Contador para armazenar
    // Conta a ocorrencia de cada digito no lugar
    atual (unidade, dezena, etc.)
    for (int i = 0; i < tamanho; i++) {
        int index = (array[i] / place) % 10;
        // Obtem o digito no lugar atual
        count[index]++;
    }
    // Calcula os indices acumulados para os digitos
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    // Constroi o vetor de saida
    for (int i = tamanho - 1; i >= 0; i--) {
        int index = (array[i] / place) % 10;
        output[count[index] - 1] = array[i];
        count[index]--;
    }
    // Copia os elementos do vetor de saida de
    volta para o array original
    for (int i = 0; i < tamanho; i++) {
        array[i] = output[i];
    }
}
// Funcao principal para realizar a ordenacao por radix
void radixSort(int *array, int tamanho) {
    int maxElement = getMax(array, tamanho);
    // Aplica Counting Sort para cada digito (unidade,
    dezena, centena, etc.)
    for (int place = 1; maxElement / place > 0; place *= 10) {
        countingSort(array, tamanho, place);
    }
}
```

Algoritmo RadixSort [Geeksforgeeks 2025].

2.11. *BucketSort*

De acordo com [Bissoli 2025], o algoritmo *BucketSort* opera segmentando um vetor em uma quantidade definida de baldes ou compartimentos. Cada balde é or-

denado de forma independente, usando o algoritmo *Optimized Bubble Sort*. Após a ordenação de todos os baldes, eles são unidos para criar o vetor final ordenado.

```
void bucketSort(int* V, int tamanho) {
    int maior = 0;
    for (int i = 0; i < tamanho; i++) {
        if (V[i] > maior) {
            maior = V[i];
        }
    }
    int numeroBaldes = (maior / 10) + 1;
    Balde baldes[numeroBaldes];
    for (int i = 0; i < numeroBaldes; i++) {
        baldes[i].topo = 0;
    }
    for (int i = 0; i < tamanho; i++) {
        int idx = V[i] / (maior / numeroBaldes + 1);
        if (baldes[idx].topo < 10) {
            baldes[idx].balde[baldes[idx].topo++] = V[i];
        }
    }
    for (int i = 0; i < numeroBaldes; i++) {
        if (baldes[i].topo > 0) {
            ordenaBolhaComParada(baldes[i].balde, baldes[i].topo);
        }
    }
    for (int i = 0, j = 0; j < numeroBaldes; j++) {
        for (int k = 0; k < baldes[j].topo; k++) {
            V[i++] = baldes[j].balde[k];
        }
    }
}
```

Algoritmo BucketSort [Bissoli 2025].

3. Resultados e Discussões

A codificação foi realizada utilizando a linguagem C, e o ambiente de desenvolvimento escolhido foi o Visual Studio Code. Em relação ao hardware, os testes foram realizados em uma máquina com as seguintes configurações:

- Fabricante: Dell;
- Modelo: Dell Optiplex 790;
- Processador: AMD Ryzen 3 3200G with Radeon Vega Graphics;
- Frequência: 3.60 GHz;
- Memória 16,0 GB ddr4 3200MHz;
- Sistema Operacional: Windows 10 Pro (x64);

3.1. Teste em um Vetor de 1000 Elementos

A Tabela 1 abaixo exibe os valores médios de desempenho dos algoritmos ao trabalhar com o vetor [1000], considerando listas ordenadas de maneira crescente, decrescente e aleatória.

Tabela 1. Teste em um Vetor de 1000 Elementos

Vetor[1000]									
	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmos	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**
Bubble Sort	0.001	499500	0	0.002	499500	499500	0.003	499500	249476
Optimized Bubble Sort	0.0000	999	0	0.003	499500	499500	0.002	497085	249476
Insertion Sort	0.0000	999	0	0.002	499500	499500	0.001	250465	249476
Binary Insertion Sort	0.0000	7987	0	0.003	8977	499500	0.001	8555	249476
Shell Sort	0.0000	8006	0	0.0000	11716	4700	0.001	15261	7766
Selection Sort	0.001	499500	0	0.002	499500	500	0.004	499500	992
Heap sort	0.0000	17583	9708	0.0000	15965	8316	0.0000	16857	9070
Quick Sort Ini	0.002	499500	999	0.002	499500	250999	0.001	10294	5935
Quick Sort Centro	0.0000	8498	1022	0.0000	9038	1903	0.0000	11685	3033
Quick Sort Med	0.0000	10031	1022	0.001	127749	1499	0.0000	11144	2974
Merge sort	0.0000	5044	9976	0.0000	4932	9976	0.0000	8708	9976
Radix Sort	0.0000	0	3000	0.0000	0	3000	0.001	0	3000
Bucket Sort	0.0000	9062	4202	0.0000	15562	9702	0.0000	6162	3202

Percebe-se que para um vetor previamente ordenado em ordem crescente, os algoritmos *Inseção Direta*, *Shell Sort* e *Selection Sort* apresentam um desempenho altamente eficiente. O nsertion Sort, em particular, destaca-se por realizar apenas 999 comparações e nenhuma troca, demonstrando sua adequação para esse tipo de entrada. Em contraste, algoritmos como o *Merge Sort* e o *Heap Sort* executam um número elevado de operações desnecessárias, pois suas abordagens estruturais não conseguem aproveitar a ordenação prévia do vetor, resultando em um desempenho inferior nesse contexto.

Quando a entrada está em ordem decrescente, a eficiência dos algoritmos muda consideravelmente. O *Shell Sort* se sobressai, pois sua estratégia de inserção com intervalos reduz significativamente o número de comparações e trocas necessárias para reorganizar os elementos. O *Heap Sort*, apesar de realizar um número considerável de comparações, ainda mantém um desempenho competitivo. Em contrapartida, os algoritmos *Bubble Sort* (tanto na versão com parada quanto sem parada) e o *Quick Sort* em sua versão inicial se mostram ineficientes, pois dependem de um grande número de operações para inverter completamente a ordenação dos elementos.

No caso de um vetor com *ordem aleatória*, os algoritmos baseados na estratégia de divisão e conquista, como *Merge Sort* e *Quick Sort* (nas versões Central e Mediana), demonstram um bom desempenho ao equilibrar o número de comparações e trocas. O *Heap Sort* também se mostra uma opção eficiente. Em contraste, o *Bubble Sort*, especialmente em sua versão sem parada, apresenta o pior desempenho, evidenciado pelo alto número de comparações e trocas realizadas, tornando-o inadequado para esse tipo de entrada.

De maneira geral, os algoritmos *Shell Sort* e *Quick Sort* (em suas versões otimizadas) se destacam como opções altamente eficientes para diferentes cenários, combinando um número reduzido de operações com tempos de execução competitivos. Em contrapartida, algoritmos mais simples, como o *Bubble Sort* e o *Selection Sort*, tendem a apresentar um desempenho insatisfatório em vetores de grande porte ou com ordenação desfavorável,

reforçando a importância da escolha adequada do algoritmo conforme as características dos dados de entrada.(O Radix apresenta 0 comparações pois se trata de um algoritimo não comparativo)

3.2. Teste em um Vetor de 10000 Elementos

A Tabela 2 abaixo exibe os valores médios de desempenho dos algoritmos ao trabalhar com o vetor [10000], considerando listas ordenadas de maneira crescente, decrescente e aleatória.

Tabela 2. Teste em um Vetor de 10000 Elementos

Vetor[10000]									
	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmos	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**
Bubble Sort	0.169	4995000	0	0.270	49995000	49995000	0.313	49995000	24807181
Optimized Bubble Sort	0.0000	9999	0	0.290	49995000	49995000	0.282	49992722	24807181
Insertion Sort	0.0000	9999	0	0.171	49995000	49995000	0.085	24817176	24807181
Binary Insertion Sort	0.0000	113631	0	0.136	123617	49995000	0.065	119034	24807181
Shell Sort	0.0000	120005	0	0.001	172578	62560	0.002	262997	148057
Selection Sort.	0.117	4995000	0	0.133	49995000	5000	0.136	49995000	9991
Heap Sort	0.002	244460	131956	0.002	226682	116696	0.002	235391	124277
Quick Sort Ini	0.119	49995000	9999	0.219	49995000	25009999	0.001	160080	92068
Quick Sort Centro	0.0000	119535	11808	0.001	124040	19359	0.001	166272	38335
Quick Sort Med	0.001	137247	11808	0.033	12527499	14999	0.001	158012	37461
Merge Sort	0.001	69008	133616	0.001	64608	133616	0.002	120404	133616
Radix Sort	0.0000	0	40000	0.001	0	40000	0.001	0	40000
Bucket Sort	0.001	106485	52181	0.0000	171485	107181	0.0000	77485	42181

A análise dos algoritmos de ordenação demonstra que o *Insertion Sort* é o mais eficiente para vetores já ordenados, pois realiza o menor número de comparações e nenhuma troca, identificando rapidamente que o vetor já está ordenado. Em contrapartida, o *Merge Sort* e o *Heap Sort* continuam realizando trocas, tornando-se menos eficientes nesse cenário. Além disso, o *Bubble Sort* se destaca negativamente por executar um número excessivo de comparações, tornando-o inadequado para vetores organizados.

Quando o vetor está ordenado de forma decrescente, o *Shell Sort* apresenta o melhor desempenho, pois sua abordagem de inserção com distanciamento reduz o número de comparações e trocas. O *Bubble Sort* e o *Quick Sort*, por outro lado, demonstram grande ineficiência, percorrendo todo o vetor repetidamente e realizando um alto número de operações. Já na ordenação de um vetor aleatório, o *Shell Sort* novamente se destaca por equilibrar bem comparações e trocas, enquanto o *Bubble Sort* continua sendo o pior algoritmo devido ao seu alto custo computacional.

De forma geral, o *Shell Sort* mostra-se um dos algoritmos mais eficientes para diferentes tipos de ordenação. O *Insertion Sort* é excelente para listas quase ordenadas, enquanto o *Radix Sort* e o *Bucket Sort* possuem desempenho muito rápido, mas dependem do tipo de dado para serem aplicáveis. Em contraste, o *Bubble Sort* se confirma como a opção menos eficiente na maioria dos cenários, sendo evitado em aplicações que exigem alta performance.(O Radix apresenta 0 comparações pois se trata de um algoritimo não comparativo)

3.3. Teste em um Vetor de 100000 Elementos

A Tabela 3 abaixo exibe os valores médios de desempenho dos algoritmos ao trabalhar com o vetor [100000], considerando listas ordenadas de maneira crescente, decrescente e aleatória.

Tabela 3. Teste em um Vetor de 100000 Elementos

Vetor[100000]									
	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmos	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**	**Tempo(s)**	**Comp**	**Trocas**
Bubble Sort	14.258	4999950000	0	29.087	4999950000	4999950000	37.828	4999950000	2498759392
Optimized Bubble Sort	0.0000	99999	0	29.854	4999950000	4999950000	34.700	4999831172	2498759392
Insertion Sort	0.0000	99999	0	17.426	4999950000	4999950000	12.140	2498859376	2498759392
Binary Insertion Sort	0.009	1468946	0	13.927	1568929	4999950000	10.703	1522636	2498759392
Shell Sort	0.01	1500006	0	0.012	2244585	844560	0.0340	4535560	3086108
Selection Sort	12.921	4999950000	0	16.122	4999950000	50000	12.947	4999950000	99988
Heap Sort	0.0260	3112517	1650854	0.022	2926640	1497434	0.0300	3019269	1574664
Quick Sort Ini	12.851	4999950000	99999	22.335	4999950000	2500099999	0.0190	2017171	1109509
Quick Sort Centro	0.006	1534481	131070	0.005	1575498	196211	0.0150	2057795	458135
Quick Sort Med	0.005	1731086	131070	3.209	1250274999	149999	0.0140	1923837	457543
Merge Sort	0.014	853904	1668928	0.014	815024	1668928	0.0240	1536528	1668928
Radix Sort	0.007	0	500000	0.008	0	500000	0.0080	0	500000
Bucket Sort	0.004	1081560	532215	0.005	1731560	1082215	0.0060	791560	432215

A análise dos algoritmos de ordenação aplicados a um vetor de 100.000 elementos revela diferenças significativas de desempenho dependendo da disposição inicial dos dados. Quando o vetor já está ordenado, o *Insertion Sort* se destaca como o mais eficiente, pois realiza o menor número de comparações e nenhuma troca. Em contrapartida, o *Merge Sort* apresenta um desempenho inferior, pois, independentemente da ordenação prévia, divide o vetor recursivamente e executa diversas trocas desnecessárias. Algoritmos como *Shell Sort*, *Heap Sort* e *Quick Sort* também demonstram boa eficiência nesse cenário, mas não superam o *Insertion Sort*.

Quando o vetor está em ordem decrescente, o *Shell Sort* obtém o melhor resultado devido à sua abordagem baseada em inserções espaçadas, reduzindo rapidamente a desordem inicial. Já o *Bubble Sort* e o *Quick Sort* têm um desempenho ruim, com alto número de comparações e trocas. O *Quick Sort*, embora rápido em média, sofre nesse caso devido à escolha ineficiente do pivô, resultando em uma grande quantidade de operações.

Para vetores com elementos aleatórios, o *Shell Sort* novamente se sobressai, oferecendo um bom equilíbrio entre comparações e trocas, tornando-se a melhor opção. O *Bubble Sort*, por outro lado, se confirma como o pior algoritmo, exigindo um número excessivo de operações para ordenar os elementos. O *Quick Sort*, apesar de geralmente ser rápido, ainda realiza muitas comparações e trocas, demonstrando que sua eficiência pode variar. (O Radix apresenta 0 comparações pois se trata de um algoritimo não comparativo)

4. Conclusão

A análise dos algoritmos de ordenação em diferentes cenários evidencia que a escolha do método mais adequado depende diretamente das características dos dados de entrada, como tamanho e nível de ordenação prévia. O *Bubble Sort*, apesar de sua implementação

simples e intuitiva, mostrou-se altamente ineficiente para conjuntos grandes devido ao elevado número de comparações e trocas.

O *Insertion Sort* apresentou bom desempenho para vetores pequenos e quase ordenados, destacando-se pela simplicidade e eficiência nesses casos. No entanto, seu alto número de trocas limita sua aplicabilidade em conjuntos de maior porte. O *Insertion Sort Binário*, ao reduzir o número de comparações por meio de busca binária, ainda enfrenta a limitação do grande número de movimentações de elementos. Já o *Selection Sort*, embora realize um número fixo de comparações independentemente da disposição inicial dos dados, executa menos trocas que o *Insertion Sort*, sendo útil em situações em que minimizar operações de escrita é um fator crítico.

Entre os algoritmos baseados na estratégia de divisão e conquista, o *Merge Sort* demonstrou eficiência consistente para grandes volumes de dados, porém com a desvantagem do alto consumo de memória adicional. O *Quick Sort*, por sua vez, destacou-se pelo excelente desempenho médio, mas sua eficiência depende diretamente da escolha do pivô, sendo que a versão que seleciona o primeiro elemento como pivô apresentou um desempenho significativamente inferior em vetores já ordenados.

O *Shell Sort* mostrou-se uma alternativa robusta, equilibrando bem o número de comparações e trocas, além de ser eficiente para diferentes tamanhos de entrada. Sua capacidade de reduzir rapidamente a desordem inicial torna-o uma opção viável para diversas aplicações.

Além disso, os algoritmos *Heap Sort*, *Radix Sort* e *Bucket Sort* apresentaram características distintas. O *Heap Sort* demonstrou estabilidade em termos de desempenho e não requer espaço adicional significativo, embora, em algumas situações, tenha sido superado pelo *Quick Sort* em velocidade. O *Radix Sort*, por sua natureza não comparativa, revelou-se altamente eficiente para determinados tipos de dados, operando em tempo linear. O *Bucket Sort*, embora eficiente, depende de uma boa estratégia de distribuição dos dados, o que pode afetar sua performance em determinados cenários.

Dessa forma, não há um algoritmo de ordenação universalmente superior. A escolha da abordagem mais adequada deve considerar não apenas o tamanho e a organização dos dados, mas também fatores como consumo de memória, número de trocas e eficiência computacional para cada situação específica.

Referências

Bissoli, D. C. (2025). Notas de aula.

Geeksforgeeks (2025). Geeksforgeeks. `https://www.geeksforgeeks.org/`.