Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Pedro Augusto Simões da Cruz - 8116

"PROJETO DE ALGORITMOS"

Rio Paranaíba - MG

2023

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Pedro Augusto Simões da Cruz" - "8116"

"PROJETO DE ALGORITMOS"

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2023

1 RESUMO

Os algoritmos de ordenação apresentam grande importância no estudo teórico de algoritmos,bem como em aplicações rotineiras e práticas do cotidiano. Diante dessa grande importância ao longo dos anos surgem alguns algoritmos de ordenação na literatura, onde são analisados casos em que cada algoritmo se enquadra. Sendo assim se torna necessário um estudo para analisar o comportamento desses algoritmos, bem como suas complexidades para indicar qual é o mais adequado para cada tipo de problema.

Os algoritmos de ordenação por troca, como Bubble Sort, Insertion Sort, Merge Sort e Selection Sort, desempenham um papel fundamental na organização de elementos. Um fator importante é a análise de complexidade, através da notação Big O, que permite avaliar o desempenho em termos de tempo e espaço. Para conjuntos menores (10, 100, 1000 elementos), algoritmos simples como Bubble Sort podem ser eficazes, por ter sua complexidade $O(n^2)$, mas à medida que os conjuntos crescem (10000, 100000, 1000000 elementos), algoritmos mais eficientes, como Merge Sort e Quick Sort, que atuam sob o paradigma de divisão e conquista se destacam, especialmente com complexidades O(n log n). Assim, a escolha do algoritmo depende do tamanho do conjunto de entrada, sendo essencial considerar a eficiência assintótica para garantir um ótimo desempenho em diferentes escalas de entrada.

Sumário

2 INTRODUÇÃO

Em várias situações do nosso dia-a-dia nos deparamos com a necessidade de trabalharmos com dados/informações devidamente ordenadas, como, por exemplo, ao procurar um contato na lista telefônica, imagine como seria difícil se estes nomes não estivessem em ordem alfabética? Então não é difícil perceber que as atividades que envolvem algum método de ordenação estão muito presentes na computação.

Para a área da computação, os algoritmos são procedimentos computacionais bem especificados, que seguem instruções específicas, utilizando valores tomados como entrada, para produzir valores como saída. Neste relatório, exploraremos os algoritmos de ordenação, divididos essencialmente em duas categorias distintas: algoritmos de troca e algoritmos de divisão e conquista.

Os algoritmos de troca são baseados em repetidas trocas de elementos adjacentes até que o conjunto de elementos esteja devidamente ordenado, geralmente são algoritmos com implementação mais simples. Os algoritmos de troca que serão vistos neste relatório são: Insertion Sort, Bubble Sort, Selection Sort e Shell Sort.

Os algoritmos de divisão e conquista são algoritmos que atuam sob o paradigma de divisão e conquista, que conquiste em dividir um problema em problemas menores, resolver estes problemas menores e recombiná-los, a fim de solucionar o problema inicial, estas etapas são conhecidas como: dividir, conquistar e combinar. Devido ao uso de recursão, a implementação desse tipo de algoritmo é levemente mais trabalhosa que a implementação de um algoritmo de troca. Os algoritmos de divisão e conquista que serão vistos neste relatório são: Merge Sort, Quick Sort e Heap Sort.

Para analisar devidamente um algoritmo, é necessário estudar sua complexidade, que é uma medida que avalia a eficiência do algoritmo, especialmente em relação a seu tempo de execução. Esta complexidade será expressa através da notação "Big O"(O), de modo que, quanto mais complexa for a operação dentro dos parêntes após o "O", maior será a

complexidade do algoritmo. Por exemplo, O(1) denota um algoritmo de tempo constante (onde o tempo de execução é o mesmo, independentemente do tamanho da entrada), enquanto O(n) denota um algoritmo de tempo linear (o tempo de execução cresce em proporção linear à entrada). Ao longo do relatório, serão apresentados códigos, com o custo de suas linhas e o número de vezes que cada linha foi executada. Através do cálculo dos custos, é possível determinar a complexidade e o tempo de execução de um determinado algoritmo.

O processo para determinar a complexidade de alguma função, geralmente utilizado em alguma recorrência, é denominado análise assintótica.

A análise e estudo de algoritmos é essencial para a área de computação como um todo. Proporciona grande conhecimento sobre a eficiência em diferentes abordagens computacionais. O relatório tratará sobre algoritmos de ordenação, análises de complexidade e análises assintóticas, além disso, a eficiência dentre os algoritmos será comparada e ilustrada através de tabelas e gráficos.

3 ALGORITMOS

3.1 Insertion Sort

O algoritmo Insertion Sort (também chamado de ordenação por inserção) é um algoritmo que constrói uma sequência ordenada dos elementos de um array, elemento por elemento. Funciona de maneira similar à como organizamos as cartas de um baralho na mão: um elemento é retirado da entrada e inserido na posição correta em uma lista que está parcialmente ordenada.

O algoritmo funciona da seguinte maneira: recebe uma lista de elementos e, a cada iteração, o algoritmo considera um elemento da lista e o compara com os demais elementos à esquerda, desloca os elementos maiores para a direita, criando uma nova posição onde o elemento pode ser inserido. Tal inserção é repetida para todos os elementos, até que a lista esteja completamente ordenada.
O pseudocódigo está apresentado abaixo, na Figura 1:

```
INSERTION-SORT(A)
1 for j ← 2 to comprimento[A]
2       do chave ← A[j]
3          ▷ Inserir A[j] na seqüência ordenada A[1..j − 1].
4          i ← j − 1
5          while i > 0 e A[i] > chave
6             do A[i + 1] ← A[i]
7                i ← i − 1
8          A[i + 1] ← chave
```

Figura 1: Pseudocódigo do Insertion Sort

A seguir está um exemplo de implementação do algoritmo Insertion Sort, feito pelo autor utilizando linguagem C, e um teste de mesa, montado com o código em questão como base.

```
        #include <stdio.h>
        #include <stdlib.h>

        void inS(int *v, int tam){
        int aux, j = i;
        aux = v[j];
        while(j>0 && aux<v[j-1]){
        v[j] = v[j-1];
          j --;
        }
        v[j] = aux;
    }
}
```

Passo	**Índice (j)**	**Chave (aux)**	**Vetor antes de ser ordenado**	**Comparações e trocas**	**Vetor após ordenação**
Inicial	-	-	$\{5, 2, 4, 6, 1, 3\}$	-	$\{5, 2, 4, 6, 1, 3\}$
1	1	2	$\{5, 2, 4, 6, 1, 3\}$	$2 < 5 \rightarrow$ move 5 uma posição à direita	$\{2, 5, 4, 6, 1, 3\}$
2	2	4	$\{2, 5, 4, 6, 1, 3\}$	$4 < 5 \rightarrow$ move 5 uma posição à direita	$\{2, 4, 5, 6, 1, 3\}$
3	3	6	$\{2, 4, 5, 6, 1, 3\}$	$6 \geq 5 \rightarrow 6$ já está na posição correta	$\{2, 4, 5, 6, 1, 3\}$
4	4	1	$\{2, 4, 5, 6, 1, 3\}$	$1 < 6 \rightarrow$ move 6, 5, 4, 2 à direita	$\{1, 2, 4, 5, 6, 3\}$
5	5	3	$\{1, 2, 4, 5, 6, 3\}$	$3 < 6$, $3 < 5$, $3 < 4 \rightarrow$ move 6, 5, 4	$\{1, 2, 3, 4, 5, 6\}$

3.2 Bubble Sort

O algoritmo Bubble Sort, também chamado de ordenação por flutuação, é um algoritmo de ordenação cuja ideia principal é percorrer um conjunto de elementos diversas vezes, e a cada vez, transportar para o topo o maior elemento da sequência. O algoritmo recebe este nome pois seu funcionamento lembra o de uma bolha, que flutua até o topo de uma superfície, assim como o elemento "flutua"até o topo da sequência.

O algoritmo funciona da seguinte maneira: recebe uma lista de elementos e compara o primeiro elemento com os elementos seguintes, e caso encontre um valor que seja menor que o do primeiro elemento, realiza uma troca de posições entre estes, prosseguindo até que o maior valor esteja na última posição. Tal troca está embutida em um laço de repetição, que é executado até que todos os elementos estejam em suas devidas posições.
O algoritmo está apresentado abaixo, na Figura 2:

```
BUBBLESORT(A)
1 for i ← 1 to comprimento[A]
2       do for j ← comprimento[A] downto i + 1
3             do if A[j] < A[j − 1]
4                   then trocar A[j] ↔ A[j − 1]
```

Figura 2: Pseudocódigo do Bubble Sort

A seguir está um exemplo de implementação do algoritmo Bubble Sort, feito pelo autor utilizando linguagem C, e um teste de mesa, montado com o código em questão como base.

```
#include <stdio.h>
#include <stdlib.h>
```

```
void bubS(int *v, int tam){

int i, j, aux;

for (j=0; j<tam; j++){

    for(i=0; i<tam-1; i++){

                if(v[i] > v[i+1]){

                                    int aux = v[i];
                                    v[i] = v[i+1];
                                    v[i+1] = aux;
                 }
    }
}tam--;
}
```

Passo	j	i	aux	**Estado do Vetor (v)**
1	0	0	5	[3, 5, 8, 6, 2]
2	0	1	-	[3, 5, 8, 6, 2]
3	0	2	8	[3, 5, 6, 8, 2]
4	0	3	8	[3, 5, 6, 2, 8]
5	1	0	-	[3, 5, 6, 2, 8]
6	1	1	-	[3, 5, 6, 2, 8]
7	1	2	6	[3, 5, 2, 6, 8]
8	2	0	-	[3, 5, 2, 6, 8]
9	2	1	5	[3, 2, 5, 6, 8]
10	3	0	3	[2, 3, 5, 6, 8]

Tabela 1: Teste de mesa para o algoritmo Bubble Sort com $v = [5, 3, 8, 6, 2]$.

3.3 Selection Sort

O algoritmo Selection Sort, também conhecido como ordenação por seleção, é um algoritmo cujo princípio é selecionar o menor elemento encontrado em uma lista de elementos e trocá-lo com a primeira posição na lista, e repetir esta operação para as seguintes posições e elementos.

O funcionamento do algoritmo se dá da seguinte maneira: selecione o menor item do vetor e a seguir troque-o com o item que está na primeira posição do vetor. Repita estas duas operações com os n — 1 itens restantes, depois com os n— 2 itens, até que reste apenas um elemento.

A seguir está um exemplo de implementação do algoritmo Selection Sort, feito pelo autor utilizando linguagem C, e um teste de mesa, montado com o código em questão como base.

```
#include <stdio.h>
#include <stdlib.h>

void selS(int *v, int tam){

int i, j, aux, menor;

for(i=0; i<tam; i++){

    menor = i;

    for(j=i+1; j<tam; j++){

        if(v[j]< v[menor]){

            menor = j;
```

```
            }
        }
        aux = v[i];
        v[i] = v[menor];
        v[menor] = aux;

    }
}
```

Passo	i	j	*menor*	*aux*	**Estado do Vetor** (v)
1	0	1	4	-	[5, 3, 8, 6, 2]
2	0	2	4	-	[5, 3, 8, 6, 2]
3	0	3	4	-	[5, 3, 8, 6, 2]
4	0	4	4	-	[5, 3, 8, 6, 2]
5	0	-	-	5	[2, 3, 8, 6, 5]
6	1	2	1	-	[2, 3, 8, 6, 5]
7	1	3	1	-	[2, 3, 8, 6, 5]
8	1	4	1	-	[2, 3, 8, 6, 5]
9	1	-	-	3	[2, 3, 8, 6, 5]
10	2	3	3	-	[2, 3, 8, 6, 5]
11	2	4	4	-	[2, 3, 8, 6, 5]
12	2	-	-	8	[2, 3, 5, 6, 8]
13	3	4	3	-	[2, 3, 5, 6, 8]
14	3	-	-	6	[2, 3, 5, 6, 8]
15	4	-	-	-	[2, 3, 5, 6, 8]

Tabela 2: Teste de mesa para o algoritmo Selection Sort com $v = [5, 3, 8, 6, 2]$.

3.4 Shell Sort

O algoritmo Shell Sort é um algoritmo baseado em outro algoritmo, já apresentado neste texto: o de ordenação por inserção, ou Insertion Sort. O Insertion Sort compara itens em posições adjacentes, e por causa disso, caso o menor item esteja no final da lista, o número final de movimentações e comparações se torna muito grande. O método Shell Sort contorna este problema fazendo comparações e movimentações entre itens que estão distantes uns dos outros.

O funcionamento do algoritmo Shell Sort ocorre da seguinte maneira: recebe uma lista de elementos, e compara os itens de uma distância h, trocando os elementos se for necessário. Após esta primeira varredura, ele diminui o valor da distância e procede com outra varredura, seguindo assim sucessivamente até que a lista esteja completamente ordenada.

A seguir está um exemplo de implementação do algoritmo Shell Sort, feito pelo autor utilizando linguagem C, e um teste de mesa, montado com o código em questão como base.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void shS(int *v, int tam){
int i, j, aux;
float k = log(tam+1)/log(3);
k = floor(k+0.5);
int h = (pow(3,k)-1)/2;

while (h>0){
    for(i=0; i<tam-h; i++){
        if(v[i]>v[i+h]){
                aux = v[i+h];
                v[i+h] = v[i];
                v[i] = aux;
                j = i-h;
                while(j>=0){
                    if(aux<v[j]){
                        v[j+h] = v[j];
                        v[j] = aux;
```

```
                    }
                    else{
                        break;
                    }
                    j = j-h;
                }}
    } h = (h-1)/3;
}}
```

Passo	h	i	j	aux	**Estado do Vetor** (v)
1	4	0	-	-	[5, 3, 8, 6, 2]
2	4	0	-	2	[2, 3, 8, 6, 5]
3	4	1	-	-	[2, 3, 8, 6, 5]
4	4	2	-	-	[2, 3, 8, 6, 5]
5	4	3	-	-	[2, 3, 8, 6, 5]
6	1	1	0	3	[2, 3, 8, 6, 5]
7	1	2	1	8	[2, 3, 8, 6, 5]
8	1	3	2	6	[2, 3, 6, 8, 5]
9	1	4	3	5	[2, 3, 6, 5, 8]
10	1	4	2	5	[2, 3, 5, 6, 8]

Tabela 3: Teste de mesa para o algoritmo Shell Sort com $v = [5, 3, 8, 6, 2]$.

3.5 Merge Sort

O Merge Sort, ou Ordenação por Mistura, é um algoritmo de ordenação que consiste em dividir uma estrutura em subconjuntos e aplicar ordenação nos elementos que foram extraídos da estrutura original, e posteriormente, misturar estes em um conjunto final ordenado. Naturalmente, é possível notar que o algoritmo Merge Sort atua sob o paradigma divisão e conquista, que consiste em dividir um problema a ser resolvido, encontra soluções para estas novas partes, e então combinar as soluções em uma solução global.

O funcionamento do algoritmo se dá em 3 etapas, a partir do momento em que recebe uma lista de elementos.

1ª etapa: Dividir a sequência de n elementos em 2 partes, gerando assim 2 partes de tamanho n/2 cada, que devem ser ordenados.

2ª etapa: Ordenar as 2 partes de forma recursiva.

3ª etapa: Intercalar as 2 partes ordenadas, obtendo a sequência ordenada final.

A etapa de dividir ocorre repetidamente até que o menor número de elementos esteja no vetor, e a partir disso, o algoritmo passa a ordenar e combinar. A Figura 3 ilustra o funcionamento do Merge Sort, e em seguida, a Figura 4 mostra seu pseudocódigo.

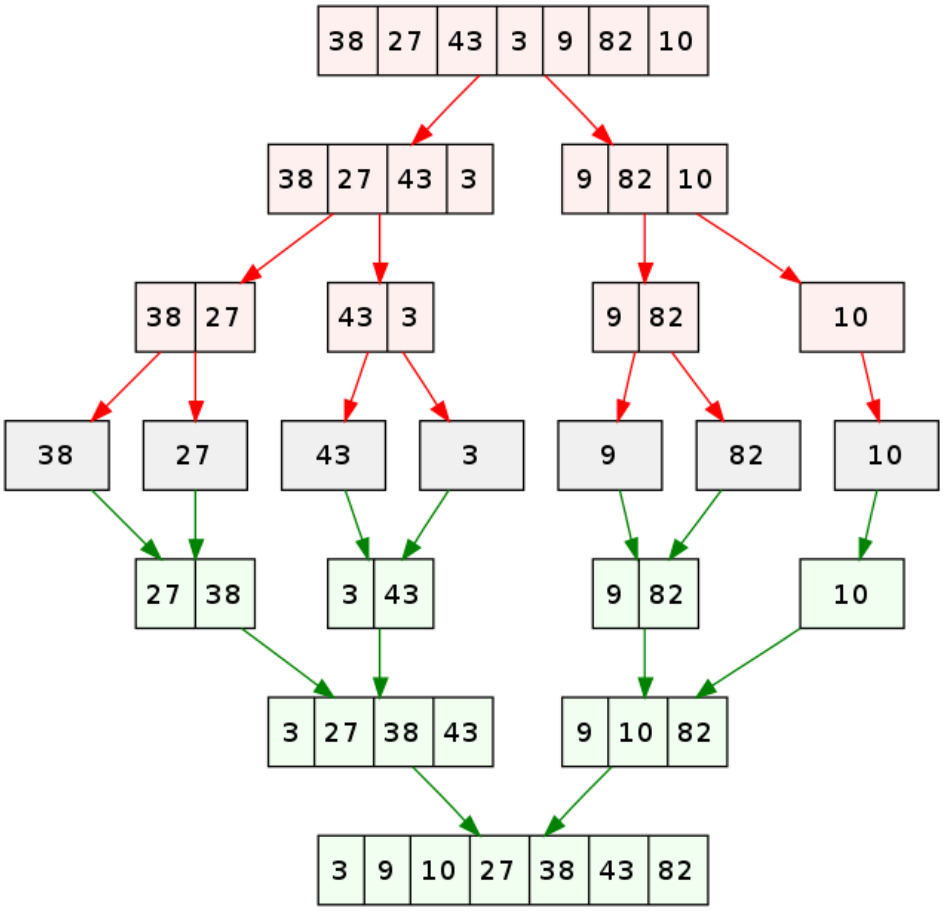

Figura 3: Funcionamento do algoritmo Merge Sort em um vetor de 7 elementos = [38, 27, 43, 3, 9, 82, 10]

```
MERGE(A, P, q, r)
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  criar arranjos L[1..n_1 + 1] e R[1..n_2 + 1]
 4  for i ← 1 to n_1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7       do R[j] ← A[q + j]
 8  L[n_1 + 1] ← ∞
 9  R[n_2 + 1] ← ∞
10   i ← 1
11   j ← 1
12   for k ← p to r
13        do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```

Figura 4: Pseudocódigo do Merge Sort

A seguir está um exemplo de implementação do algoritmo Merge Sort, feito pelo autor utilizando linguagem C, e um teste de mesa, montado com o código em questão como base.

```
void mS(int *v, int *aux, int inicio, int fim){
int meio, i, j, k;

if(inicio<fim){
    meio = ((inicio + fim)/2);
    mS(v, aux, inicio, meio);
    mS(v, aux, meio+1, fim);
    i = inicio;
    j = meio + 1;
    k = inicio;

    while(i <= meio && j<= fim){
        if(v[i] < v[j]){
            aux[k] = v[i];
            i++;
            k++;
        }
        else{
            aux[k] = v[j];
            j++;
            k++;
    }}
    while(i <= meio){
        aux[k] = v[i];
        i++;
        k++;
    }
    while(j <= fim){
        aux[k] = v[j];
```

```
        j++;
        k++;
    }
    for(i=inicio; i<=fim; i++){
        v[i] = aux[i];
    }}}
```

Etapa	Descrição	Estado do Vetor (v)	Observação
1	Início do algoritmo com $v = [38, 27, 43, 3, 9, 82, 10]$	[38, 27, 43, 3, 9, 82, 10]	Vetor original
2	Divisão: $inicio = 0, meio = 3, fim = 6$	[38, 27, 43, 3, 9, 82, 10]	Divisão recursiva
3	Subvetor esquerdo: $inicio = 0, fim = 3$	[38, 27, 43, 3]	
4	Subdivisão: $inicio = 0, meio = 1, fim = 3$	[38, 27, 43, 3]	
5	Subvetor esquerdo: $inicio = 0, fim = 1$	[38, 27]	
6	Subvetor direito: $inicio = 2, fim = 3$	[43, 3]	
7	Mesclagem: Ordena $[38, 27]$ para $[27, 38]$	[27, 38]	Mesclagem
8	Mesclagem: Ordena $[43, 3]$ para $[3, 43]$	[3, 43]	Mesclagem
9	Mesclagem: Combina $[27, 38]$ e $[3, 43]$ para $[3, 27, 38, 43]$	[3, 27, 38, 43]	Mesclagem final
10	Subvetor direito: $inicio = 4, fim = 6$	[9, 82, 10]	
11	Subdivisão: $inicio = 4, meio = 5, fim = 6$	[9, 82, 10]	
12	Subvetor esquerdo: $inicio = 4, fim = 5$	[9, 82]	
13	Subvetor direito: $inicio = 6, fim = 6$	[10]	
14	Mesclagem: Ordena $[9, 82]$ para $[9, 82]$	[9, 82]	
15	Mesclagem: Combina $[9, 82]$ e $[10]$ para $[9, 10, 82]$	[9, 10, 82]	
16	Mesclagem final: Combina $[3, 27, 38, 43]$ e $[9, 10, 82]$	[3, 9, 10, 27, 38, 43, 82]	Vetor ordenado

Tabela 4: Teste de mesa do Merge Sort com o vetor $[38, 27, 43, 3, 9, 82, 10]$.

3.6 Quick Sort

O Quick Sort, ou Ordenação Rápida, é um algoritmo de ordenação que, assim como o Merge Sort, utiliza o paradigma de divisão e conquista, dividindo uma estrutura em subconjuntos, ordenando os elementos destes e juntando-os em um conjunto final, agora ordenado. O Quick Sort funciona da seguinte maneira: recebe uma lista de elementos, divide esta em 2 subconjuntos, estabelece um valor dentre os elementos como um pivô, e passa a realizar comparações e trocas de posição dentre os valores dos elementos, onde todos os valores menores que o pivô devem estar à sua esquerda, e os maiores, à direita. Através de chamadas recursivas, subconjuntos menores são formados e ordenados, até que todo o conjunto esteja devidamente ordenado. O algoritmo Quick Sort não possui uma fase de combinação distinta, como é o caso do algoritmo Merge Sort. Esse algoritmo opera usando uma estratégia de

divisão e conquista em que as etapas de divisão e combinação são integradas em um único processo.

O Quick Sort age sob o paradigma de divisão e conquista pelas seguintes etapas:

Dividir: Ocorre através do procedimento em geral denominado Particiona, que é o procedimento de dividir o conjunto de elementos, resultando em 2 subconjuntos.

Conquistar: Os subconjuntos são ordenados recursivamente.

Combinar: Não ocorre, pois ao fim da execução, cada elemento já será colocado na sua posição correta.

O pseudocódigo do Particiona e do Quick Sort estão representados abaixo:

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  trocar A[i] ← A[j]
7  trocar A[i + 1] ↔ A[r]
8  return i + 1
```

Figura 5: Pseudocódigo do Particiona

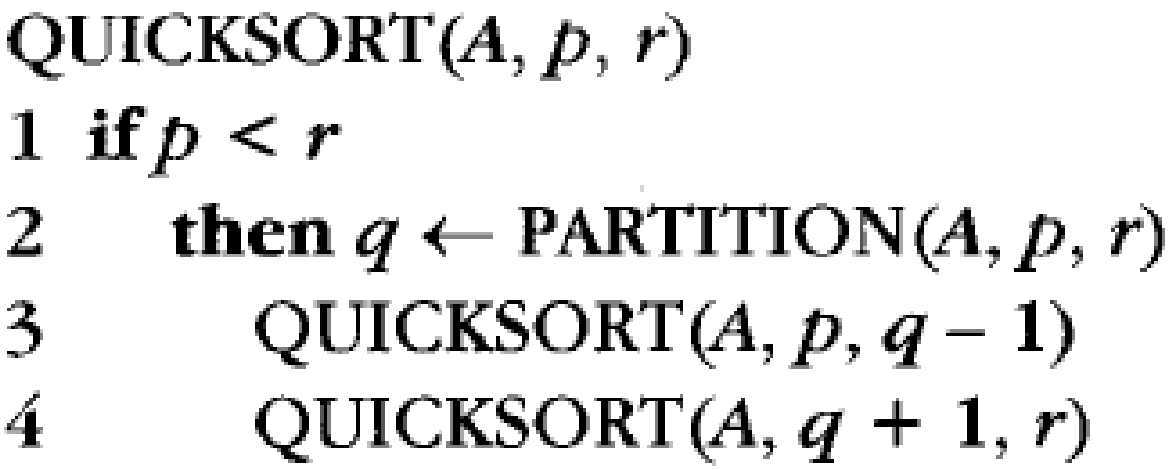

Figura 6: Pseudocódigo do Quick Sort

A seguir está um exemplo de implementação da função Particiona e do algoritmo Quick Sort, feito pelo autor utilizando linguagem C. O pivô escolhido para a realização do procedimento no exemplo é o da posição inicial, e a função particionaETrocaPI faz o particionamento e realiza a troca entre elementos.

```
int particionaETroca_PI(int *v, int inicio, int fim) {
    int pivo = v[inicio];
    int i = inicio + 1;
    int j = fim;

    while (i <= j) {
        while (i <= j && v[i] <= pivo) {
            i = i + 1;
        }
        while (i <= j && v[j] > pivo) {
            j = j - 1;
        }
        if (i < j) {
            int aux = v[i];
```

```
            v[i] = v[j];
            v[j] = aux;
        }
    }
    v[inicio] = v[j];
    v[j] = pivo;
    return j;
}



void qS_PI(int *v, int inicio, int fim) {
    while (inicio < fim) {
        int pos = particionaETroca_PI(v, inicio, fim);
        if (pos - inicio < fim - pos) {
            qS_PI(v, inicio, pos - 1);
            inicio = pos + 1;
        } else {
            qS_PI(v, pos + 1, fim);
            fim = pos - 1;
        }}}
```

3.7 Heap Sort

O Heap Sort é um algoritmo de ordenação que também atua sob o paradigma de divisão e conquista, que mescla características de dois algoritmos de ordenação já vistos anteriormente: o Insertion Sort e o Merge Sort. Assim como o Insertion Sort, sua ordenação é local, ou seja, ajusta os valores diretamente no array que foi dado, sem ter que criar listas auxiliares externas para armazenar dados intermediários, e assim como o Merge Sort, seu tempo de execução é $O(n.log(n))$. Este algoritmo faz uso da estrutura de dados Heap (binário), que pode ser visto como uma árvore binária praticamente completa, em que cada nó da árvore corresponde a

um elemento do arranjo que armazena o valor do nó, como ilustrado na figura a seguir.

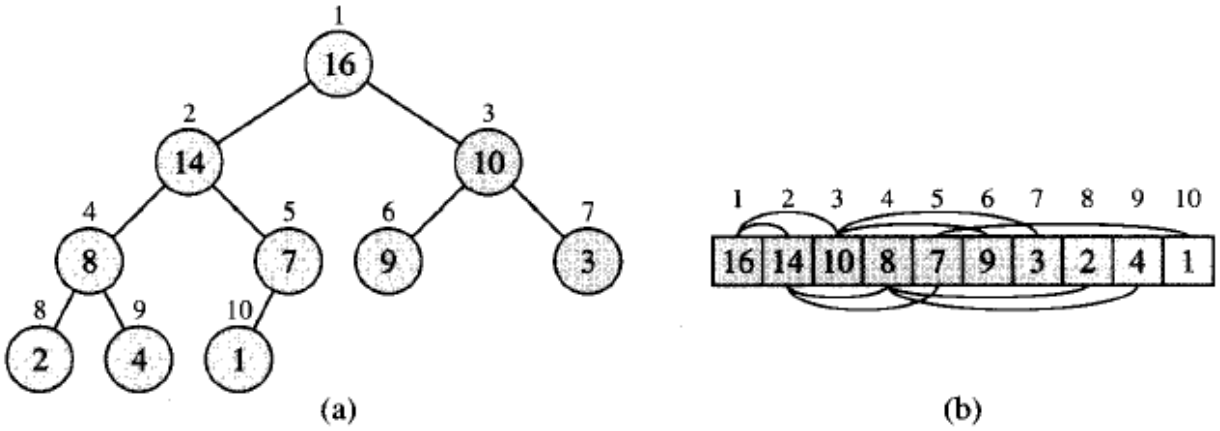

Figura 7: Representação de um Heap, onde (a) é a árvore binária, e (b), seu respectivo arranjo

Um heap pode ser máximo ou mínimo. Um heap é dito máximo quando o maior elemento do arranjo está na raiz, e os elementos subsequentes são menores que este, e é dito mínimo quando o menor elemento do arranjo está na raiz, e os elementos subsequentes são maiores que este. Ao observar o heap da Figura 7, pode-se afirmar que se trata de um heap máximo. Neste relatório, haverá maior enfoque no heap mínimo.

O algoritmo Heap Sort é composto por 3 partes principais, ligadas entre si: Heapify (ou Verifica), Build (ou Constroi) e Heapsort, de modo que há variações se o heap em questão for máximo ou mínimo. O Heapify é uma sub-rotina cujo propósito é verificar se a propriedade de máximo ou mínimo está sendo devidamente seguida, verificando se o item em questão deveria estar mais abaixo no arranjo, e fazer com que ele "desça"até a posição correta. Segue o pseudocódigo Min-Heapify, que é o procedimento Heapify aplicado a um heap mínimo.

```
1  function MIN-HEAPIFY(A, i)
2    L <- LEFT(i)
3    R <- RIGHT(i)
4    if L <= tamanho-do-heap[A] and A[L] < A[i]
5      then menor <- L
6      else menor <- i
7    if R <= tamanho-do-heap[A] and A[R] < A[menor]
8      then menor <- R
9    if menor is different from i
10     then trocar A[i] <-> A[menor]
11          MIN-HEAPIFY(A, menor)
```

Figura 8: Pseudocódigo de um Heapify para um heap mínimo

O Build é uma sub-rotina que constrói um Heap, ou seja, a partir de um arranjo de elementos dado anteriormente, o procedimento Heapify é aplicado várias vezes sobre este até que o arranjo de elementos esteja em forma de Heap. A seguir, está um pseudocódigo Build-Min-Heap, que é o procedimento Build aplicado a um heap mínimo.

```
1  function BUILD-MIN-HEAP(A)
2    tamanho-do-heap[A] <- comprimento[A]
3    for i <- comprimento[A]/2 down to 1
4      do MIN-HEAPIFY(A, i)
```

Figura 9: Pseudocódigo de um Build para um heap mínimo

O algoritmo Heapsort, em si, começa utilizando o Build para construir um heap, e então, troca o elemento da raiz com o último elemento do heap, fixando-o nesta posição, e passa então a utilizar o Heapify a partir do primeiro item, mas não modifica o último item do arranjo. É importante notar que a ordenação será diferente, dependendo do tipo de heap. Segue o pseudocódigo de um Heapsort sobre um heap mínimo.

```
1 function HEAPSORT(A)
2   BUILD–MIN–HEAP(A)
3   for i <- comprimento[A] down to 2
4     do trocar A[1] <-> A[i]
5        tamanho–do–heap[A] <- tamanho–do–heap[A] - 1
6        MIN–HEAPIFY(A, 1)
```

Figura 10: Pseudocódigo de um Heapsort para um heap mínimo

4 ANÁLISE DE COMPLEXIDADE

4.1 Insertion Sort

A Tabela 5 a seguir foi montada a partir do pseudocódigo descrito anteriormente, na seção 3.1. A partir dela, começaremos a análise de complexidade do Insertion Sort e definiremos seu melhor, pior e médio casos.

Linha	**Instruções**	**Custo**	**Vezes**
1	For j ← 2 to comprimento	$C1$	n
2	do chave ← A[j]	$C2$	n - 1
3	//inserir A[j] na sequência ordenada A(1,...,j+1)	$C3 = 0$	n - 1
4	i ← j - 1	$C4$	n - 1
5	while i > 0 e A[i] > Chave	$C5$	$\sum_{j=2}^{n} T_j$
6	do A[i+1] ← A[i]	$C6$	$\sum_{j=2}^{n}(T_j - 1)$
7	i ← i - 1	$C7$	$\sum_{j=2}^{n}(T_j - 1)$
8	A[i + 1] ← Chave	$C8$	n - 1

Tabela 5: Tabela de cálculo da complexidade do Insertion Sort

Através de um cálculo de Custo X Vezes, obtemos a seguinte fórmula geral para o cálculo de complexidade do algoritmo Insertion Sort:

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5\left[\sum_{j=2}^{n} T_j\right] + C_6\left[\sum_{j=2}^{n}(T_j - 1)\right] + C_7\left[\sum_{j=2}^{n}(T_j - 1)\right]$$

$$+ C_8(n-1)$$

4.1.1 Melhor caso

No melhor caso do Insertion Sort, o conjunto já está ordenado. Neste caso, na linha 5 é falsa a condição A[i] > Chave, e as linhas 6 e 7 não são executadas.

$$T(n) = [C1n] + [C2(n-1)] + [C4(n-1)] + [C5(n-1)] + [C8(n-1)]$$

$$T(n) = [(C1 + C2 + C4 + C5 + C8)n] + [-C2 - C4 - C5 - C8]$$

$$T(n) = an + b$$

4.1.2 Pior caso

O pior caso ocorre quando o conjunto estiver ordenado em ordem inversa. Nessa situação, deve-se comparar cada elemento A[j] com cada elemento do subconjunto ordenado inteiro A[i, ...,j-1], e então Tj = j para 2,3,4, ...,n, observando que:

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5\left[\sum_{j=2}^{n} T_j\right] + C_6\left[\sum_{j=2}^{n}(T_j - 1)\right] + C_7\left[\sum_{j=2}^{n}(T_j - 1)\right]$$

$$+C8(n-1)$$

$$T(n) = C1n + C2(n-1) + C3(n-1) + C4(\frac{1}{2}n^2 + n - 1) + C5(\frac{1}{2}n^2 - n) + C6(\frac{1}{2}n^2 - n) + C7n - C7$$

$$T(n) = (\frac{1}{2}C4 + C5 + C6)n^2 + (C1 + C2 + C3 + \frac{1}{2}C4 - C5 - C6 + C7)n + (-C2 - C3 - C4 - C7)$$

$$T(n) = a(n^2) + bn + c$$

4.1.3 Médio Caso:

O médio caso do algoritmo Insertion Sort ocorre quando o conjunto de elementos está semi-ordenado, ou seja, não está completamente ordenado, mas também não está completamente desordenado. Desta forma, as operações serão feitas com j assumindo o valor 1/2, pois essa situação não vai ocorrer no pior caso e nem no melhor caso.

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5 \frac{1}{2}\left[\sum_{j=2}^{n} T_j\right] + C_6 \frac{1}{2}\left[\sum_{j=2}^{n} (T_j - 1)\right] + C_7 \frac{1}{2}\left[\sum_{j=2}^{n} (T_j - 1)\right]$$

$$+ C8(n-1)$$

$$T(n) = C1n + C2(n-1) + C3(n-1) + C4(\frac{1}{4}n^2 + n - 1) + C5(\frac{1}{4}n^2 - n) + C6(\frac{1}{4}n^2 - n) + C7n - C7$$

$$T(n) = (\frac{1}{4}C4 + C5 + C6)n^2 + (C1 + C2 + C3 + \frac{1}{4}C4 - C5 - C6 + C7)n + (-C2 - C3 - C4 - C7)$$

$$T(n) = a(n^2) + bn + c$$

4.2 Bubble Sort

A Tabela 6 a seguir foi montada a partir do pseudocódigo descrito anteriormente, na seção 3.2. A partir dela, começaremos a análise de complexidade do Insertion Sort e definiremos seu melhor, pior e médio casos.

Linha	Instruções	Custo	Vezes
1	Para i = 1 até comprimento[A] - 1	$C1$	$n-1$
2	Para j = 1 até comprimento[A] - i	$C2$	$\sum_{j=1}^{n-1} T_j$
3	Se A[j] > A[j + 1] então	$C3$	$\sum_{j=1}^{n-1}(T_j - 1)$
4	Troca A[j] com A[j+1]	$C4$	$\sum_{j=1}^{n-1}(T_j - 1)$

Tabela 6: Tabela de cálculo de complexidade do Bubble Sort.

Através de um cálculo de Custo X Vezes, obtemos a seguinte fórmula geral para o cálculo de complexidade do algoritmo Bubble Sort:

$$T(n) = C_1(n-1) + C_2\left[\sum_{j=1}^{n-1} T_j\right] + C_3\left[\sum_{j=1}^{n-1}(T_j - 1)\right] + C_4\left[\sum_{j=1}^{n-1}(T_j - 1)\right]$$

4.2.1 Melhor caso

No melhor caso do Bubble Sort, o conjunto de dados já está ordenado, então a condição A[j] > A[j + 1] na linha 3 é falso, logo, a linha 4 não é executada.

$$T(n) = C_1(n-1) + C_2\left[\sum_{j=1}^{n-1} T_j\right] + C_3\left[\sum_{j=1}^{n-1}(T_j - 1)\right]$$

$$T(n) = C_1(n-1) + C_2\left(\frac{1}{2}n^2 - n\right) + C_3\left(\frac{1}{2}n^2 - 3n + 2\right)$$

$$T(n) = \left(\frac{1}{2}C_2 + C_3\right)n^2 + \left(C_1 + \frac{1}{2} - C_2 - C_3\right)n + \left(-C_1 + \frac{1}{2} + 2C_3\right)$$

$$T(n) = a(n^2) + bn + c$$

4.2.2 Pior caso

Se o conjunto estiver ordenado em ordem inversa, resulta o pior caso. Com isso, deve-se comparar cada elemento A[j] com cada elemento do subconjunto ordenado.

$$T(n) = C_1(n-1) + C_2\left[\sum_{j=1}^{n-1} T_j\right] + C_3\left[\sum_{j=1}^{n-1}(T_j - 1)\right] + C_4\left[\sum_{j=1}^{n-1}(T_j - 1)\right]$$

$$T(n) = C_1(n-1) + C_2\left(\frac{1}{2}n^2 - n\right) + C_3\left(\frac{1}{2}n^2 - 3n + 2\right) + C_4\left(\frac{1}{2}n^2 - 3n + 2\right)$$

$$T(n) = \left(\frac{1}{2}C_2 + C_3 + C_4\right)n^2 + \left(C_1 + \frac{1}{2} - C_2 - 3C_3 - 3C_4\right)n + \left(-C_1 + \frac{1}{2} + 2C_3 + 2C_4\right)$$

$$T(n) = a(n^2) + bn + c$$

4.2.3 Médio caso

O médio caso do algoritmo Bubble Sort ocorre quando o conjunto de elementos está semi-ordenado, ou seja, não está completamente ordenado, mas também não está completamente desordenado. Desta forma, as operações serão feitas com j assumindo o valor $1/2$, pois essa situação não vai ocorrer no pior caso e nem no melhor caso.

$$T(n) = C_1(n-1) + C_2\frac{1}{2}\left[\sum_{j=1}^{n-1} T_j\right] + C_3\frac{1}{2}\left[\sum_{j=1}^{n-1}(T_j - 1)\right] + C_4\frac{1}{2}\left[\sum_{j=1}^{n-1}(T_j - 1)\right]$$

$$T(n) = C_1(n-1) + C_2\left(\frac{1}{4}n^2 - n\right) + C_3\left(\frac{1}{4}n^2 - 3n + 2\right) + C_4\left(\frac{1}{4}n^2 - 3n + 2\right)$$

$$T(n) = \left(\frac{1}{4}C_2 + C_3 + C_4\right)n^2 + \left(C_1 + \frac{1}{4} - C_2 - 3C_3 - 3C_4\right)n + \left(-C_1 + \frac{1}{4} + 2C_3 + 2C_4\right)$$

$$T(n) = a(n^2) + bn + c$$

4.3 Selection Sort

A Tabela 7 a seguir foi montada a partir do pseudocódigo descrito anteriormente, na seção 3.3. A partir dela, começaremos a análise de complexidade do Selection Sort e definiremos seu melhor, pior e médio casos.

Linha	**Instruções**	**Custo**	**Vezes**
1	Para $i \leftarrow 1$ até $n-1$	C_1	n
2	menor $\leftarrow i$	C_2	$n-1$
3	Para $j \leftarrow i+1$ até n	C_3	$\sum_{j=2}^{n} T_j$
4	Se $A[j] < A[\text{menor}]$	C_4	$\sum_{j=2}^{n}(T_j - 1)$
5	menor $\leftarrow j$	C_5	$\sum_{j=2}^{n}(T_j - 1)$
6	Troca$(A[i], A[\text{menor}])$	C_6	$n-1$

Tabela 7: Tabela de cálculo de complexidade do Selection Sort.

Através de um cálculo de Custo X Vezes, obtemos a seguinte fórmula geral para o cálculo de complexidade do algoritmo Selection Sort:

$$T(n) = C_1 n + C_2(n-1) + C_3\left[\sum_{j=1}^{n-1} T_j\right] + C_4\left[\sum_{j=1}^{n-1}(T_j - 1)\right] + C_5\left[\sum_{j=1}^{n-1}(T_j - 1)\right] + C_6(n-1)$$

4.3.1 Melhor caso

No melhor caso do Selection Sort, o conjunto já está ordenado, porém, o Selection Sort sempre vai executar todas as linhas, a fim de descobrir o menor elemento.

$$T(n) = C_1 n + C_2(n-1) + C_3 \left[\sum_{j=1}^{n-1} T_j\right] + C_4 \left[\sum_{j=1}^{n-1} (T_j - 1)\right] + C_5 \left[\sum_{j=1}^{n-1} (T_j - 1)\right] + C_6(n-1)$$

$$T(n) = C_1 n + C_2(n-1) + C_3 \frac{n(n+1)}{2} - 1 + C_4 \frac{n(n-1)}{2} + C_5 \frac{n(n-1)}{2} + C_6(n-1)$$

$$T(n) = \left(\frac{C_3}{2} + \frac{C_4}{2} + \frac{C_5}{2}\right) n^2 + \left(C_1 + C_2 + \frac{C_3}{2} + \frac{C_4}{2} - \frac{C_5}{2} + C_6\right) n - \left(C_2 + \frac{C_3}{2} + C_6\right)$$

$$T(n) = a(n^2) + bn + c$$

4.3.2 Pior caso

O pior caso ocorrerá quando o conjunto de elementos estiver ordenado na ordem inversa.

$$T(n) = C_1 n + C_2(n-1) + C_3 \frac{n(n+1)}{2} - 1 + C_3(n-1) + C_4 \frac{n(n-1)}{2} + C_5 \frac{n(n-1)}{2} + C_6(n-1)$$

$$T(n) = \left(\frac{C_3}{2} + \frac{C_4}{2} + \frac{C_5}{2}\right) n^2 + \left(C_1 + C_2 + C_3 + \frac{C_4}{2} - \frac{C_5}{2}\right) n - (C_2 + C_3 + C_6)$$

$$T(n) = a(n^2) + bn + c$$

4.3.3 Médio caso

No médio caso, as operações serão realizadas com $j = 1/2$, pois essa situação não ocorrerá no melhor caso ou no pior caso.

$$T(n) = C_1 n + C_2(n-1) + C_3\frac{n(n+1)}{2} - 1 + C_4\frac{\frac{1}{2}n(n-1)}{2} + C_5\frac{\frac{1}{2}n(n-1)}{2} + C_6(n-1)$$

$$T(n) = C_3 n^2 + \left(C_1 + C_2 + C_3 + \frac{C_4}{2} + C_5\right)n - \left(C_2 + \frac{C_3}{2} + C_4 + C_5 + C_6\right)$$

$$T(n) = a(n^2) + bn + c$$

4.4 Shell Sort

Até o momento em que este relatório foi escrito, ainda não foi possível calcular a complexidade do algoritmo Shell Sort, devido ao fato de que possui uma matemática muito complexa, e prová-la se mostrou extremamente desafiador.

4.5 Merge Sort

A Tabela 8 a seguir foi montada a partir do pseudocódigo descrito anteriormente, na seção 3.4. A partir dela, começaremos a análise de complexidade do Merge Sort e definiremos seu melhor, pior e médio casos.

Linha	**Instruções**	**Custo**	**Vezes**
1	if P<R	C_1	O(1)
2	then Q $\leftarrow$ (P+R)/2	C_2	O(1)
3	Merge sort (A,P,Q)	C_3	O(n/2)
4	Merge sort (A,Q+1,R)	C_4	O(n/2)
5	Merge sort (A,P,Q,R)	C_5	O(n)

Tabela 8: Tabela de cálculo de complexidade do Merge Sort.

A fórmula para calcular a complexidade do algoritmo Merge Sort e definir seu melhor, pior e médio casos, é baseada na recorrência que descreve o funcionamento do algoritmo:

$$T(n) = 2T(n/2) + n$$

Não há diferenças no cálculo para melhor, pior ou médio caso neste algoritmo. Todos são definidos a partir da seguinte sequência de operações:

$$T(n) = 2T(n/2) + 2$$

$$n^{\log_2(2)} = n$$

$$T(n) = O(n^{\log_2(2)})$$

$$= O(n)$$

Logo,

$$T(n) = O(n.log(n))$$

4.6 Quick Sort

As Tabelas 9 e 10 a seguir foram montadas a partir dos pseudocódigos descritos anteriormente, na seção 3.5. A partir delas, começaremos a análise de complexidade da função Particiona e do Quick Sort, e definiremos seu melhor, pior e médio casos.

Linha	Instruções	Custo	Vezes
1	P ← Esq	C_1	1
2	R ← Dir	C_2	1
3	V[P] ← Pivo	C_3	1
4	while(Esq < Dir)	C_4	n
5	while(Esq <= Final e V[Esq] <= Pivo	C_5	n-1
6	Esq++	C_6	n-2
7	while(Dir=>0 e V[Dir] > Pivo	C_7	n-1
8	Dir–	C_8	n-2
9	if(Esq<Dir)	C_9	n-1
10	then Troca V[Esq] e V[Dir	C_{10}	n-1
11	Troca V[P] e V[Dir]	C_{11}	1
12	Return Dir	$C_{12} = 0$	1

Tabela 9: Tabela de cálculo de complexidade do Particiona, está dividida em 2 partes

Linha	**Instruções**	**Custo**	**Vezes**
1	if R<P	C_1	O(1)
2	then Q = Particiona (V,P,R)	C_2	O(n)
3	Quick sort (V,P,Q-1)	C_3	O(n/2)
4	Quick sort (V,Q+1,R)	C_4	O(n/2)

Tabela 10: Tabela de cálculo de complexidade do Quick Sort.

A partir da exposição do Particiona e do Quick Sort, será feita a análise de complexidade do Particiona, e a definição do melhor, pior e médio casos do Quick sort.

4.6.1 Complexidade do Particiona

A complexidade do Particiona se dá pelas seguintes operações:

$$T(n) = C_1(1)+C_2(1)+C_3(1)+C_4(n)+C_5(n-1)+C_6(n-2)+C_7(n-1)+C_8(n-2)+C_9(n-1)+C_{10}(n-1)$$

$$+C_{11}(1)$$

$$T(n) = n(C_4 + C_5 + C_6 + C_7 + C_8 + C_9 + C_{10})$$

$$+(C_1 + C_2 + C_3 - C_5 - 2C_6 - C_7 - 2C_8 - C_9 - C_{10} + C_{11})$$

$$T(n) = an + b$$

4.6.2 Melhor caso

O melhor caso do Quick sort ocorre quando as partições têm sempre o mesmo tamanho, ou seja, quando são partições balanceadas.

$$T(n) = 2T(n/2) + 2$$

$$n^{\log_2(2)} = n$$

$$T(n) = O(n^{\log_2(2)})$$

$$= O(n)$$

Logo,

$$T(n) = O(n.log(n))$$

4.6.3 Pior caso

Uma característica do Quick sort é sua ineficiência para arquivos já ordenados, quando a escolha do pivô é inadequada. A escolha sistemática dos extremos de um arquivo já ordenado leva ao seu pior caso. Nesse caso, as partições serão extremamente desiguais, e o método Quick será chamado recursivamente n vezes, eliminando apenas um item em cada chamada.

Quando ocorre, o pior caso pode ser definido como uma relação de recorrência sob a equação:

$$T(n) = T(n-1) + n$$

Ao resolver a recorrência, descobrimos a fórmula fechada:

$$f(n) = \frac{n^2 + n}{2}$$

E, ao fazermos a análise assintótica de f(n), descobrimos que a complexidade do Quick sort no pior caso é:

$$O(n^2)$$

4.6.4 Médio caso

O médio caso do Quick sort ocorre quando as partições não estão completamente balanceadas como no melhor caso, porém também não estão completamente desbalanceadas como em seu pior caso. Em média, o tempo de execução do Quick sort é:

$$T(n) = O(n.log(n))$$

4.7 Heap Sort

Para realizarmos a análise de complexidade do Heap Sort e dos procedimentos que o compõem, as tabelas 8, 9 e 10 serão referenciadas.

4.7.1 Heapify

Seja T(n) o tempo de execução para um heap de tamanho n, a etapa de dividir com a determinação do maior entre o nó pai e os nós filhos, assim $D(n) = \theta(1)$, pois 2 comparações são feitas. A etapa de combinação, que ocorre em outros algoritmos de divisão e conquista, não ocorre, porque a cada passo o elemento selecionado é colocado em sua posição correta. O pior caso ocorre quando o último nível do heap tem a metade completa dos nós, e temos aproximadamente 2/3 dos nós. Nesta situação, ocorre o maior sub-problema, e o tempo do Heapify pode ser descrito pela recorrência:

$$T(n) <= T(2n/3) + \theta(1), n > 1$$

Utilizando o método do Teorema Mestre, de acordo com o caso 2, a solução para a recorrência é:

$$T(n) = \theta(log(n))$$

4.7.2 Build

Devido à linha 2 na tabela 9, há $\theta(n)$ chamadas ao procedimento da linha 3, que tem tempo proporcional a $\theta(log(n))$, então o tempo de execução do Build é $\theta(n.log(n))$.

4.7.3 Heapsort

Devido à linha 2 na tabela 10, há $\theta(n)$ chamadas ao procedimento da linha 5, que tem tempo de execução proporcional a $\theta(log(n))$. Portanto, o tempo de execução do Heapsort é $\theta(n.log(n))$.

5 TABELA E GRÁFICO

5.1 Insertion Sort

A partir da implementação do código apresentado na seção 3.4 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Insertion Sort, como demonstrado na Tabela 11 e na Figura 11 a seguir.

————————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0	0	0	0,003
Decrescente	0	0	0,001	0,116	11,197	1145,033
Aleatório	0	0	0	0,061	5,663	562,132

Tabela 11: Tabela que demonstra o tempo de execução do algoritmo Insertion Sort em vetores com diferentes quantidades de dados e em diferentes ordens

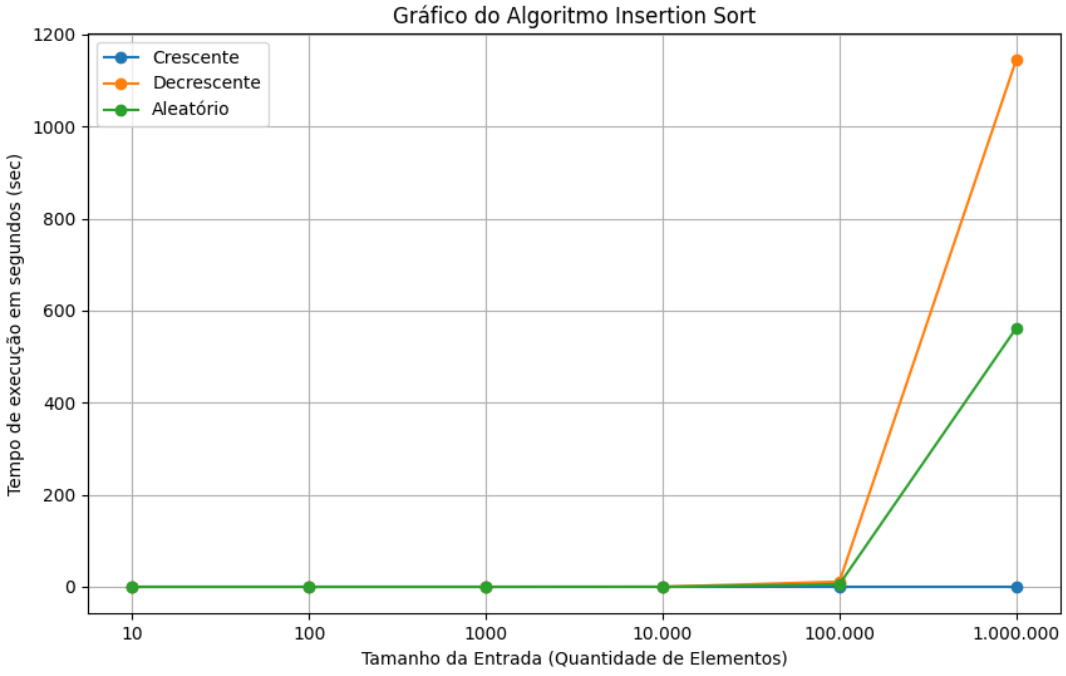

Figura 11: Gráfico ilustrando o comportamento do algoritmo Insertion Sort em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

Como está devidamente ilustrado na tabela e no gráfico apresentados anteriormente, o algoritmo Insertion Sort é útil para operações de ordenação, mas apenas demonstra um desempenho excepcional para situações em que o conjunto de elementos já está em ordem. Sendo assim, é um algoritmo muito útil para quando alguns poucos elementos são acrescentados a um conjunto que já tinha sido ordenado previamente, o que manteria seu tempo de execução bem baixo.

5.2 Bubble Sort

A partir da implementação do código apresentado na seção 3.2 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Bubble Sort, como demonstrado na Tabela 12 e na Figura 12 a seguir.

————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0,002	0,194	19,638	1959,074
Decrescente	0	0	0,003	0,269	28,153	2605,384
Aleatório	0	0	0,004	0,308	34,954	3575,224

Tabela 12: Tabela que demonstra o tempo de execução do algoritmo Bubble Sort em vetores com diferentes quantidades de dados e em diferentes ordens

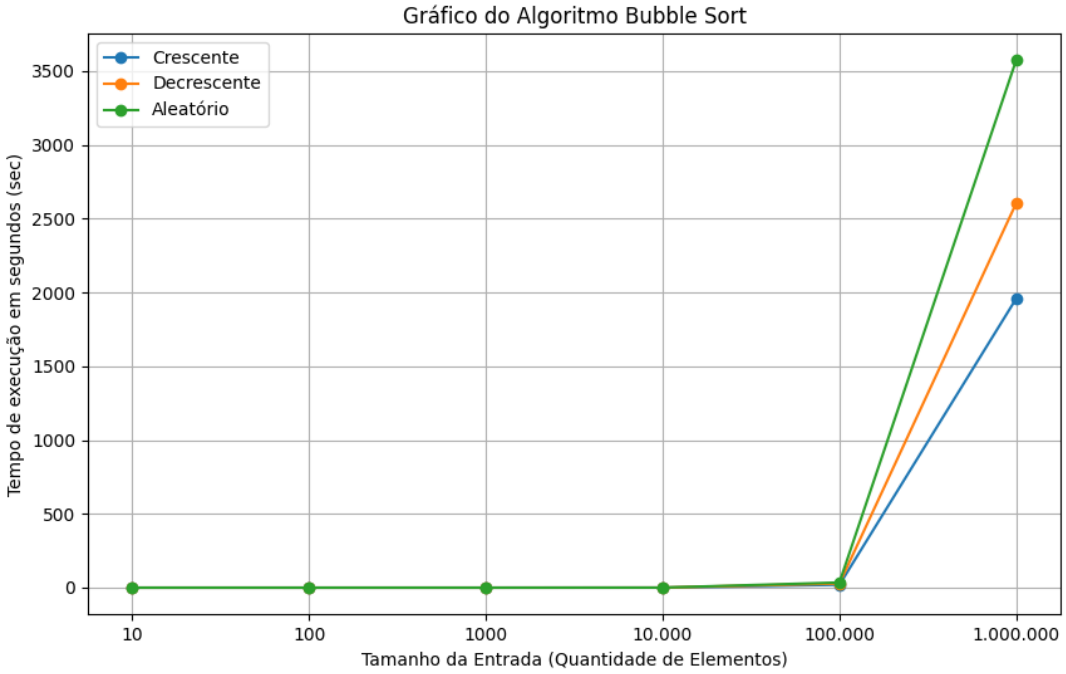

Figura 12: Gráfico ilustrando o comportamento do algoritmo Bubble Sort, em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

Um dos mais simples de implementar, o algoritmo Bubble Sort possui um ótimo desempenho para a ordenar elementos em listas pequenas e exige pouco espaço de armazenamento, porém, não é um algoritmo muito utilizado por causa de suas desvantagens.

Como principal desvantagem, pode ser citada a complexidade do algoritmo, que é $O(n^2)$ independentemente do caso, isso significa que esse tipo de ordenação exige n^2 passos de processamento para cada número n de elementos que serão ordenados, isso significa que não é um bom algoritmo para um número grande de dados, o que está devidamente ilustrado no gráfico da Figura 12. Por causa disso, o Bubble Sort é pouco usado em aplicações reais se comparado a outros algoritmos de ordenação.

5.3 Selection Sort

A partir da implementação do código apresentado na seção 3.3 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Selection Sort, como demonstrado na Tabela 13 e na Figura 13 a seguir.

————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0,001	0,103	10,272	1021,027
Decrescente	0	0	0,001	0,097	9,678	1001,792
Aleatório	0	0	0,002	0,109	10,148	1021,331

Tabela 13: Tabela que demonstra o tempo de execução do algoritmo Selection Sort em vetores com diferentes quantidades de dados e em diferentes ordens

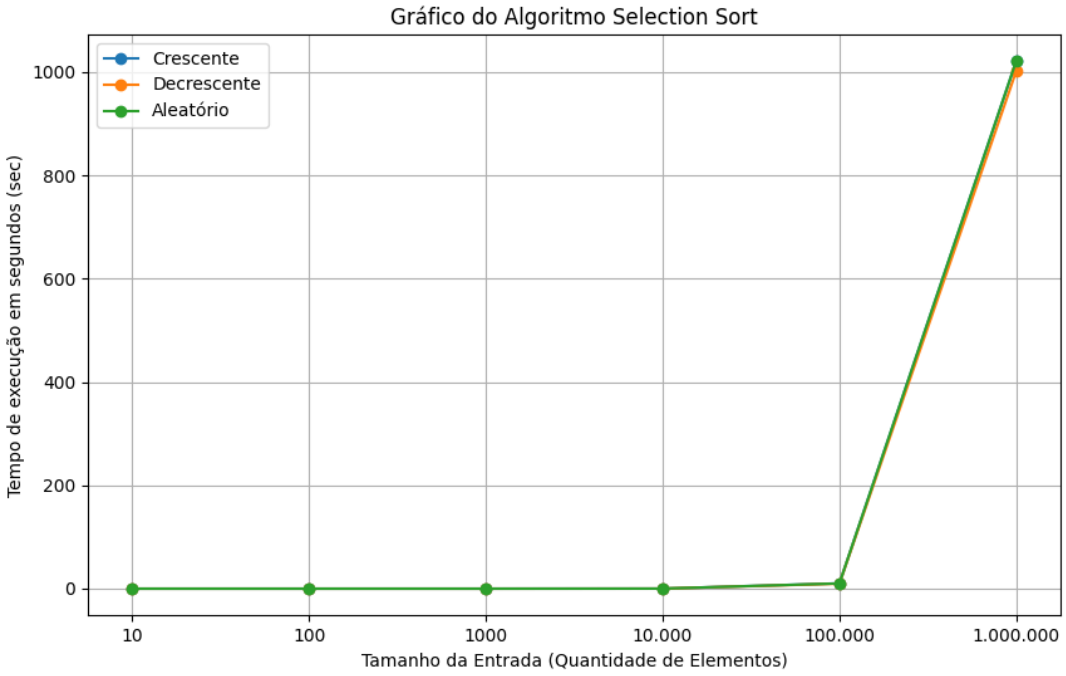

Figura 13: Gráfico ilustrando o comportamento do algoritmo Selection Sort em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

Dentre os algoritmos de ordenação, o algoritmo Selection Sort é um dos mais simples que existem, juntamente com o Bubble Sort. Tem como custo linear referente ao tamanho da

entrada para o número de movimentos de registros, ou seja, possui um número pequeno de movimentações entre os elementos, apresentando uma grande vantagem.

O Selection Sort possui como desvantagens o fato de que percorre a lista fazendo comparações e trocas independentemente se a lista já está ordenada. Além disto, o algoritmo não é estável, fazendo trocas entre elementos de valor semelhante.

5.4 Shell Sort

A partir da implementação do código apresentado na seção 3.4 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Shell Sort, como demonstrado na Tabela 14 e na Figura 14 a seguir.

————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0	0	0,002	0,026
Decrescente	0	0	0	0	0,005	0,045
Aleatório	0	0	0	0,001	0,021	0,253

Tabela 14: Tabela que demonstra o tempo de execução do algoritmo Shell Sort em vetores com diferentes quantidades de dados e em diferentes ordens

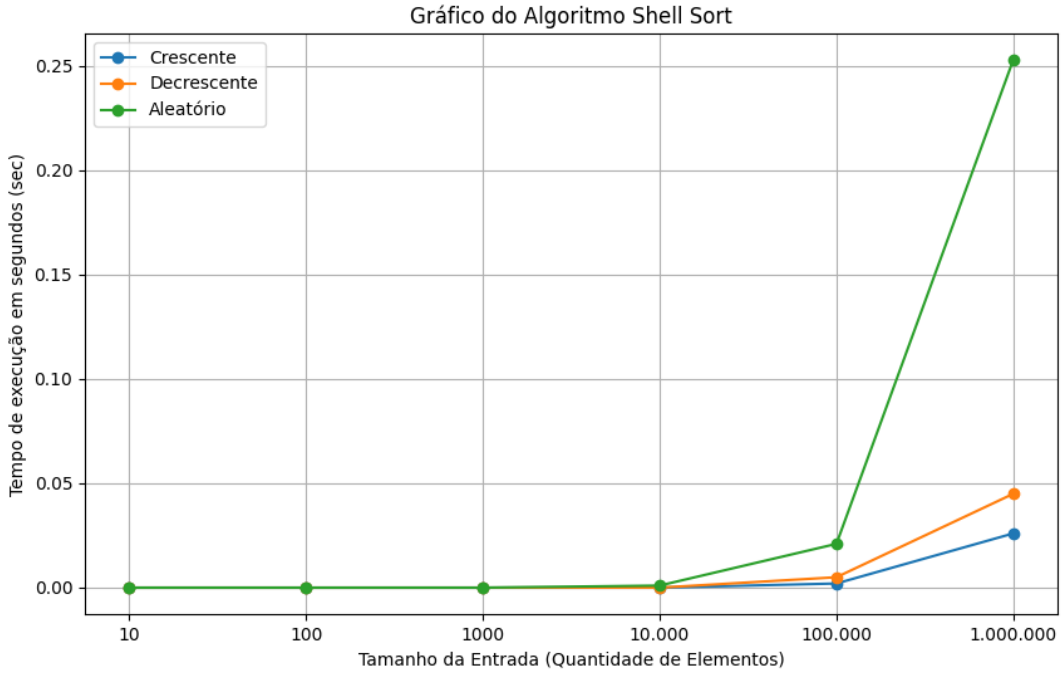

Figura 14: Gráfico ilustrando o comportamento do algoritmo Shell Sort em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

O algoritmo Shell Sort é, de longe, o algoritmo de troca mais eficiente, mostrando-se uma ótima opção para ordenar um conjunto de elementos, independentemente do tamanho ou ordem da lista, pois, assim como o gráfico e a tabela anterior ilustram, seu tempo de execução é de menos de 1 segundo.

Porém, mesmo com sua alta eficiência, o Shell Sort apresenta algumas desvantagens, como o fato do número de vezes que executa as operações ser sempre a mesma, ou seja, mesmo que a lista de elementos esteja previamente ordenada, o algoritmo será executado e o número de comparações permanecerá igual. Além disso, o Shell Sort não é um algoritmo estável, então fará trocas entre posições com elementos de mesmo valor.

5.5 Merge Sort

A partir da implementação do código apresentado na seção 3.5 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Merge Sort, como demonstrado na Tabela 15 e na Figura 15 a seguir.

————————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0	0,001	0,007	0,082
Decrescente	0	0	0	0	0,008	0,083
Aleatório	0	0	0	0,001	0,015	0,158

Tabela 15: Tabela que demonstra o tempo de execução do algoritmo Merge Sort em vetores com diferentes quantidades de dados e em diferentes ordens

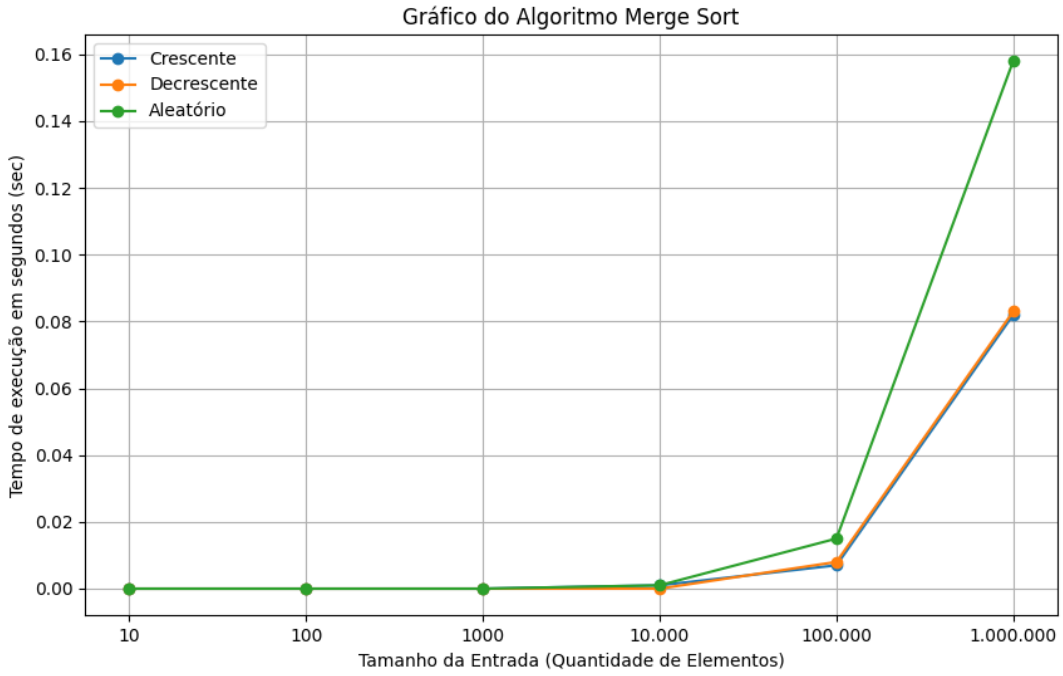

Figura 15: Gráfico ilustrando o comportamento do algoritmo Merge Sort em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

O algoritmo Merge Sort possui grande eficiência para uma grande quantidade de dados, podendo ordená-los em menos de 1 segundo. Além disso, é um algoritmo estável. Em

conjuntos pequenos de dados, porém, apresenta menor performance se comparado a outros algoritmos de ordenação, como Insertion Sort.

Como desvantagens, podem ser citadas o uso de um vetor auxiliar para operar, o que aumenta consumo de memória e tempo de execução, e o fato de que o Merge Sort executa divisões e intercalações no conjunto de dados mesmo que este já esteja parcialmente ordenado, o que poderia acarretar em overhead computacional.

5.6 Quick Sort

A partir da implementação do código apresentado na seção 3.6 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Quick Sort, como demonstrado na Tabela 16 e na Figura 16 a seguir.

————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0,001	0,001	0,013	0,133
Decrescente	0	0	0	0,001	0,012	0,147
Aleatório	0	0	0	0,002	0,031	0,366

Tabela 16: Tabela que demonstra o tempo de execução do algoritmo Quick Sort em vetores com diferentes quantidades de dados e em diferentes ordens

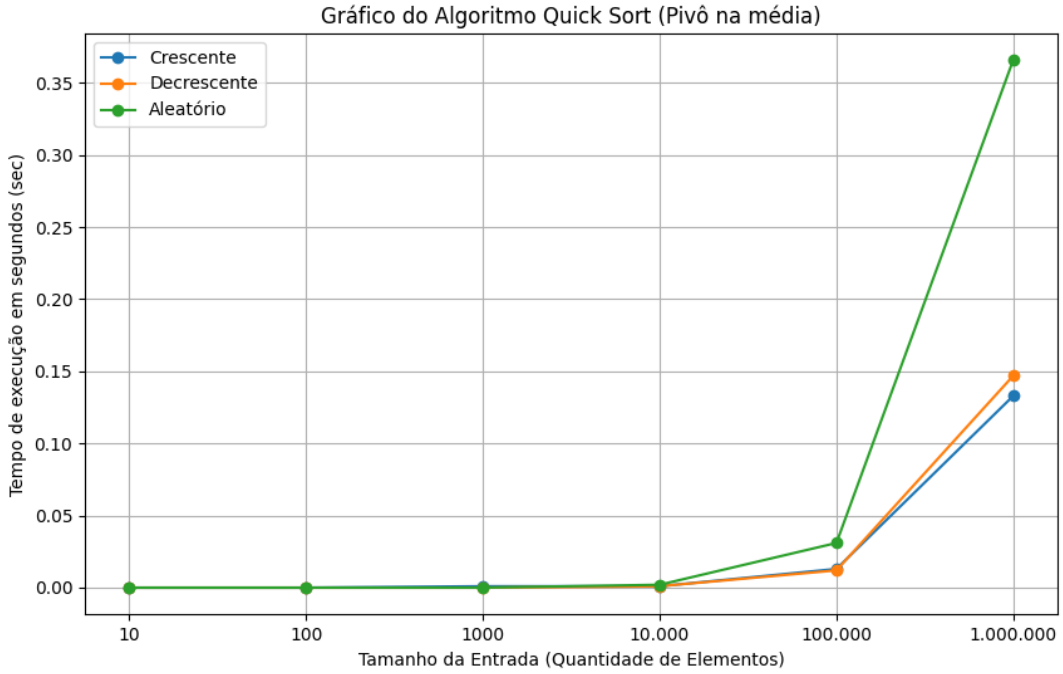

Figura 16: Gráfico ilustrando o comportamento do algoritmo Quick Sort (com o pivô na média) em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

O Quick Sort tem como principal característica sua eficiência: é um algoritmo de ordenação muito rápido, de complexidade $O(n.log(n))$.

Sua eficiência depende da escolha de um bom pivô, pois se um pivô ruim (como a posição inicial na lista de elementos) for escolhido, isso poderá gerar seu pior caso, que é de complexidade $O(n^2)$. No algoritmo representado na Tabela 16 e na Figura 16, o pivô selecionado é a média entre o início, o final e o elemento do meio do conjunto de dados, o que faz com que as partições sejam balanceadas e o melhor caso seja gerado.

5.7 Quick Sort com pivô no início

A partir da implementação do código (devidamente modificado para que o pivô definido no particiona estivesse no início) apresentado na seção 3.6 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o com-

portamento do algoritmo Quick Sort com pivô do valor da posição inicial, como demonstrado na Tabela 17 e na Figura 17 a seguir.

————————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0,003	0,246	24,411	2469,278
Decrescente	0	0	0,003	0,264	27,86	2636,745
Aleatório	0	0	0	0,003	0,029	0,415

Tabela 17: Tabela que demonstra o tempo de execução do algoritmo Quick Sort (com pivô no início) em vetores com diferentes quantidades de dados e em diferentes ordens

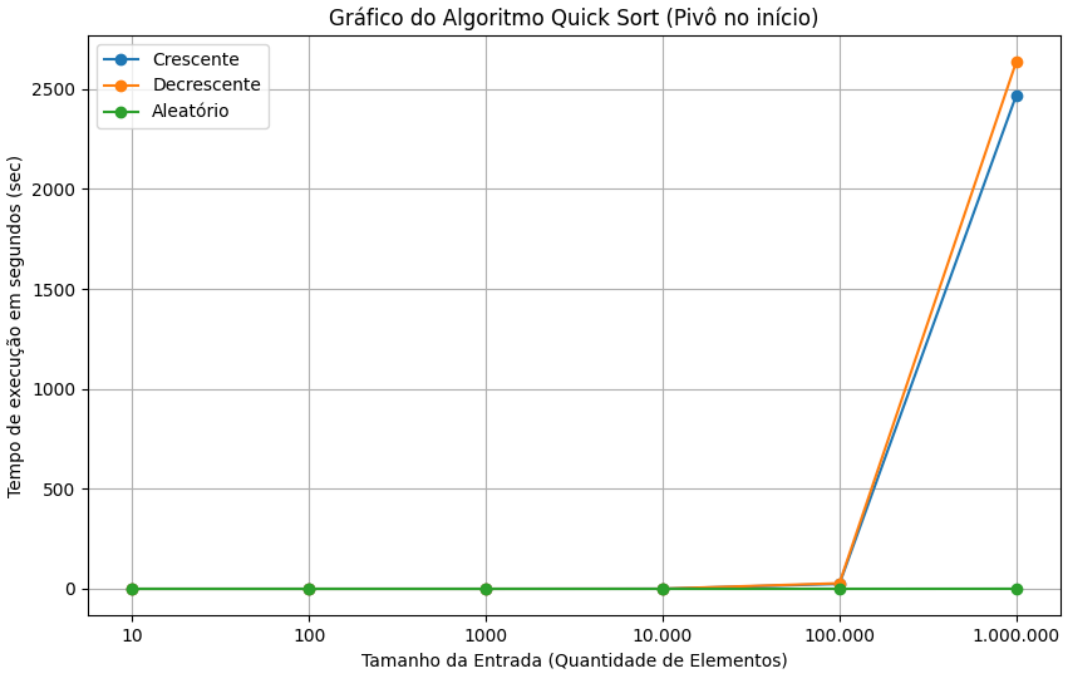

Figura 17: Gráfico ilustrando o comportamento do algoritmo Quick Sort (com o pivô no início) em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

Ilustrando o pior caso, é possível notar, ao observar a Tabela 17 e a Figura 17, que quando o pivô está na posição inicial, a eficiência do Quick Sort é extremamente baixa, apresentando um tempo de execução muito longo.

5.8 Quick Sort com pivô aleatório

A partir da implementação do código (devidamente modificado para que o pivô definido no particiona estivesse aleatório) apresentado na seção 3.6 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Quick Sort com pivô de valor aleatório, como demonstrado na Tabela 18 e na Figura 18 a seguir.

————————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0	0,001	0,021	0,336
Decrescente	0	0	0	0,002	0,031	0,434
Aleatório	0	0	0	0,003	0,044	0,416

Tabela 18: Tabela que demonstra o tempo de execução do algoritmo Quick Sort (com pivô aleatório) em vetores com diferentes quantidades de dados e em diferentes ordens

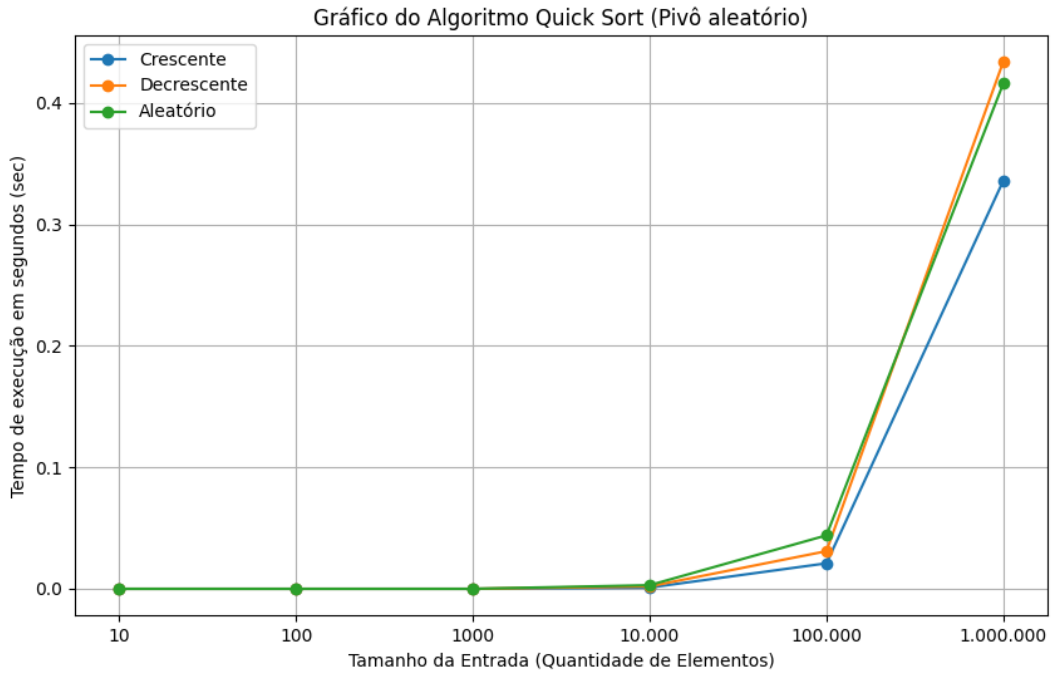

Figura 18: Gráfico ilustrando o comportamento do algoritmo Quick Sort (com o pivô no início) em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

A Tabela 18 e a Figura 18 ilustram o caso médio do Quick Sort, mostrando o comportamento médio esperado do Quick Sort, quando o pivô é um valor aleatório. É possível observar que seu tempo de execução é baixo, e sua eficiência é grande.

5.9 Heap Sort

A partir da implementação do código apresentado na seção 3.7 e de testes feitos logo em seguida, com vetores de dados de diferentes tamanhos e em diferentes ordens, pode-se ilustrar o comportamento do algoritmo Quick Sort, como demonstrado na Tabela 19 e na Figura 19 a seguir.

————	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Crescente	0	0	0,001	0,004	0,044	0,507
Decrescente	0	0	0	0,005	0,045	0,666
Aleatório	0	0	0,001	0,004	0,061	0,857

Tabela 19: Tabela que demonstra o tempo de execução do algoritmo Heap Sort em vetores com diferentes quantidades de dados e em diferentes ordens

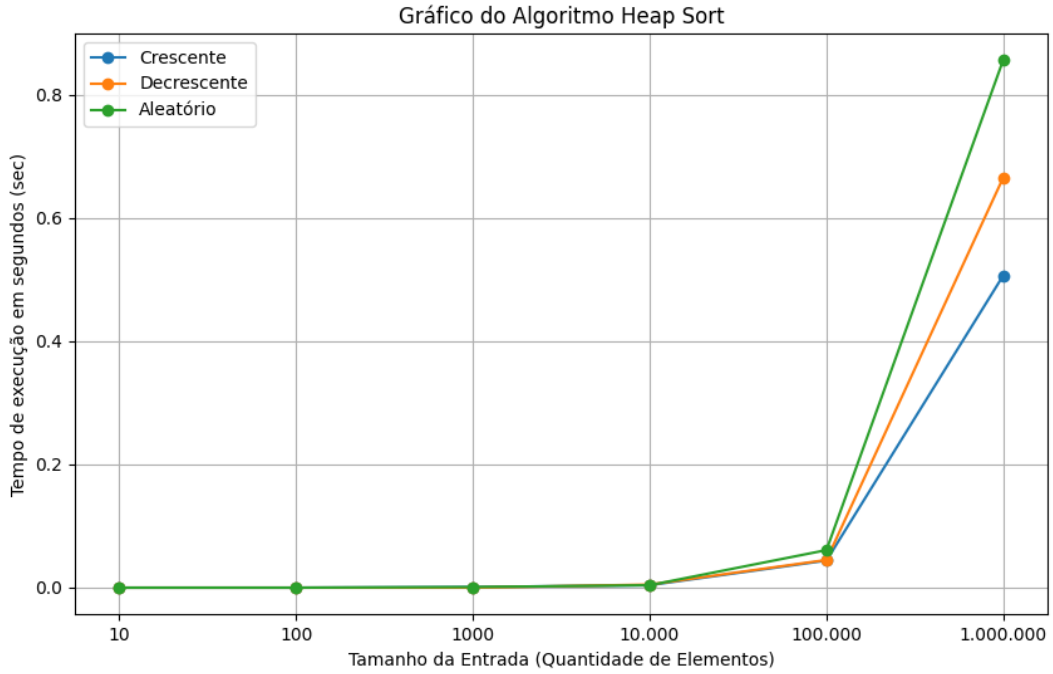

Figura 19: Gráfico ilustrando o comportamento do algoritmo Heap Sort em um vetor ordenado de diferentes formas (crescente, decrescente e aleatória) e com diferentes números de dados (10, 100, 1000, 10000, 100000 e 1000000).

Como um dos algoritmos mais reconhecidos pela sua eficiência em uso de memória e tempo de execução, o Heap Sort é consistente e possui complexidade $O(n.log(n))$, independentemente da entrada, além disso, não utiliza memória adicional como outros algoritmos de Divisão e Conquista já apresentados anteriormente, como o Quick Sort, o que denota a eficiência do Heap Sort, especialmente para arquivos grandes. Como desvantagens, pode-se citar que sua complexidade faz com que o Heap Sort não seja recomendado para arquivos com poucos elementos, e além disso, o Heap Sort não é um algoritmo estável.

5.10 Comparação entre algoritmos

Após apresentar a complexidade, tabelas e gráficos contendo todas as informações dos algoritmos, é importante notar que existem diferenças entre suas performances, mesmo que os elementos recebidos sejam os mesmos. Esta seção se dedica a comparar os tempos dos diferentes algoritmos, tendo como base a quantidade de elementos e como estes estavam ordenados previamente (em ordem crescente ou ordem descrescente). Os algoritmos foram testados a partir da implementação dos códigos apresentados na Seção 3 deste relatório.

A seguir, estão tabelas e gráficos que representam a performance entre diferentes algoritmos para os mesmos conjuntos de dados. Os gráficos e tabelas estão divididos dentre os algoritmos de troca (Insertion, Bubble, Selection e Shell) e algoritmos de divisão e conquista (Merge, Quick e Heap).

5.10.1 Entrada ordenada de forma crescente

Quando a lista de elementos já está previamente ordenada, tem-se um comportamento muito semelhante na maioria dos algoritmos de troca, isto é, à medida que o número de elementos cresce, o tempo de execução do algoritmo também cresce de maneira proporcional, ou seja, o algoritmo segue sua execução normalmente, com 2 exceções. O Shell Sort, por ser um algoritmo com uma natureza mais eficiente, possui um aumento pouco significativo em seu tempo de execução. O Insertion Sort, por sua vez, encontra seu melhor caso neste cenário, e seu tempo de execução é muito próximo de zero, mesmo que o número de elementos seja alto.

A Tabela 20 em conjunto com a Figura 20, ilustram este comportamento.

Algoritmo	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Insertion Sort	0	0	0	0	0	0,003
Bubble Sort	0	0	0,002	0,194	19,638	1959,074
Selection Sort	0	0	0,001	0,103	10,272	1021,027
Shell Sort	0	0	0	0	0,002	0,026

Tabela 20: Comparação do tempo de execução dos algoritmos em entradas de diferentes tamanhos

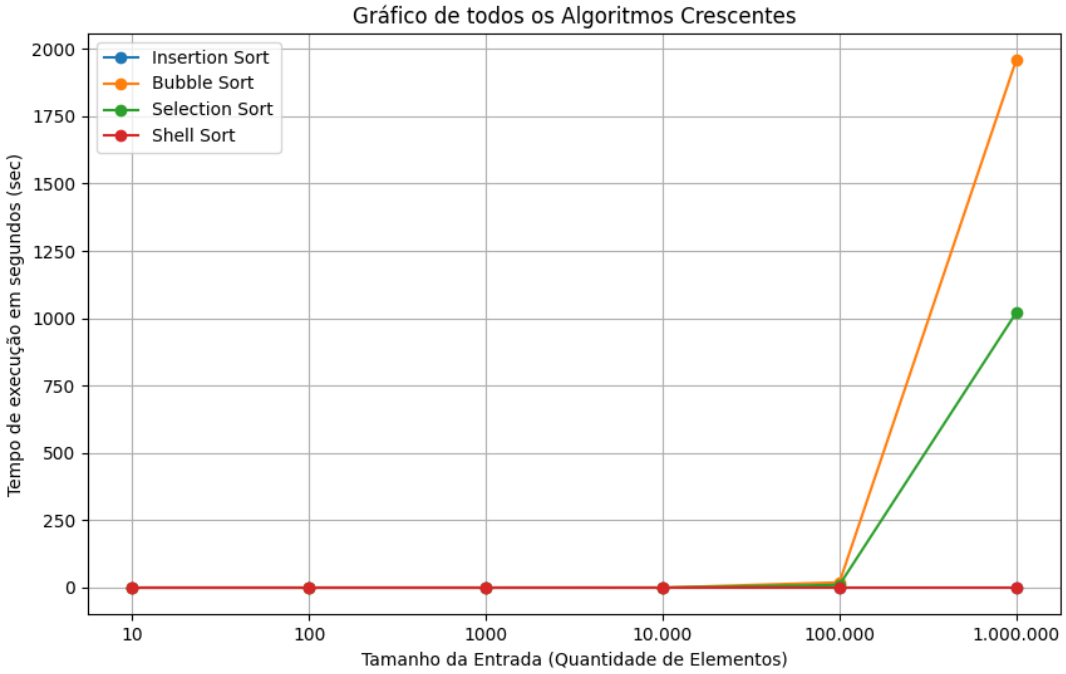

Figura 20: Gráfico de comparação entre o tempo de execução dos algoritmos de troca, quando a entrada é crescente

Dentre os algoritmos de divisão e conquista, o comportamento também é o mesmo que o observado entre os algoritmos de troca, porém o tempo de execução se mantém baixo, bem próximo de 0. Porém, há um comportamento diferente para o algoritmo Quick Sort quando seu pivô está no início da lista de elementos, pois, nesta situação, o algoritmo está em seu pior caso, sendo necessária uma quantidade exorbitante de tempo para concluir sua execução, se comparado aos outros algoritmos que seguem o paradigma de divisão e conquista.

A Tabela 21, em conjunto com a Figura 21, ilustram este comportamento.

Algoritmo	10	100	1.000	10.000	100.000	1.000.000
Merge Sort	0	0	0	0,001	0,007	0,082
Quick Sort (média)	0	0	0,001	0,001	0,013	0,133
Quick Sort (início)	0	0	0,003	0,246	24,411	2469,278
Quick Sort (aleatório)	0	0	0	0,001	0,021	0,336
Heap Sort	0	0	0,001	0,004	0,044	0,507

Tabela 21: Comparação do tempo de execução dos algoritmos em entradas de diferentes tamanhos

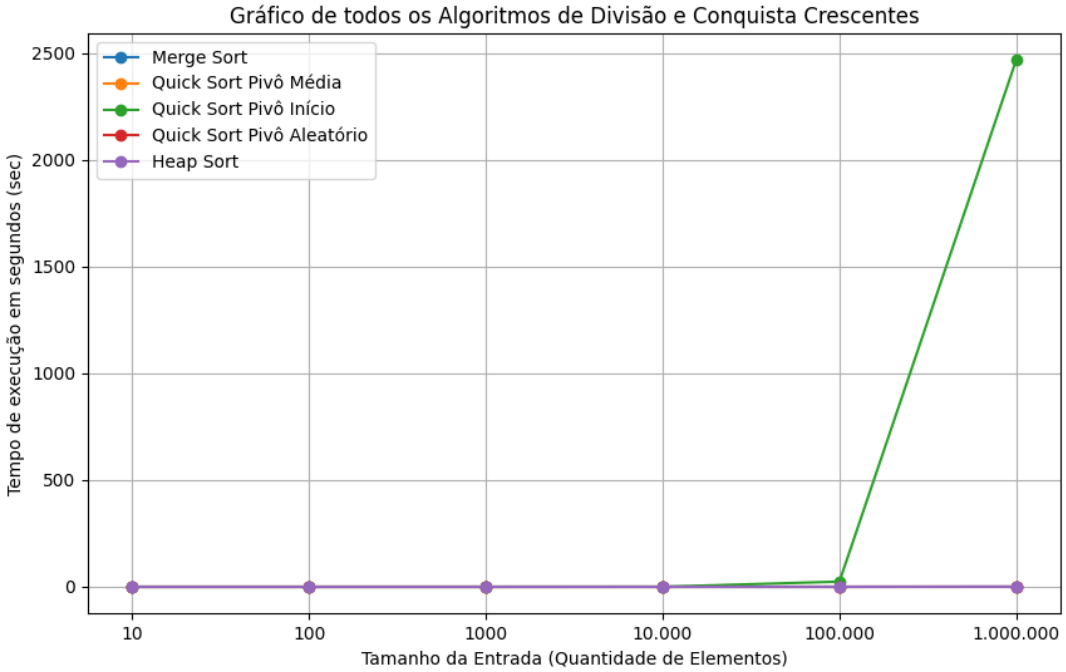

Figura 21: Gráfico de comparação entre o tempo de execução dos algoritmos de divisão e conquista, quando a entrada é crescente

5.10.2 Entrada ordenada de forma decrescente

Quando a lista de elementos está previamente organizada em ordem descrescente, notam-se mudanças no comportamento de 2 dos algoritmos de troca: o Bubble Sort e o Insertion Sort, não havendo mudanças notáveis no comportamento do Selection Sort e do Shell Sort. O Bubble Sort encontra seu pior caso, sendo possível notar um significativo aumento em seu tempo de execução. Da mesma forma, o Insertion Sort também encontra seu pior caso, havendo um grande aumento em seu tempo de execução, especialmente se comparado com

o desempenho que apresenta com uma lista de elementos já em ordem crescente, visto na Seção 5.7.1.

A Tabela 22, em conjunto com a Figura 22, ilustram esse comportamento.

Algoritmo	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Insertion Sort	0	0	0,001	0,116	11,197	1145,033
Bubble Sort	0	0	0,003	0,269	28,153	2605,384
Selection Sort	0	0	0,001	0,097	9,678	1001,792
Shell Sort	0	0	0	0	0,005	0,045

Tabela 22: Comparação do tempo de execução dos algoritmos em entradas de diferentes tamanhos

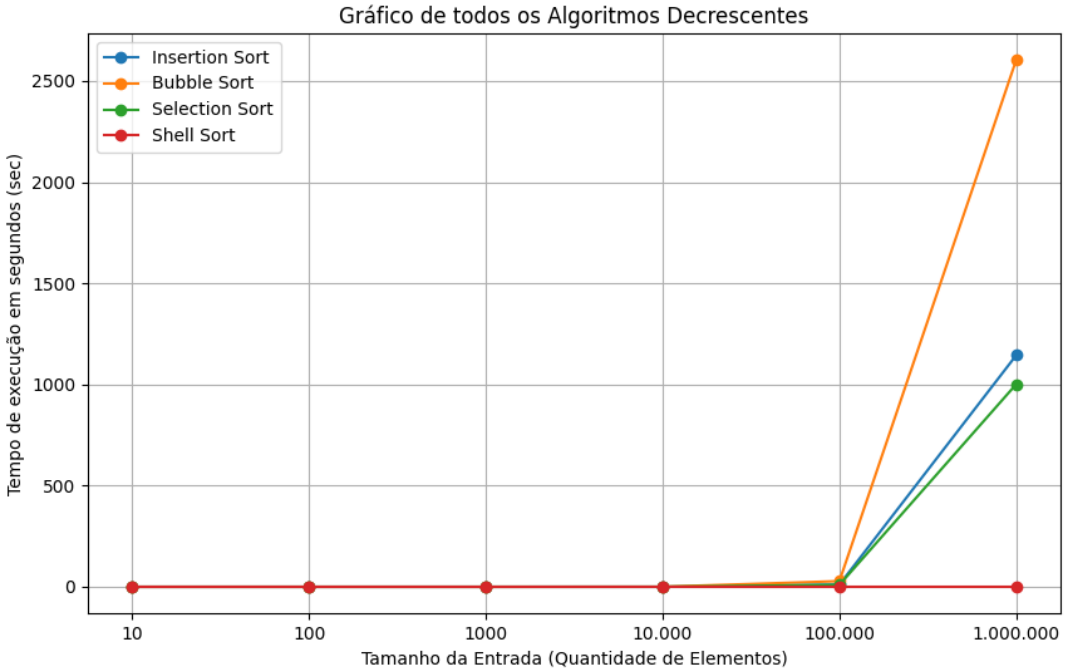

Figura 22: Gráfico de comparação entre o tempo de execução dos algoritmos de troca, quando a entrada é decrescente

Já dentre os algoritmos de divisão e conquista, quando os elementos estão previamente ordenados em ordem decrescente, ocorre o mesmo que visto na situação em que os elementos estão em ordem crescente: não há mudanças notáveis na performance, com exceção do algoritmo Quick Sort. Quando o pivô escolhido para a execução do Quick Sort está na posição inicial da lista, ele está em seu pior caso, e por causa disto, há um aumento significativo em seu tempo de execução, comparado à outras situações. Este aumento de tempo é mais notável quando o número de elementos é alto.

A Tabela 23, em conjunto com a Figura 23, ilustram esse comportamento.

Algoritmo	**10**	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Merge Sort	0	0	0	0	0,008	0,083
Quick Sort (média)	0	0	0	0,001	0,012	0,147
Quick Sort (início)	0	0	0,003	0,264	27,86	2636,745
Quick Sort (aleatório)	0	0	0	0,002	0,031	0,434
Heap Sort	0	0	0	0,005	0,045	0,666

Tabela 23: Comparação do tempo de execução dos algoritmos em entradas de diferentes tamanhos

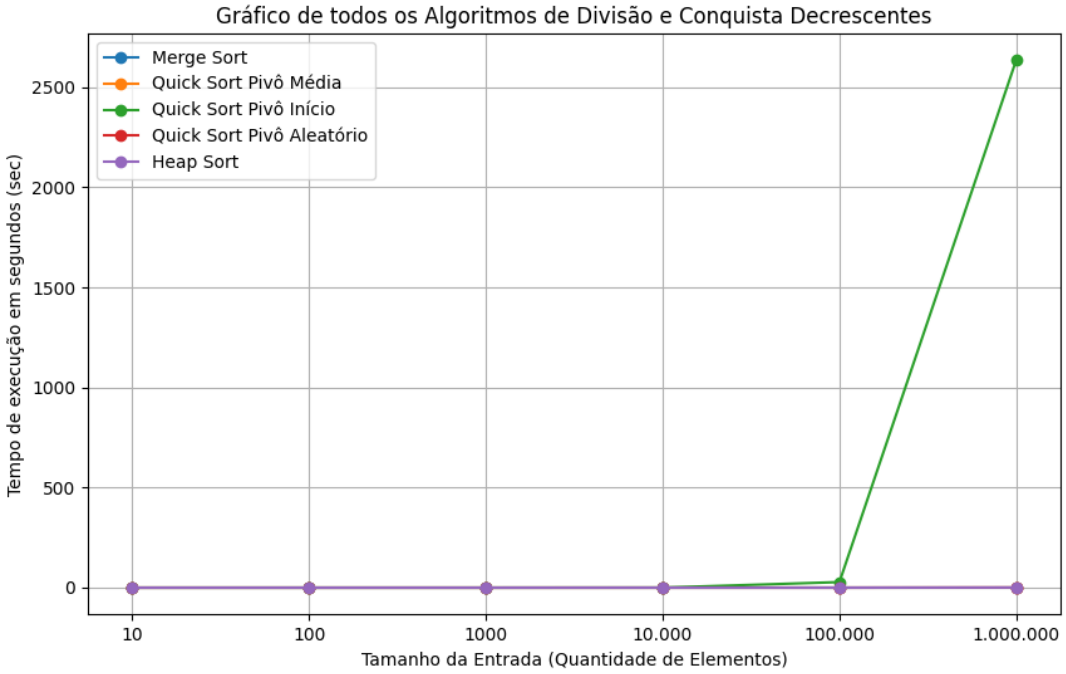

Figura 23: Gráfico de comparação entre o tempo de execução dos algoritmos de divisão e conquista, quando a entrada é decrescente

6 CONCLUSÃO

Portanto, levando em consideração os estudos e os testes feitos nos diversos Métodos de Ordenação, como InsertionSort, SelectionSort e MergeSort, ao longo deste trabalho, pode-se perceber que em uma ordenação com 10 e 100 instâncias, os algoritmos possuem o mesmo desempenho, tanto para crescente, decrescente e randômico. Porém, a partir do momento em que o conjunto de dados se aproxima de 1.000.000, seu comportamento passa a diferenciar-se, sendo possível observar mudanças significativas em alguns algoritmos a partir do momento em que o número de dados é 10.000.

O algoritmo Insertion Sort possui, assim como foi observado nos gráficos, um tempo de execução imediato para vetores com 100 elementos, porém, a partir de 1000 elementos, este tempo começa a crescer. Por exemplo o algoritmo Insertion Sort, agindo em um vetor decrescente com 10000 elementos, completa sua execução em 0,1 segundo, e em um vetor decrescente com 1000000 elementos, a execução só pode ser completa após 1145 segundos (aproximadamente 19 minutos). Isto ilustra que esse algoritmo é mais útil para lidar com poucos dados ou com dados que já estão previamente ordenados.

O algoritmo Bubble Sort possui notáveis problemas, pois quando sua entrada é muito grande (como 1 milhão de dados, por exemplo), seu tempo de execução é maior do que todos os outros algoritmos vistos até agora. Mesmo que o algoritmo Bubble seja mais simples e intuitivo, seu desempenho com problemas grandes faz com que não seja a melhor opção de algoritmo.

O algoritmo Selection Sort possui praticamente o mesmo desempenho, independentemente da ordenação dos dados em questão, de modo que o que influencia em seu tempo de execução é apenas a quantidade de dados, que é diretamente proporcional ao tempo de execução, ou seja, quanto maior o número de dados, maior o tempo de execução, possui um desempenho melhor que outros algoritmos abordados aqui, mas não chega a ser o melhor.

O algoritmo Shell Sort é um dentre os que apresentam melhor performance, pois independentemente do tamanho da entrada ou da ordenação da mesma, o algoritmo levou menos de 1 segundo para ser executado, conforme visto nos gráficos apresentados.

O algoritmo Merge Sort também leva menos de 1 segundo para completar seu funcionamento, independentemente do tamanho da entrada ou da ordem, demonstrando sua alta eficácia para lidar com dados de diferentes ordens e tamanhos. Vale destacar o fato de que o algoritmo possui praticamente o mesmo tempo de execução entre a ordem crescente ou decrescente.

O algoritmo Quick Sort possui uma notável velocidade para todas as ordens e quantidades de elementos, porém, sua eficiência é quase inteiramente decidida com a posição em que o pivô foi colocado, tendo em vista que, quando o pivô é escolhido com base na média aritmética dentre os valores do conjunto de dados, encontra seu melhor caso e possui um tempo de execução muito baixo. Porém, se o pivô escolhido for na posição inicial, encontra seu pior caso e seu tempo de execução aumenta de forma galopante, chegando a ser um dos maiores tempos de execução dentre todos os algoritmos estudados neste relatório, com mais de 2600 segundos.

O algoritmo Heap Sort é um dos mais eficientes, tendo pouca variação de tempo, independentemente da ordem dos dados ou da quantidade destes.

À luz disto, é possível concluir que os algoritmos que atuam sob o paradigma de divisão e conquista, explorados a partir da seção 3.5, são os mais eficientes, com baixo tempo de execução, salvo algumas exceções.

7 REFERÊNCIAS BIBLIOGRÁFICAS

1. Cormen, Thomas H., et al. "Algoritmos: teoria e prática." *Editora Campus* 2 (2002): 296.

2. Ziviani, Nivio. *Projeto de algoritmos: com implementações em Pascal e C*. Vol. 2. Luton: Thomson, 2004.

3. Murça, Leonardo Aguilar. "Análise de Complexidade dos Métodos de Ordenação."

4. Coelho, Hebert, and Nádia Félix. "Métodos de Ordenação: Selection, Insertion, Bubble, Merge (Sort)."

5. de Sene Dias, Andrew Carlos, Nayara Almeida Vilela, and Walteno Martins Parreira Júnior. "ANÁLISE SOBRE ALGUNS MÉTODOS DE ORDENAÇÃO DE LISTAS: SELEÇÃO, INSERÇÃO E SHELLSORT."Intercursos Revista Científica 13.1 (2014).

8 Função para calcular o tempo

void operacoes(int tipo, int tamanho)

$clock_t start_t, end_t$; //Variavel para guardar o tempo

double tempoGasto;

int * vetor = gerarSequencia(tipo, tamanho); //Gera sequencia de numeros

//Salva a entrada de numeros

salvarEntrada(tipo, tamanho, vetor);

$start_t = clock()$; //Calcula o tempo atual

insertionSort(vetor, tamanho); //Ordena o Vetor

$end_t = clock()$; //Calcula o tempo apos ordenação

tempoGasto = $((end_t - start_t)$ / (double)$CLOCKS_PER_SEC)$; //Calcula diferença de tempo

salvarTempo(tipo, tamanho, tempoGasto); //Salva o tempo gasto

//Salva a saida do programa

salvarSaida(tipo, tamanho, vetor);

//Libera a memoria

free(vetor);