Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT
Bacharelado em Sistemas de Informação – BSI

Relatório técnico sobre a disciplina de Estrutura de Dados

Iago Gouveia Gurgel

Orientador: Prof. Dr. João Paulo Souza Medeiros

Relatório técnico apresentado ao Curso de Bacharelado em Sistemas de Informação como parte das atividades na disciplina de Estrutura de Dados.

Laboratório de Elementos do Processamento da Informação – LabEPI
Caicó, RN, 28 de agosto de 2024

Relatório técnico sobre a disciplina de Estrutura de Dados

Iago Gouveia Gurgel

Relatório aprovado em 28 de agosto de 2024 pelo examinador:

Prof. Dr. João Paulo Souza Medeiros (orientador) .... DCT/UFRN

"When does a man die? When he is hit by a bullet? No!
When he suffers a disease? No!
When he ate a soup made out of a poisonous mushroom? No!
A man dies when he is forgotten!"
Dr. Hiruruku

Sumário

1 Introdução **1**
1.1 O que são algoritmos e qual a sua importância no mundo real? 1
1.2 O que são estruturas de dados? 2
1.3 Procedimento de análise de eficiência de um algoritmo 2
1.4 Notação assintótica e definições formais 3
1.4.1 Notação $O(n)$ 3
1.4.2 Notação $\Omega(n)$ 3
1.4.3 Notação $\Theta(n)$ 4

2 Análise de algoritmos de ordenação **5**
2.1 Definição formal do problema de ordenação 5
2.2 Insertion Sort 5
2.2.1 Introdução 5
2.2.2 Implementação 6
2.2.3 Análise do algoritmo e notação assintótica 6
2.2.4 Comparação teórica-prática 7
2.2.5 Discussão sobre tempo de execução e uso de memória 7
2.3 Selection Sort 8
2.3.1 Introdução 8
2.3.2 Implementação 8
2.3.3 Análise do algoritmo e notação assintótica 8
2.3.4 Comparação teórica-prática 9
2.3.5 Discussão sobre tempo de execução e uso de memória 10
2.4 Heap Sort 10
2.4.1 Introdução 10
2.4.2 Implementação 10
2.4.3 Análise do algoritmo e notação assintótica 11
2.4.4 Comparação teórica-prática 12
2.4.5 Discussão sobre tempo de execução e uso de memória 12
2.5 Merge Sort 12
2.5.1 Introdução 12
2.5.2 Implementação 13
2.5.3 Análise do algoritmo e notação assintótica 14
2.5.4 Comparação teórica-prática 14
2.5.5 Discussão sobre tempo de execução e uso de memória 14
2.6 Quick Sort 15
2.6.1 Introdução 15

2.6.2 Implementação . . . 15
2.6.3 Análise do algoritmo e notação assintótica . . . 16
2.6.4 Comparação teórica-prática . . . 17
2.6.5 Discussão sobre tempo de execução e uso de memória . . . 17
2.7 Counting Sort . . . 17
2.7.1 Introdução . . . 17
2.7.2 Implementação . . . 18
2.7.3 Análise do algoritmo e notação assintótica . . . 19
2.7.4 Comparação teórica-prática . . . 19
2.7.5 Discussão sobre tempo de execução e uso de memória . . . 19
2.8 Comparações . . . 20

3 Análise de algoritmos de árvores e tabelas de dispersão 23

4 Conclusões 25

Referências Bibliográficas 27

Capítulo 1

Introdução

"He who knows all the answers has
not been asked all the questions."
Confucius

O estudo e análise de algoritmos é fundamental para o desenvolvimento do conhecimento no estudo da Computação. Para isso, foi realizado um estudo baseado nos conteúdos da disciplina de Estrutura de Dados ministrada no curso de Bacharelado de Sistemas de Informação da UFRN com o material adicional Cormen et al. (2022) com objetivo de compreender profundamente os conteúdos da disciplina.

1.1 O que são algoritmos e qual a sua importância no mundo real?

Para Cormen et al. (2022), os algoritmos são qualquer procedimento computacional bem definido capaz de produzir um conjunto de valores como saída, a partir de um conjunto de valores como entrada. Um algoritmo também pode ser descrito como um conjunto de instruções ou passos que devem ser executados com objetivo de produção de um valor significativo para o contexto qual foi executado. Sua importância é fundamental para qualquer sistema computacional, como exemplo, pode-se imaginar o problema de ordenação de valores, sendo este, um exemplo bastante recorrente na computação, o mesmo pode ser definido de maneira formal como:

Input: Uma sequência de n números $(a_1, a_2, a_3, \ldots, a_n)$

Output: Uma reordenação $(a_1, a_2, a_3, \ldots, a_n)$ da sequência para qual $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$.

Para medir a importância dos algoritmos na sociedade, podemos visualizar algumas das suas aplicações no mundo real:

1. Ordenação de valores
2. Compressão de dados
3. O menor caminho entre dois locais

4. Cálculos estatísticos
5. Segurança
6. Transformações em dados

Para cada uma dessas aplicações, existem algoritmos que solucionam o problema em questão. No entanto, é importante ressaltar que, entre as múltiplas implementações que solucionam um problema, nem todas serão implementações ótimas, ou seja, otimizadas e eficientes para solucionar o problema no menor tempo possível. No capítulo 2, será possível observar as diferenças em tempo de execução de diferentes implementações de soluções para um mesmo problema.

1.2 O que são estruturas de dados?

As estruturas de dados são, de acordo com Cormen et al. (2022), formatos de armazenamento de dados que permitem fácil acesso e modificação. Também, permitem organizar os dados de maneiras diferentes, possibilitando assim que algoritmos manipulem os dados de maneiras diferentes. Karumanchi (2017) discorre sobre a divisão das estruturas de dados em 2 tipos:

- Estruturas de dados lineares: Onde os elementos são acessados de forma sequencial. Exemplos como: Listas encadeadas, Pilhas e Filas.
- Estruturas de dados não-lineares: Onde os elementos são armazenados e acessados de forma não-linear. Exemplos: Árvores e grafos.

1.3 Procedimento de análise de eficiência de um algoritmo

Para realizar a análise e medição de um algoritmo, é necessário compreender o algoritmo e seu contexto. Para isso, vamos tomar como exemplo o algoritmo de busca por um elemento em um vetor de forma linear. Esse algoritmo é nomeado de busca linear e pode ser visto em 1.1.

Algoritmo 1.1: Algoritmo de Busca linear

```
LINEAR-SEARCH(A, x)
n ← len(A)
for y from 1 to n
    if A[y] = x then
        return y
return NILL
```

Ao analisar o algoritmo apresentado, pode-se compreender que seu funcionamento é baseado em percorrer de forma sequencial e linear o vetor testando pelo elemento a ser buscado. Se esse elemento existir no vetor, seu índice será retornado como valor da função, se não, será retornado o valor nulo. Para medir sua eficiência, é necessário calcular seu tempo de execução. Para isso, é necessário perceber que a função de busca linear é intrinsicamente associada ao tamanho n do vetor A e a probabilidade da existência de x no vetor. Imaginando que cada linha do pseudo-código apresentado em 1.1 executa em um tempo específico, podemos pensar a equação 1.1 para seu tempo de execução de pior caso. Observe que: para a linha z do código temos C_z como seu tempo de execução.

$$T(n) = C_2 + \sum_{i=1}^{n}(C_3 + C_4) + C_5 + C_6 \tag{1.1}$$

Como definido em Cormen et al. (2022), pode-se expressar essa equação como $T(n) = an + b$, com $a = (C_3 + C_4)$ e $b = (C_2 + C_5 + C_6)$ mas, antes que seja determinado a ordem do tempo de execução dessa função, é fundamental compreender a segunda determinante para sua métrica de eficiência. Um fator previamente mencionado foi que, para realizar-se a medida do tempo de execução, também é necessário compreender a probabilidade p que um elemento qualquer x esteja presente nesse vetor A, como apresenta a equação 1.2.

$$p = \frac{n}{\mathbb{Z}} \tag{1.2}$$

Ajustando a equação $T(n) = an + b$ com a probabilidade p de que x pertença a A, percebe-se que:

$$T(n) = (1 - p)(an + b) + p(an + b)$$

Com isso, é possível determinar que o tempo de execução do algoritmo é linear, já que a probabilidade p de que x pertença a A é muito baixa, e mesmo que $x \in A$, a probabilidade de ocorrência do pior caso é mais alta que do melhor caso, sendo o pior caso $T(n) = an + b$ e o melhor caso $T(n) = 1$.

1.4 Notação assintótica e definições formais

Ao realizar a análise de um algoritmo, é possível perceber a sua razão de crescimento. O estudo das razões de crescimento do tempo de execução dos algoritmos é o estudo da eficiência assintótica de um algoritmo, isso é, qual a razão de um crescimento de um algoritmo qualquer quando o número de entradas do mesmo tende ao infinito.

1.4.1 Notação $O(n)$

A notação $O(n)$ indica o limite superior da razão de crescimento em uma função de comportamento assintótico. Para o caso do algoritmo 1.1, pode-se determinar que:

$$O(n) = n^x \; \forall \; x \geq 1$$

Ou também, qualquer função com razão de crescimento superior ao cresimento linear pode ser usado como $O(n)$ da função de busca linear.

1.4.2 Notação $\Omega(n)$

A notação $\Omega(n)$ indica o limite inferior da razão de crescimento em uma função de comportamento assintótico. Para o caso do algoritmo 1.1, pode-se determinar que:

$$\Omega(n) = n^x \; \forall \; x \leq 1$$

Ou também, qualquer função com razão de crescimento inferior ao cresimento linear pode ser usado como $\Omega(n)$ da função de busca linear.

1.4.3 Notação $\Theta(n)$

A notação $\Theta(n)$ indica o limite justo da razão de crescimento em uma função de comportamento assintótico. Determina-se $\Theta(n)$ de uma função qualquer provando que seu cresimento é tanto $O(f(n))$ como $\Omega(f(n))$, dessa forma, prova-se que essa função é $\Theta(f(n))$. Para o caso do algoritmo 1.1, pode-se determinar que:

$$O(n) = n$$
$$\Omega(n) = n$$
$$\Theta(n) = n$$

Dessa forma, é possível determinar que $\Theta(n) = n$ para a busca linear.

Capítulo 2

Análise de algoritmos de ordenação

"Nothing in life is to be feared, it is only to be understood.
Now is the time to understand more, so that we may fear less."
Marie Curie

2.1 Definição formal do problema de ordenação

O problema de ordenação, como apresentado no Capítulo 1, é um problema bastante recorrente na Computação. Sua definição formal consiste em:

Input: Uma sequência de n números $(a_1, a_2, a_3, \ldots, a_n)$

Output: Uma reordenação $(a_1, a_2, a_3, \ldots, a_n)$ da sequência para qual $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$.

Para este algoritmo, foram pensadas diversas soluções e neste capítulo, será realizado o estudo e compreensão das mesmas. É importante perceber que, podemos aproximar a ordem de crescimento do algoritmo de ordenação por inserção de forma bastante precisa utilizando as notações assintóticas, mas não é possível determinar o tempo de execução para um número de entradas n com muita precisão devido a aspectos computacionais como:

- Chaveamento e prioridade de processo no sistema operacional
- Tempo de execução de cada instrução do código compilado
- Velocidade de clock inconstante

2.2 Insertion Sort

2.2.1 Introdução

O Insertion Sort é um algoritmo de ordenação relativamente eficiente com número de entradas baixo, seu funcionamento consiste em consumir o vetor a ser ordenado de forma sequencial buscando pela localização correta para o elemento no vetor e inserindo-o na mesma.

2.2.2 Implementação

Para o algoritmo de ordenação baseado em inserção, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.1.

Algoritmo 2.1: Algoritmo de ordenação por inserção

```
INSERTION−SORT(A)
n ← len(A)
i ← 2
while i < n do
    j ← i
    while j > 1 and A[j − 1] > A[j]
        swap(A[j − 1], A[j])
        j ← j − 1
    i ← i + 1
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
void iSort(int * v, int n)
{
    int j, i;
    i = 1;
    while(i < n)
    {
        j = i;
        while(j > 0 && v[j - 1] > v[j])
        {
            swap(&v[j - 1], &v[j]);
            j -= 1;
        }
        i += 1;
    }
}
```

2.2.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação por inserção, é necessário perceber os tempos de execução de cada linha do pseudo-código 2.1. Nesse caso, pode-se ter como base a equação 2.1.

$$T(n) = C_2 + C_3 + \sum_{i=1}^{n}(C_4 + C_5 + C_9) + \sum_{j=1}^{n^2}(C_6 + C_7 + C_8) \tag{2.1}$$

Portanto, é plausível relacionar 2.1 com 2.2.

$$T(n) = an^2 + bn + c \; para \; a = (C_6 + C_7 + C_8), \;\; b = (C_4 + C_5 + C_9) \; e \; c = (C_2 + C_3) \tag{2.2}$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação por inserção:

$$
\begin{aligned}
O(n) &= n^x \forall x \geq 2 \\
\Omega(n) &= n^x \forall x \leq 2 \\
\Theta(n) &= n^2
\end{aligned}
$$

2.2.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação por inserção, pode-se observar o gráfico em 2.1 que apresentam o tempo de execução para o algoritmo de ordenação por inserção para 2048 casos diferentes com o número de entradas n variando de 1 a 65536.

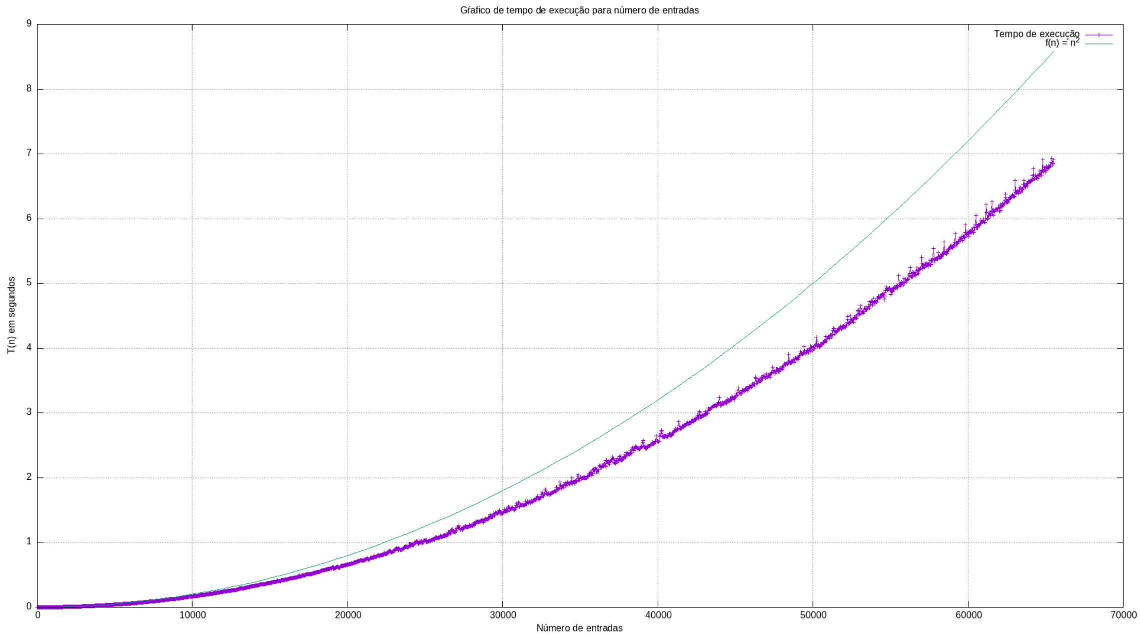

Figura 2.1: Gráfico com tempo de execução do algoritmo de ordenação por inserção

No gráfico 2.1, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função quadrática para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.1, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.2.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação por inserção é eficiente para vetores com poucas entradas a partir de um certo limiar e do contexto, existem outros algoritmos de ordenação mais eficientes que o mesmo. Sobre seu uso de memória, é constante porque utiliza apenas algumas variáveis auxiliares.

2.3 Selection Sort

2.3.1 Introdução

O Selection Sort é um algoritmo de ordenação relativamente eficiente com número de entradas baixo, seu funcionamento consiste em consumir o vetor a ser ordenado de forma sequencial buscando pelo menor elemento do vetor e inserindo-o na posição correta.

2.3.2 Implementação

Para o algoritmo de ordenação baseado em seleção, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.2.

Algoritmo 2.2: Algoritmo de ordenação por seleção

```
SELECTION−SORT(A)
n ← len(A)
i ← 1
while i < n − 1 do
    min ← i
    j ← i + 1
    while j < n do
        if A[j] < A[min] then
            min ← j
    if i ≠ min then
        swap(A[i], A[min])
    i ← i + 1
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
void sSort(int *v, int n)
{
    int min;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (v[j] < v[min])
            {
                min = j;
            }
        }
        if (i != min)
        {
            swap(&v[i], &v[min]);
        }
    }
}
```

2.3.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação por seleção, é necessário perceber os tempos de execução de cada linha do pseudo-código 2.2. Nesse caso, pode-se ter como base a equação 2.3.

$$T(n) = C_2 + C_3 + \sum_{i=1}^{n-1}(C_4 + C_5 + C_6 + C_9 + C_{10} + C_{11} + C_{12}) + \sum_{j=1}^{n^2}(C_7 + C_8) \quad (2.3)$$

Portanto, é plausível relacionar a equação 2.3 com a equação 2.4.

$$T(n) = an^2+bn+c\, para\; a = (C_7+C_8),\; b = (C_4+C_5+C_6+C_9+C_{10}+C_{11}+C_{12})\; e\; c = (C_2+C_3) \quad (2.4)$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação por seleção:

$$O(n) = n^x \forall x \geq 2$$
$$\Omega(n) = n^x \forall x \leq 2$$
$$\Theta(n) = n^2$$

2.3.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação por seleção, pode-se observar o gráfico 2.2 que apresentam o tempo de execução para o algoritmo de ordenação por seleção para 2048 casos diferentes com o número de entradas n variando de 1 a 65536.

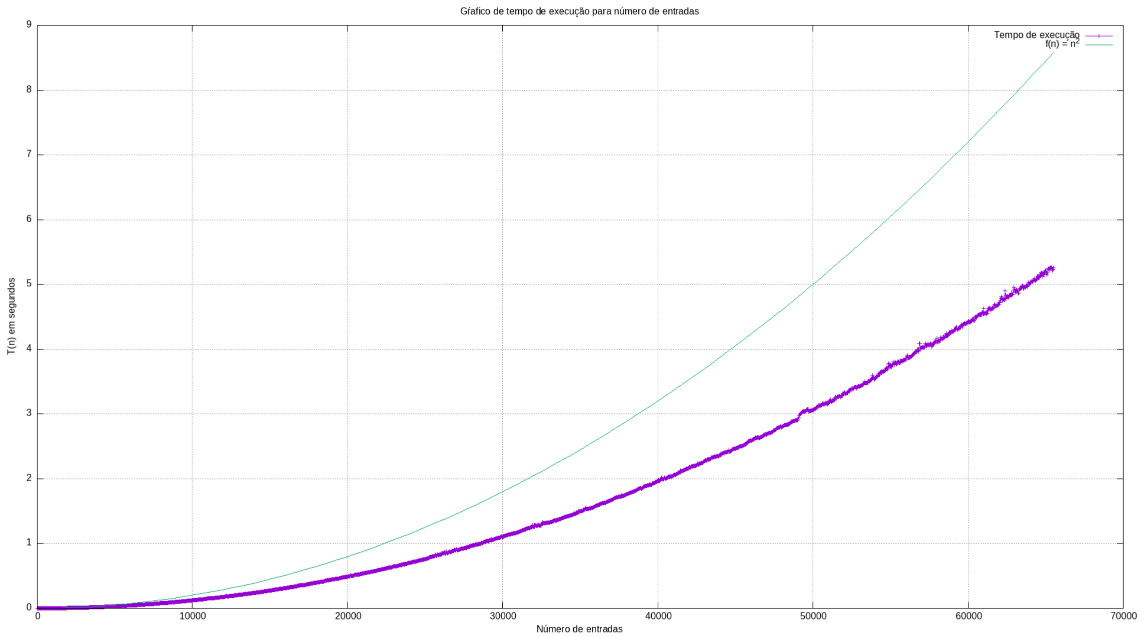

Figura 2.2: Gráfico com tempo de execução do algoritmo de ordenação por seleção

No gráfico 2.2, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função quadrática para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.2, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.3.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação por seleção é eficiente para vetores com poucas entradas a partir de um certo limiar e do contexto, existem outros algoritmos de ordenação mais eficientes que o mesmo. Sobre seu uso de memória, é constante porque utiliza apenas algumas variáveis auxiliares.

2.4 Heap Sort

2.4.1 Introdução

O *Heap* Sort é um algoritmo de ordenação bastante eficiente, seu funcionamento é baseado na estrutura de dados chamada *Heap*, que pode ser implementada utilizando um vetor que mantém as propriedades de *Heap*, seja ela a propriedade de *max-heap* ou a propriedade de *min-heap*.

- *max-heap*: é uma propriedade da estrutura de dados *Heap* que indica que a raiz ou primeiro indíce do vetor é o maior elemento da *Heap*.
- *min-heap*: é uma propriedade da estrutura de dados *Heap* que indica que a raiz ou primeiro indíce do vetor é o menor elemento da *Heap*.

2.4.2 Implementação

Para o algoritmo de ordenação baseado na estrutura de uma *Heap*, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.3.

Algoritmo 2.3: Algoritmo de ordenação utilizando uma *Heap*

```
HEAP−SORT(A)
n ← len(A)
BUILD−MAX−HEAP(A)
i ← n − 1
while i > 1 do
    swap(A[0], A[i])
    MAX−HEAPIFY(A)
    i ← i − 1
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
void heapify(int * v, int n, int i)
{

    int largest = i;

    int left = 2 * i + 1;

    int right = 2 * i + 2;

    if (left < n && v[left] > v[largest])
    {
        largest = left;
    }

    if (right < n && v[right] > v[largest])
```

```
    {
        largest = right;
    }

    if (largest != i)
    {
        swap(&v[i], &v[largest]);

        heapify(v, n, largest);
    }
}

void hSort(int * v, int n)
{
    int i;

    for (i = n / 2 - 1; i >= 0; i--)
    {
        heapify(v, n, i);
    }

    // Heap sort
    for (int i = n - 1; i > 0; i--)
    {

        swap(&v[0], &v[i]);

        heapify(v, i, 0);
    }
}
```

2.4.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação pela estrutura da *Heap*, é necessário perceber os tempos de execução de cada linha do pseudo-código 2.3. Nesse caso, pode-se ter como base a equação 2.5.

$$T(n) = C_2 + T(BUILDMAXHEAP) + C_4 + \sum_{i=2}^{n-1}(C_6 + T(MAXHEAPIFY) + C_8) \quad (2.5)$$

Sabendo que, o tempo de execução $T(BUILDMAXHEAP) = O(n)$ e o tempo de execução $T(MAXHEAPIFY) = O(\log n)$, podemos assumir a relação entre a equação 2.5 com a equação 2.6.

$$T(n) = n + an + c \; para \; a = (C_6 + \log n + C_8) \; e \; c = (C_2 + C_4) \quad (2.6)$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação pela estrutura da *Heap*:

$$O(n) = n \times \log n$$
$$\Omega(n) = n \times \log n$$
$$\Theta(n) = n \times \log n$$

2.4.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação pela estrutura da *Heap*, pode-se observar o gráfico 2.3 que apresentam o tempo de execução para o algoritmo de ordenação pela estrutura da *Heap* para 8192 casos diferentes com o número de entradas n variando de 1 a 1048576.

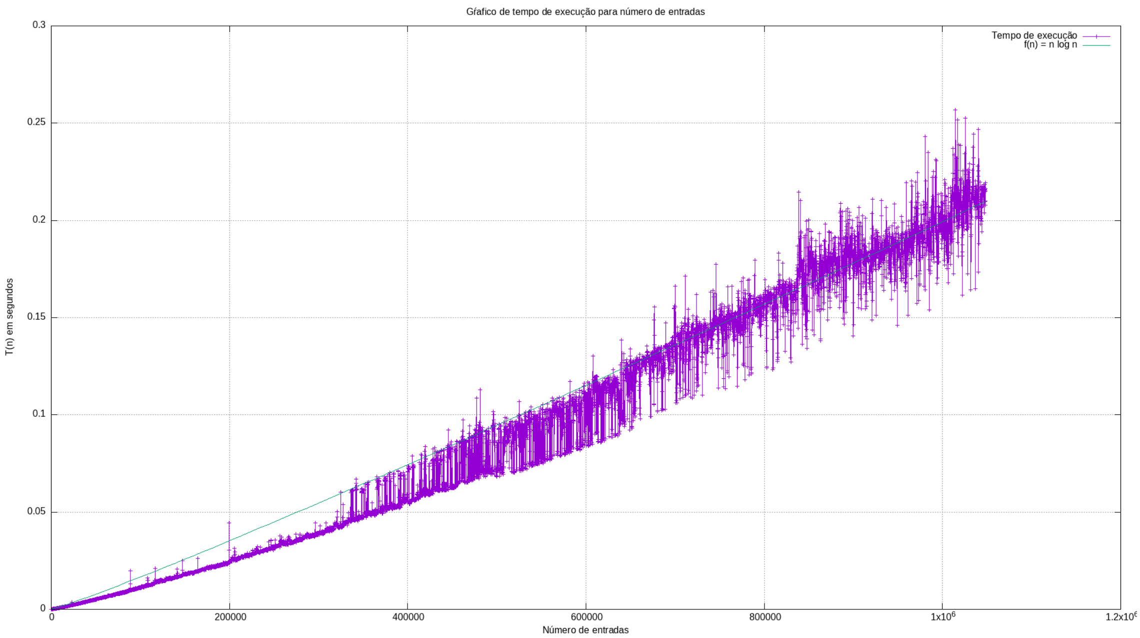

Figura 2.3: Gráfico com tempo de execução do algoritmo de ordenação pela estrutura da *Heap*

No gráfico 2.3, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função $n \times \log n$ para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.3, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.4.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação pela estrutura da *Heap* é eficiente para vetores com muitas entradas. Sobre seu uso de memória, é constante porque utiliza apenas algumas variáveis auxiliares e transfigura o vetor origem de forma direta.

2.5 Merge Sort

2.5.1 Introdução

O *Merge* Sort é um algoritmo de ordenação bastante eficiente, seu funcionamento é baseado na estratégia de ordenação de dividir para conquistar. Basicamente, o processo de ordenação consiste em separar de forma recursiva o vetor, até que contenha 1 elemento e então, realiza o procedimento *merge*, que consiste produzir um novo vetor ordenado dos elementos de mesma profundidade topológica.

2.5.2 Implementação

Para o algoritmo de ordenação por mesclagem, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.4.

Algoritmo 2.4: Algoritmo de ordenação por mesclagem

```
MERGE-SORT(A, l, r)
if l < r then
    m ← l + (r − l) / 2

    MERGE-SORT(A, l, m)
    MERGE-SORT(A, m + 1, r)

    MERGE(A, l, m, r)
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
void merge(int * v, int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = v[l + i];
    for (j = 0; j < n2; j++)
        R[j] = v[m + 1 + j];

    i = 0;

    j = 0;

    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            v[k] = L[i];
            i++;
        }
        else {
            v[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        v[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        v[k] = R[j];
        j++;
        k++;
    }
```

```
}

void mSort(int * v, int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;

        mSort(v, l, m);
        mSort(v, m + 1, r);

        merge(v, l, m, r);
    }
}
```

2.5.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação por mesclagem, é necessário perceber os tempos de execução de cada linha do pseudo-código 2.4. Nesse caso, pode-se ter como base a equação 2.7.

$$T(n) = \sum_{i=1}^{\log n} (C_2 + C_3 + T(MERGE)) \tag{2.7}$$

Sabendo que, o tempo de execução $T(MERGE) = O(n)$, podemos assumir a relação entre a equação 2.7 com a equação 2.8.

$$T(n) = a \times \log n + c \; para \; a = (C_2 + C_3 + T(MERGE)) \tag{2.8}$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação por mesclagem:

$$O(n) = n \times \log n$$
$$\Omega(n) = n \times \log n$$
$$\Theta(n) = n \times \log n$$

2.5.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação por mesclagem, pode-se observar o gráfico 2.4 que apresentam o tempo de execução para o algoritmo de ordenação por mesclagem para 8192 casos diferentes com o número de entradas n variando de 1 a 1048576.

No gráfico 2.4, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função $n \times \log n$ para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.4, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.5.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação por mesclagem é eficiente para vetores com muitas entradas. Sobre seu uso de memória, é na ordem de $S(n) = n$ porque utiliza vetores auxiliares que crescem de forma linear com n.

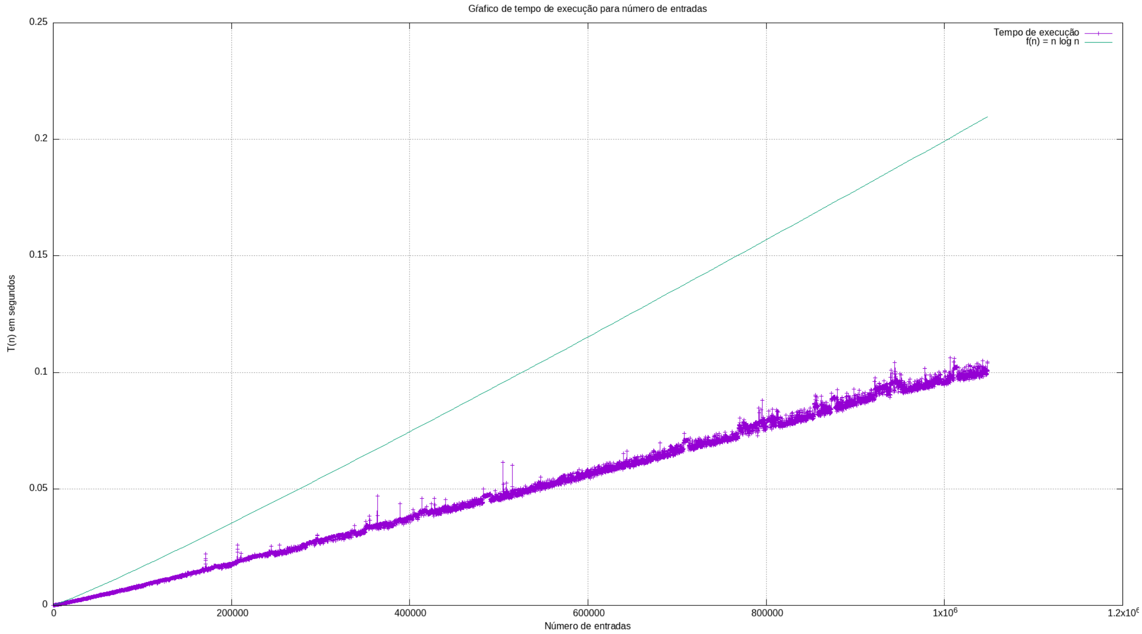

Figura 2.4: Gráfico com tempo de execução do algoritmo de ordenação por mesclagem

2.6 Quick Sort

2.6.1 Introdução

O *Quick* Sort é um algoritmo de ordenação bastante eficiente, seu funcionamento é baseado na estratégia de ordenação de dividir para conquistar. Basicamente, o processo é particionar um vetor de forma recursiva enquanto realiza a ordenação nele, utilizando uma lógica de pivot e marcador para troca.

2.6.2 Implementação

Para o algoritmo de ordenação por particionamento, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.5 retirado do livro Cormen et al. (2022).

Algoritmo 2.5: Algoritmo de ordenação por particionamento

```
QUICK-SORT(A, p, r)
if p < r then
    q = PARTITION(A, p, r)
    QUICK-SORT(A, p, q − 1)
    QUICK-SORT(A, q + 1, r)
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
int partition(int * v, int s, int e)
{

    int pivot = v[s];

    int i = s - 1;
    int j = e + 1;
```

```

    while (1)
    {
        do
        {
            i += 1;
        } while (v[i] < pivot);

        do
        {
            j -= 1;
        } while (v[j] > pivot);

        if (i >= j)
        {
            return j;
        }

        swap(&v[i], &v[j]);
    }
}

void qSort(int * v, int s, int e)
{
    if (s >= 0 && e >= 0 && s < e)
    {
        int p = partition(v, s, e);
        qSort(v, s, p);
        qSort(v, p + 1, e);
    }
}
```

2.6.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação por particionamento, é necessário perceber os tempos de execução de cada linha do pseudocódigo 2.5. Nesse caso, pode-se ter como base a equação 2.9.

$$T(n) = \sum_{i=1}^{\log n} (C_2 + T(PARTITION)) \tag{2.9}$$

Sabendo que, o tempo de execução $T(PARTITION) = O(n)$, podemos assumir a relação entre a equação 2.9 com a equação 2.10.

$$T(n) = a \times \log n + c \; para \; a = (C_2 + T(PARTITION)) \tag{2.10}$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação por particionamento:

$$O(n) = n \times \log n$$
$$\Omega(n) = n \times \log n$$
$$\Theta(n) = n \times \log n$$

2.6.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação por particionamento, pode-se observar o gráfico 2.5 que apresentam o tempo de execução para o algoritmo de ordenação por particionamento para 8192 casos diferentes com o número de entradas n variando de 1 a 1048576.

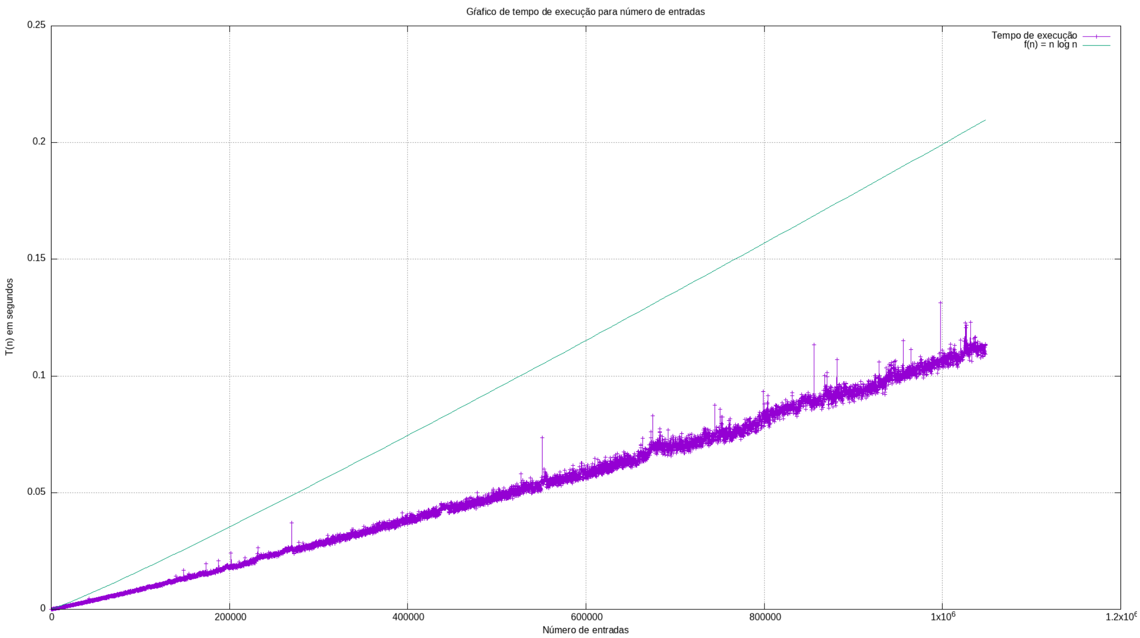

Figura 2.5: Gráfico com tempo de execução do algoritmo de ordenação por particionamento

No gráfico 2.5, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função $n \times \log n$ para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.5, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.6.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação por particionamento é eficiente para vetores com muitas entradas. Sobre seu uso de memória, é constante porque utiliza apenas algumas variáveis auxiliares e transfigura o vetor origem de forma direta.

2.7 Counting Sort

2.7.1 Introdução

O *Counting* Sort é um algoritmo de ordenação bastante eficiente dada a uma quantidade de memória equivalente as necessidades. Seu funcionamento é baseado num método de contagem, iterando pelo vetor origem e contando as aparições dos elementos num vetor auxiliar K.

Uma das características desse algoritmo é, como será apresentado, seu tempo de execução linear mas sua grande utilização de memória a depender da necessidade. Sendo, seu uso de memória a equação 2.11.

$$M(n) = n + k \text{ } para \text{ } k = |\min n| + |\max n| \tag{2.11}$$

2.7.2 Implementação

Para o algoritmo de ordenação por contagem, o pseudo-código utilizado para desenvolver o algoritmo pode ser observado em 2.6 retirado do livro Cormen et al. (2022).

Algoritmo 2.6: Algoritmo de ordenação por contagem

```
COUNTING-SORT(A, n, K)
let B[1:n] and C[0:K] be new arrays
for i ← 0 to k
    C[i] ← 0
for j ← 1 to n
    C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
    C[i] = C[i] + C[i − 1]
for j ← n downto 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
return B
```

Esse pseudo-código foi implementado na linguagem de programação C e pode ser observado no código seguinte:

```c
void cSort(unsigned char * v, int n)
{
    int i, j;

    int size = __UINT8_MAX__;

    unsigned char B[n];
    int C[size + 1];

    for (i = 0; i < size + 1; i++)
    {
        C[i] = 0;
    }

    for (j = 0; j < n; j++)
    {
        C[v[j]]++;
    }

    for (i = 1; i <= size; i++)
    {
        C[i] = C[i] + C[i - 1];
    }

    for (j = n - 1; j >= 0; j--)
    {
        B[C[v[j]] - 1] = v[j];
        C[v[j]]--;
    }

    for (i = 0; i < n; i++)
    {
        v[i] = B[i];
```

```
    }

}
```

Essa implementação do algoritmo em C, leva em consideração que o vetor a ser ordenado é de caracteres, portanto, o vetor auxiliar K sempre será igual a 256, por ser o máximo possível de distância entre o mínimo e o máximo do vetor origem.

2.7.3 Análise do algoritmo e notação assintótica

Para que seja determinada a razão de crescimento do algoritmo de ordenação por contagem, é necessário perceber os tempos de execução de cada linha do pseudo-código 2.6. Nesse caso, pode-se ter como base a equação 2.12.

$$T(n) = C_2 + C_{14} + \sum_{i=1}^{n}(C_5 + C_{12} + C_{13}) + \sum_{i=1}^{k}(C_3 + C_8) \tag{2.12}$$

Podemos assumir a relação entre a equação 2.12 com a equação 2.13.

$$T(n) = an + bk + c \; para \; a = (C_5 + C_{12} + C_{13}) \; , b = (C_3 + C_8) \; e \; c = (C_1 + C_{14}) \tag{2.13}$$

Com isso, pode-se determinar as seguintes notações assintóticas para o algoritmo de ordenação por contagem:

$$O(n) = n + k$$
$$\Omega(n) = n + k$$
$$\Theta(n) = n + k$$

2.7.4 Comparação teórica-prática

Para melhor compreensão do tempo de execução do algoritmo de ordenação por contagem, pode-se observar o gráfico 2.6 que apresentam o tempo de execução para o algoritmo de ordenação por contagem para 8192 casos diferentes com o número de entradas n variando de 1 a 1048576.

No gráfico 2.6, é possível perceber que o tempo de execução do algoritmo se aproxima da função $f(n)$ que é uma função linear para n com uma redução de escala para melhor percepção e comparação. Então, utilizando como base o gráfico 2.6, pode-se confirmar que a ordem de crescimento determinada é precisa.

2.7.5 Discussão sobre tempo de execução e uso de memória

Sobre seu tempo de execução, o algoritmo de ordenação por contagem é eficiente para vetores com muitas entradas desde que exista memória suficiente. Sobre seu uso de memória, é $S(n) = K$. Como apontado previamente, se por definição, a distância entre o mínimo e o máximo do vetor origem for relativamente pequena comparada a memória, é um dos melhores algoritmos disponíveis para utilização.

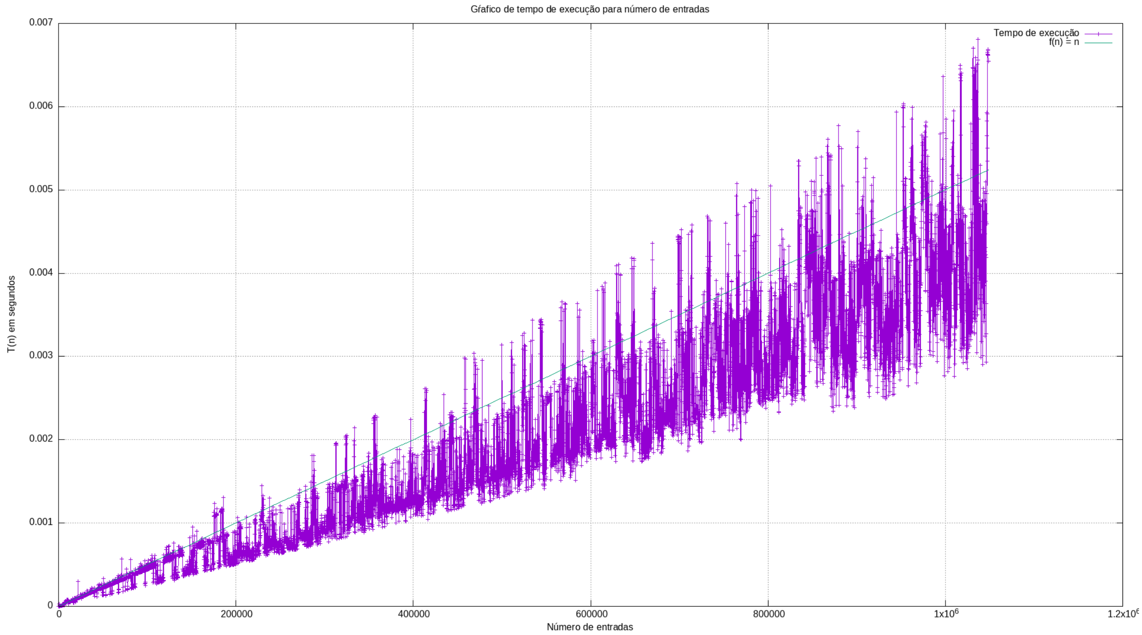

Figura 2.6: Gráfico com tempo de execução do algoritmo de ordenação por contagem

2.8 Comparações

Com as explicações para os algoritmos nas seções (2.2, 2.3 2.4, 2.5, 2.6, 2.7), pode-se racionalizar alguns pensamentos entre os algoritmos. Primeiro, se memória não for um fator relevante, portanto, ou existe muita memória no computador ou K é uma distância muito pequena, o algoritmo apresentado na seção 2.7 é o melhor dos apresentados por executar em tempo linear. Se memória for um fator, dentre os algoritmos da ordem $\Theta(n) = n \log n$, provavelmente quem se sobressai é o algoritmo baseado em particionamento apresentado na seção 2.6 porque não utiliza memória auxiliar linear como o 2.5 e apresenta um comportamento mais estável que o 2.4 como pode-se observar no gráfico 2.7. Já, se o número de entradas for quase irrelevante, pode-se aplicar um dos algoritmos de ordem $\Theta(n) = n^2$ como os apresentados nas seções (2.2, 2.3).

Na tabela 2.1, podemos ver de forma explícita, os tempos de execução e uso de memória para cada algoritmo.

Tabela 2.1: Comparação de algoritmos

Algoritmo	**T(n)**	**S(n)**
Counting	$\Theta(n)$	$\Theta(k)$
Quick	$\Theta(n \log n)$	$\Theta(1)$
Merge	$\Theta(n \log n)$	$\Theta(n)$
Heap	$\Theta(n \log n)$	$\Theta(1)$
Insertion	$\Theta(n^2)$	$\Theta(1)$
Selection	$\Theta(n^2)$	$\Theta(1)$

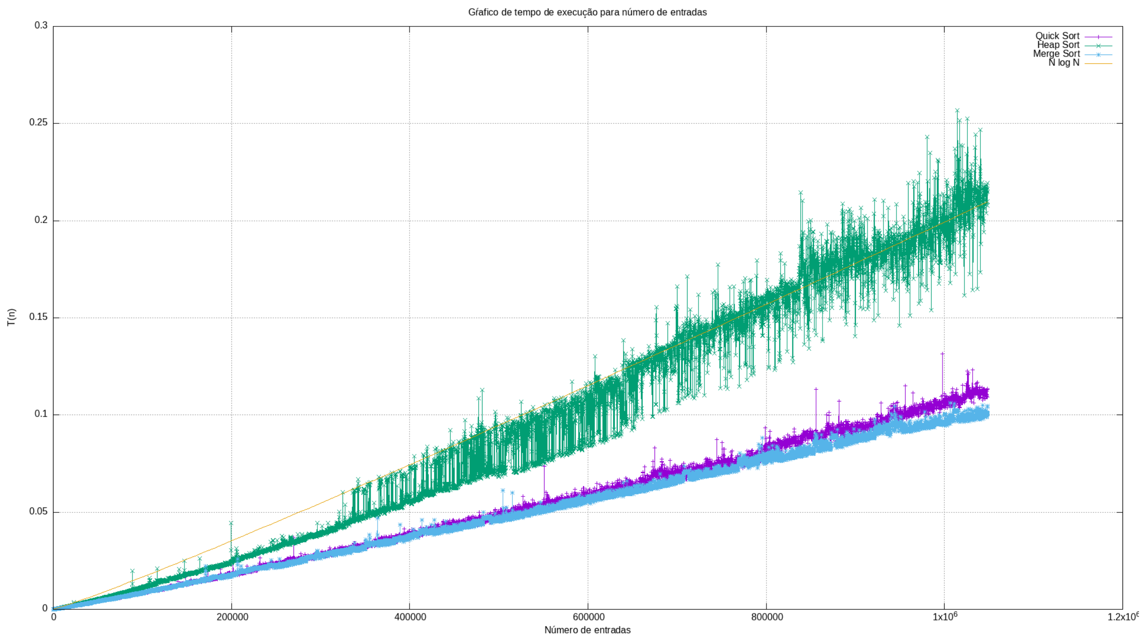

Figura 2.7: Gráfico com tempo de execução dos algoritmos n log n

Capítulo 3

Análise de algoritmos de árvores e tabelas de dispersão

"The greatest teacher, failure is."
Yoda

Capítulo 4

Conclusões

"These petals... so white. Can you see them too... watching me from wherever you are?
Perhaps dreams aren't such bad things after all..."
Elderbug

Referências Bibliográficas

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest e Clifford Stein (2022), *Introduction to Algorithms*, 4ª edição, The MIT Press.
(Citado nas páginas 1, 2, 3, 15 e 18)

Karumanchi, Narasimha (2017), *Data Structures and Algorithms Made Easy*, 5ª edição, CareerMonk.
(Citado na página 2)