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

Atividade Prática I

Dayanne Xavier Lucena

Orientador: Prof. Dr. João Paulo De Souza Medeiros

Relatório apresentado ao Curso de Bacharelado em Sistemas de Informação como parte avaliativa da matéria de Estrutura de Dados.

Caicó, RN, 28 de maio de 2023

Sumário

1. Introdução

A eficiência na ordenação de dados é um desafio fundamental na nossa área de atuação. Com a crescente disponibilidade de enormes quantidades de dados armazenados, a necessidade de algoritmos de ordenação rápidos e eficientes tornou-se cada vez mais crucial.

Neste relatório, faremos uma análise e comparação dos diferentes algoritmos de ordenação apresentados pelo professor na matéria de Estrutura de Dados, a fim de mostrar sua complexidade, desempenho e características distintas. Os algoritmos que serão examinados incluem: selection-sort(), insertion-sort(), merge-sort(), quick-sort() e distribution-sort().

Ao longo deste relatório, iremos explorar os princípios básicos desses algoritmos, discutindo suas vantagens e desvantagens em termos de tempo de execução, além de observar seu comportamento em diferentes cenários, considerando casos de melhor e pior desempenho. Para melhor visualização e compreensão, utilizaremos gráficos comparativos.

2. Códigos dos algoritmos

Todos os códigos feitos nesse trabalho foram baseados nos pseudocódigos que se encontram no GitHub do professor, sendo assim, foram escritos seus respectivos códigos usando a linguagem C, fazendo mudanças necessárias para que rodasse na linguagem usada , como declarar as variáveis usadas, todas elas com seus tipos correspondentes, e também precisamos considerar que o vetor na linguagem que usamos começa do 0, entre outras coisas.

2.1 Selection-sort()

Na figura 2.1 se encontra o Pseudocódigo do selection, e na figura 2.1 temos o código feito em C. As principais mudanças entre os dois:

1 - Declarar as variáveis, em C só compila se todas elas tiverem sido declaradas.

2 - O primeiro for começa de 0, pois o vetor em C começa em 0.

3- O primeiro for vai até menor que n - 1; no pseudocódigo ele ia até n - 1, ou seja até a penúltima posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua penúltima posição será n - 2.

4- O segundo for vai até menor que n - 1 no pseudocódigo ele ia até n, ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1 .

5- Swap foi trocada por uma subistituição direta no código para melhor visualização.

```
algorithm selection-sort(v, n)
|     for i from 1 to (n − 1) do
|     |     m ← i
|     |     for j from (i + 1) to n do
|     |     |     if v[m] > v[j] then
|     |     |     |     m ← j
|     |     swap(v[m], v[i])
```

Algoritmo 2.1: Pseudocódigo Selection().

```
void selection_sort(int *v, unsigned int n) {
    unsigned int i, j;
    int m;
    int aux;

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

        for (j = (i + 1); j < n; j++) {
            if (v[m] > v[j]) {
                m = j;
            }
        }

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

Figura 2.1: Código em C: Selection().

2.2 Insertion-sort()

Na figura 2.2 se encontra o Pseudocódigo do selection, e na figura 2.2 temos o código feito em C. As principais mudanças entre os dois:

1 - Declarar as variáveis, em C só compila se todas elas tiverem sido declaradas.

2 - O primeiro for começa de 1, no pseudocódigo ele começa de 2, que seria a segunda posição do vetor, a segunda posição de um vetor em C é 1.

3 - O primeiro for vai até menor que n - 1 no pseudocódigo ele ia até n, ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1.

4 - O while começa de 0, no pseudocódigo ele começa de 1, que seria a primeira posição do vetor, a primeira posição de um vetor em C é 0.

5 - Swap foi trocada por uma subistituição direta no código para melhor visualização.

```
void insertion_sort(int *v, unsigned int n) {
    unsigned int i, e;
    int aux;

    for (e = 1; e < n; e++) {
        i = e;

        while ((i > 0) && (v[i - 1] > v[i])) {
            aux = v[i - 1];
            v[i - 1] = v[i];
            v[i] = aux;

            i = i - 1;

        }


    }
}
```

Figura 2.2: Código em C: Insertion().

algorithm insertion-sort(v, n)
| for e from 2 to n do
| | $i \leftarrow e$
| | while $i > 1$ and $v[i-1] > v[i]$ do
| | | swap($v[i-1], v[i]$)
| | | $i \leftarrow i - 1$

Algoritmo 2.2: Pseudocódigo Insertion().

2.3 Merge-sort()

Na figura 2.3 se encontra o Pseudocódigo do selection, e na figura 2.3 temos o código feito em C. As principais mudanças entre os dois:

1 - Declarar as variáveis, em C só compila se todas elas tiverem sido declaradas.

2 - Alocar o tamanho do vetor auxiliar, se não criar o vetor, não dá para usar.
3 - O primeiro for começa de 0, no pseudocódigo ele começa de 1, que seria a primeira posição do vetor, a primeira posição de um vetor em C é 0.

4 - O primeiro for vai até menor que n (e - s + 1); no pseudocódigo ele ia até n (e - s + 1), ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1.

5 - O segundo for começa de 0, no pseudocódigo ele começa de 1, que seria a primeira posição do vetor, a primeira posição de um vetor em C é 0.

6 - O segundo for vai até menor que n (e - s + 1); no pseudocódigo ele ia até n (e - s + 1), ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1.

7 - Libera a memória do vetor que foi alocado, é importante fazer isso em C porque caso não faça a memória ficará bloqueada até o programa encerrar.

```
algorithm merge-sort(v, s, e)
|     if s < e then
|     |     m ← ⌊(s + e)/2⌋
|     |     merge-sort(v, s, m)
|     |     merge-sort(v, m + 1, e)
|     |     merge(v, s, m, e)

algorithm merge(v, s, m, e)
|     p ← s
|     q ← m + 1
|     for i from 1 to (e − s + 1) do
|     |     if (q > e) or ((p ≤ m) and (v[p] < v[q])) then
|     |     |     w[i] ← v[p]
|     |     |     p ← p + 1
|     |     else
|     |     |     w[i] ← v[q]
|     |     |     q ← q + 1
|     for i from 1 to (e − s + 1) do
|     |     v[s + i − 1] ← w[i]
```

Algoritmo 2.3: Pseudocódigo Merge().

2.4 Quick-sort()

Na figura 2.4 se encontra o Pseudocódigo do selection, e na figura 2.4 temos o código feito em C. As principais mudanças entre os dois:

1 - Declarar as variáveis, em C só compila se todas elas tiverem sido declaradas.

2 - No pseudocódigo d começa sendo "d = s - 1", escolhi começar com "d = s".

3 - Como fiz a mudança de "d = s - 1"para "d = s"primeiro troco as posições de lugar e depois incremento mais um no d.

4 - Como fiz a mudança de "d = s - 1"para "d = s"quando o for terminar de rodar o valor de "d"já irá corresponder ao índice do pivô, sendo assim ele irá para seu lugar correspondente ao fazer a troca, e irá retorna seu índice "d".

```
1  void merge_sort(int *v, unsigned int s, unsigned int e) {
2      unsigned int m;
3
4      if (s < e) {
5
6          m = ((s + e) / 2);
7
8          merge_sort(v, s, m);
9          merge_sort(v, m + 1, e);
10         merge(v, s, m, e);
11
12     }
13
14 }
15
16 void merge(int *v, unsigned int s, unsigned int m, unsigned int e)
       {
17
18     unsigned int p = s;
19     unsigned int q = m + 1;
20
21     int *w = (int *) malloc((e - s + 1) * sizeof(int));
22
23     for (int i = 0; i < (e - s + 1); i++) {
24         if ((q > e) || ((p <= m) && v[p] < v[q])) {
25             w[i] = v[p];
26             p = p + 1;
27
28         }
29
30         else {
31             w[i] = v[q];
32             q = q + 1;
33
34         }
35
36     }
37
38     for (int i = 0; i < (e - s + 1); i++) {
39         v[s + i] = w[i];
40
41     }
42
43     free(w);
44
45 }
```

Figura 2.3: Código em C: Merge().

```
algorithm quick-sort(v, s, e)
|     if s < e then
|     |     p ← partition(v, s, e)
|     |     quick-sort(v, s, p − 1)
|     |     quick-sort(v, p + 1, e)

algorithm partition(v, s, e)
|     d ← s − 1
|     for i from s to (e − 1) do
|     |     if v[i] ≤ v[e] then
|     |     |     d ← d + 1
|     |     |     swap(v[d], v[i])
|     swap(v[d + 1], v[e])
|     return (d + 1)
```

Algoritmo 2.4: Pseudocódigo Quick().

2.5 Distribution-sort()

Na figura 2.5 se encontra o Pseudocódigo do selection, e na figura 2.5 temos o código feito em C. As principais mudanças entre os dois:

1 - Declarar as variáveis, em C só compila se todas elas tiverem sido declaradas.

2 - Alocar o tamanho dos vetores auxiliares, se não criar o vetor, não dá para usar.
3 - O primeiro, o segundo, o terceiro e o quarto for começam de 0, no pseudocódigo ele começa de 1, que seria a primeira posição do vetor, a primeira posição de um vetor em C é 0.

4 - O terceiro for começa de 1, no pseudocódigo ele começa de 2, que seria a segunda posição do vetor, a segunda posição de um vetor em C é 1.

6 - O primeiro e o terceiro for vai até menor que n (b - s + 2); no pseudocódigo ele ia até n (b - s + 1), ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1, usamos (b - s + 2) porque foi o tamanho do vetor alocado.

7 - O segundo, o quarto e o quinto for vai até menor que n, no pseudocódigo ele ia até n, ou seja até a última posição do vetor de tamanho n, mas em C o vetor de tamaho n só tem de 0 posições até n - 1, sua última posição será n - 1.

8 - No quarto for ao ordenar os elementos no vetor w se fez necessário colocar o -1"para acessar corretamente o endereço de memória correspondente.

9 - Libera a memória dos vetoesr que foram alocados, é importante fazer isso em C porque caso não faça a memória ficará bloqueada até o programa encerrar

```
1  void quick_sort(int *v, int s, int e) {
2      unsigned int p;
3
4      if (s < e) {
5          p = partition(v, s, e);
6
7          quick_sort(v, s, p - 1);
8          quick_sort(v, p + 1, e);
9
10     }
11 }
12
13 unsigned int partition(int *v, unsigned int s, unsigned int e) {
14     unsigned int d = s;
15
16     for (int i = s; i <= (e - 1); i++) {
17         if (v[i] <= v[e]) {
18
19             swap(&v[d], &v[i]);
20
21             d = d + 1;
22
23         }
24
25     }
26
27     swap(&v[d], &v[e]);
28
29     return d;
30
31 }
```

Figura 2.4: Código em C: Quick().

```
algorithm distribution-sort(v, n)
|     s ← min(v, n)
|     b ← max(v, n)
|     for i from 1 to (b − s + 1) do
|     |     c[i] ← 0
|     for i from 1 to n do
|     |     c[v[i] − s + 1] ← c[v[i] − s + 1] + 1
|     for i from 2 to (b − s + 1) do
|     |     c[i] ← c[i] + c[i − 1]
|     for i from 1 to n do
|     |     d ← v[i] − s + 1
|     |     w[c[d]] ← v[i]
|     |     c[d] ← c[d] − 1
|     for i from 1 to n do
|     |     v[i] ← w[i]
```

Algoritmo 2.5: Pseudocódigo Distribution().

```
void distribution_sort(int *v, unsigned int n) {
    int s = min(v, n);
    int b = max(v, n);
    int d;

    int *c = (int *) malloc((b - s + 2) * sizeof(int));
    int *w = (int *) malloc(n * sizeof(int));

    for (int i = 0; i < (b - s + 2); i++) {
        c[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        c[v[i] - s + 1] = c[v[i] - s + 1] + 1;
    }

    for (int i = 1; i < (b - s + 2); i++) {
        c[i] = c[i] + c[i - 1];
    }

    for (int i = 0; i < n; i++) {
        d = v[i] - s + 1;
        w[c[d] - 1] = v[i];
        c[d] = c[d] - 1;
    }

    for (int i = 0; i < n; i++) {
        v[i] = w[i];
    }

    free(c);
    free(w);
}
```

Figura 2.5: Código em C: Distribution().

3. Tempo de execução

3.1 Selection-sort()

O algoritmo de ordenação Selection Sort (ou ordenação por seleção) é um método simples e intuitivo para ordenar um conjunto de dados. Sua abordagem consiste em encontrar repetidamente o menor elemento não ordenado e colocá-lo na posição correta no início da sequência ordenada. Ele é um algoritmo in-place, ou seja, ele ordena no própio vetor, tendo um uso de memória constante em relação ao vetor. (Veja a figura 2.1)

O tempo de execução do Selection Sort é de ordem $\Theta(n^2)$, esse comportamento se dará pelo fato da existência de um for dentro de outro, que sempre irão ser executados, não existindo nada que o faça ter um melhor ou pior caso. (Veja a equação 3.1)

$$T(n) = c_2 + c_3 + c_4 + c_6 * n + c_7(n-1) + \sum_{i=1}^{n}(i * c_9) + \sum_{i=1}^{n-1}(i * c_{10}) \tag{3.1}$$

Observe que não coloquei a linha 9, isso se dá pelo motivo que essa linha pode ou não ser executada de acordo como resultado do "if"antes dela, mas ela não irá interferir na ordem do nosso algoritmo.

Resolvendo o primeiro somatório teremos:
1ºTiramos a constante de dentro do somatório, isso é possível por causa da operação de multiplicação, ela se encontra em evidência com todos os termos.

$$\sum_{i=1}^{n}(i * c_9) = c_9 * \sum_{i=1}^{n}(i) \tag{3.2}$$

2ºO resultado do somatório que possui só o "i"é:

$$\sum_{i=1}^{n}(i) = (n+1) * \frac{n}{2} \tag{3.3}$$

3ºQue podemos reescrever como:

$$\frac{n^2+n}{2} \tag{3.4}$$

4ºNão esqueça da constante que estava acompanhando:

$$c_9 * (\frac{n^2+n}{2}) \tag{3.5}$$

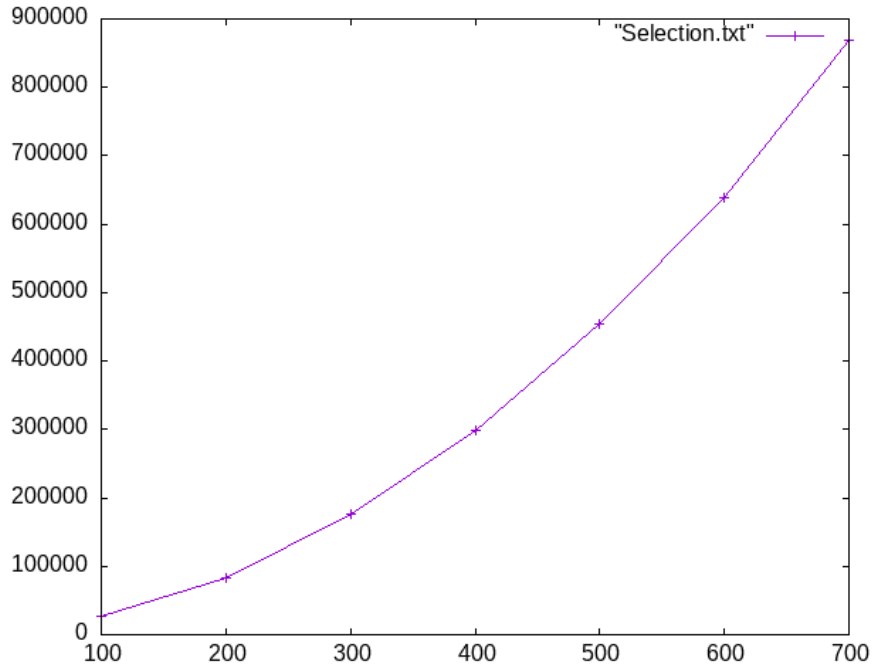

Figura 3.1: Tempo de execução do Selection Sort: $\Theta(n^2)$ (tamanho do vetor x tempo) .

Para o segundo somatório se aplica a mesma lógica, mas como ele vai até n - 1, teremos que tirar o n da nossa soma de termos, então:
1ºSubtraimos n dos dois lados para manter a igualdade, assim podemos continuar normalmente:

$$\sum_{i=1}^{n-1}(i) - n = ((n+1) * \frac{n}{2}) - n \tag{3.6}$$

2ºEntão adicionamos a constante:

$$c_{10} * (\frac{n^2+n}{2} - n) \tag{3.7}$$

Logo podemos reescrever nossa equação geral como: (Veja equação 3.8)

$$T(n) = c_2 + c_3 + c_4 + c_6 * n + c_7(n-1) + c_9 * (\frac{n^2+n}{2}) + c_{10} * (\frac{n^2+n}{2} - n) \tag{3.8}$$

Sendo assim podemos concluir que o tempo de execução do Selection Sort é de ordem $\Theta(n^2)$. Isso também pode ser observado no gráfico (Veja a figura 3.1) feito a partir de simulações produzidas pelo código apresentado anteriormente.

3.2 Insertion-sort()

O Insertion Sort é um algoritmo de ordenação simples que percorre o vetor a ser ordenado e insere cada elemento na posição correta dentro da porção já ordenada do vetor. Ele pecorre o vetor e a cada iteração, o elemento atual é comparado com os elementos

anteriores na parte ordenada do vetor. O objetivo é encontrar a posição correta onde o elemento deve ser inserido. Ele é um algoritmo in-place. (Veja a figura 2.2)

O tempo de execução do Insertion Sort diferentemente do selection possui melhor caso, pior caso, e o caso esperado, isso irá depender de como se encontra o vetor que ele recebe. No pior caso (o vetor estiver ordenado de forma decrescente) e no caso esperado (vetor aleatório) ele é de ordem $\Theta(n^2)$ que ocorre quando ele precisa entrar nos dois laços de repetição, mas no melhor caso (o vetor já estiver ordenado) ele nunca entrará no while, tendo assim um comportamento linear, sendo de ordem $\Theta(n)$.

3.2.1 Melhor caso Insertion-sort()

Essa será a equação quando o código se excaixar no melhor caso: (Veja figura 3.9)

$$T(n) = c_2 + c_3 + c_5 * n + (c_6 + c_8) * (n - 1) \tag{3.9}$$

Sendo assim podemos concluir que o tempo de execução do melhor caso do Insertion Sort é de ordem $\Theta(n)$.

3.2.2 Pior caso Insertion-sort()

Essa será a equação quando o código se excaixar no pior caso: (Veja figura 3.10)

$$T(n) = c_2 + c_3 + c_5 * n + (c_6) * (n - 1) + \sum_{i=1}^{n} i * c_8 + \sum_{i=1}^{n-1} i * (c_9 + c_{10} + c_{11} + c_{13}) \tag{3.10}$$

Desenvolvendo os somatórios:

$$\sum_{i=1}^{n} i * c_8 \tag{3.11}$$

$$c_8 * \frac{n^2 + n}{2} \tag{3.12}$$

$$\sum_{i=1}^{n-1} i * c_9 + c_{10} + c_{11} + c_{13}) \tag{3.13}$$

$$(c_9 + c_{10} + c_{11} + c_{13}) * (\frac{n^2 + n}{2} - n) \tag{3.14}$$

Juntando tudo teremos:

$$T(n) = c_2 + c_3 + c_5 * n + (c_6) * (n-1) + c_8 * \frac{n^2 + n}{2} + (c_9 + c_{10} + c_{11} + c_{13}) * (\frac{n^2 + n}{2} - n) \tag{3.15}$$

Sendo assim podemos concluir que o tempo de execução do melhor caso do Insertion Sort é de ordem $\Theta(n^2)$.

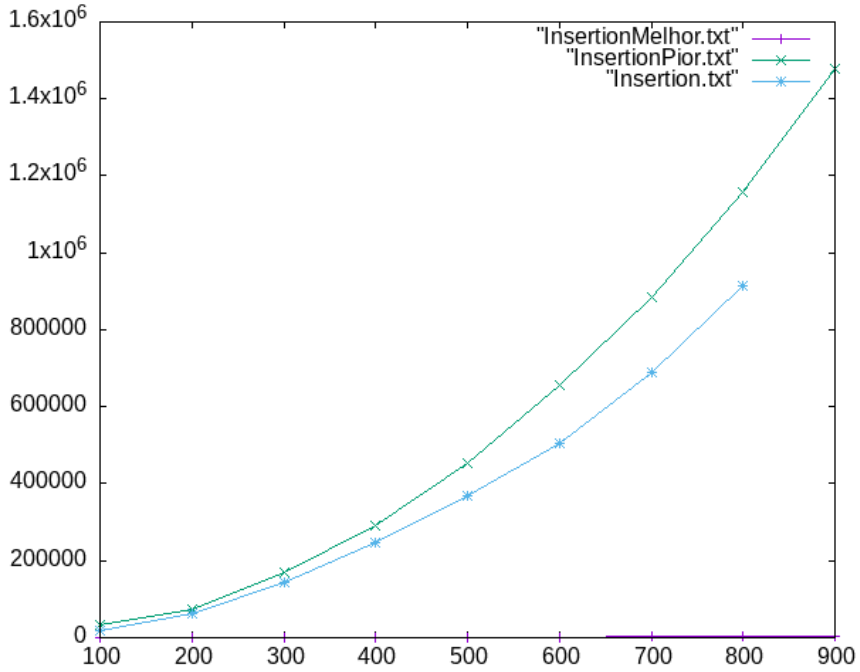

Figura 3.2: Tempo de execução do Insertion Sort no caso esperado ($\Theta(n^2)$), melhor ($\Theta(n)$) e pior caso ($\Theta(n^2)$). (tamanho do vetor x tempo) .

3.2.3 Caso esperado Insertion-sort()

O caso esperado do insertion sort estará mais perto do seu pior caso, sendo $\Theta(n^2)$, isso irá acontecer pois a maioria das vezes será preciso pecorrer os dois laços de repetição. Confira na figura 3.2, o crescimento do caso esperado e do pior é tão grande que o tempo do melhor caso parece constante.

3.3 Merge-sort()

O Merge Sort é um algoritmo de ordenação que segue o paradigma "Dividir para Conquistar". Ele divide a lista em sublistas menores de forma recursiva e ordena essas sublistas, em seguida combina as sublistas ordenadas para obter a lista final ordenada. Ele não é um algoritmo in-place, pois ele cria um vetor auxiliar para ordenar a parte do vetor, e depois sobrescreve no vetor original, não tendo uso de memória constante. (Veja a figura 2.3)

O tempo de execução do Merge Sort é de $\Theta(n * \log_2 n)$ sendo um pouco pior que o linear, e melhor que o quadrático. Isso se deve ao fato dele usar a estratégia de dividir para conquistar, ou seja, ele vai dividindo o vetor várias vezes até ficarem só dois ou um, então os ordena. Primeiro calcularemos o tempo de execução do merge() (Veja a equação 3.16).

$$T(n) = c_3 + c_4 + c_6 + c_8(n+1) + c_9 * n + c_{1016} * n + c_{1117} * n + c_{23}(n+1) + c_{24} * n + c_{28} \quad (3.16)$$

Podemos observar que ele será linear, então poderemos reescrever como an + b. Sabendo disso poderemos prosseguir e desenvolver o do merge-sort(). Vamos primeiro ver qual será o caso base para nosso algortimo, o caso base acontece quando não ocorre a recursão, ou seja, a função não chama ele mesmo, isso acontecerá quando nosso vetor só tiver uma posição (Veja a equação 3.17).

$$T(1) = c_{33} + c_{35} \tag{3.17}$$

Então vamos analisar o caso quando existe a recursão, analisando a recorrência. (Veja a equação 3.18)

$$T(n) = c_{33} + c_{35} + c_{37} + 2 * T^{ms}(\frac{n}{2}) + T^{m}(n) \tag{3.18}$$

Observe que como existe duas chamadas recursivas, sendo uma após a outra, temos o número dois multiplicando o termo recursivo do merge-sort, e também precisamos considerar o valor do tempo de execução do merge que foi calculado anteriormente (Veja a equação 3.16). Agora iremos observar o comportamento da recorrência para achar um padrão. Para melhor visualização considere o que é mostrado na equação 3.19.

$$c_{33} + c_{35} + c_{37} = c \tag{3.19}$$

1ºVamos substituir "n"por "n/2"na equação 3.18:

$$T(\frac{n}{2}) = a + 2 * T^{ms}(\frac{n}{4}) + T^{m}(\frac{n}{2}) \tag{3.20}$$

2ºSubstituindo na equação 3.18 teremos:

$$T(n) = a + 2 * [a + 2 * T^{ms}(\frac{n}{4}) + T^{m}(\frac{n}{2})] + T^{m}(n) \tag{3.21}$$

Desenvolvendo:

$$T(n) = 3a + 4 * T^{ms}(\frac{n}{4}) + 2 * T^{m}(\frac{n}{2}) + T^{m}(n) \tag{3.22}$$

3ºVamos substituir "n"por "n/4"na equação 3.18:

$$T(\frac{n}{4}) = a + 2 * T^{ms}(\frac{n}{8}) + T^{m}(\frac{n}{4}) \tag{3.23}$$

4ºSubstituindo na equação 3.22 teremos:

$$T(n) = 3a + 4 * [a + 2 * T^{ms}(\frac{n}{8}) + T^{m}(\frac{n}{4})] + 2 * T^{m}(\frac{n}{2}) + T^{m}(n) \tag{3.24}$$

Desenvolvendo:

$$T(n) = 7a + 8 * T^{ms}(\frac{n}{8}) + 4 * T^{m}(\frac{n}{4}) + 2 * T^{m}(\frac{n}{2}) + T^{m}(n) \tag{3.25}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = (2^{x} - 1) * a + 2^{x} * T^{ms}(\frac{n}{2^{x}}) + 2^{x-1} * T^{m}(\frac{n}{2^{x-1}}) + 2^{x-2} * T^{m}(\frac{n}{2^{x-2}}) + ... + 2^{0} * T^{m}(\frac{n}{2^{0}}) \tag{3.26}$$

$$T(n) = (2^x - 1) * a + 2^x * T^{ms}(\frac{n}{2^x}) + \sum_{i=0}^{x-1} 2^i * T^m(\frac{n}{2^i}) \tag{3.27}$$

Agora vamos igualar a recorrência ao caso base, então teremos:

$$\frac{n}{2^x} = 1 \tag{3.28}$$

$$2^x = n \tag{3.29}$$

$$\log_2 2^x = \log_2 n \tag{3.30}$$

$$x = \log_2 n \tag{3.31}$$

Substituindo na equação 3.27 teremos:

$$T(n) = (2^{\log_2 n} - 1) * a + 2^{\log_2 n} * T^{ms}(\frac{n}{2^{\log_2 n}}) + \sum_{i=0}^{(\log_2 n)-1} 2^i * T^m(\frac{n}{2^i}) \tag{3.32}$$

Simplificando teremos:

$$T(n) = (n-1) * a + n * T^{ms}(1) + \sum_{i=0}^{(\log_2 n)-1} 2^i * T^m(\frac{n}{2^i}) \tag{3.33}$$

$$T(n) = (n-1) * a + n * (c_{33} + c_{35}) + \sum_{i=0}^{(\log_2 n)-1} 2^i * T^m(\frac{n}{2^i}) \tag{3.34}$$

Desenvolvendo o somatório, lembrando que o T^m é igual an + b, mas usaremos bn + c pois já usamos "a"antes.

$$\sum_{i=0}^{(\log_2 n)-1} 2^i * (b * (\frac{n}{2^i}) + c) \tag{3.35}$$

$$\sum_{i=0}^{(\log_2 n)-1} \frac{b * 2^i * n}{2^i} + 2^i * c \tag{3.36}$$

$$\sum_{i=0}^{(\log_2 n)-1} b * n + 2^i * c \tag{3.37}$$

Podemos separar em dois somatórios:

$$\sum_{i=0}^{(\log_2 n)-1} b * n \tag{3.38}$$

$$\sum_{i=0}^{(\log_2 n)-1} 2^i * c \tag{3.39}$$

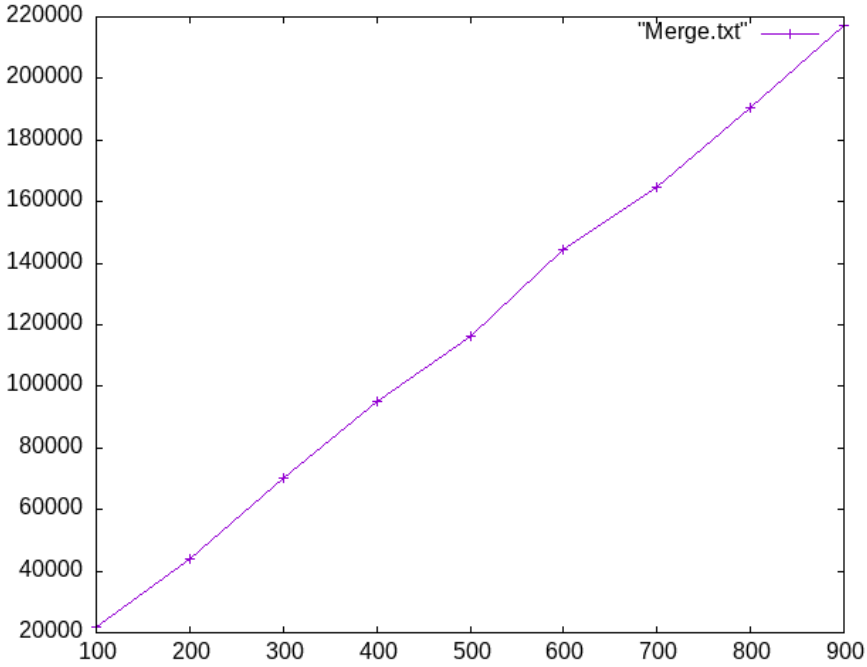

Figura 3.3: Tempo de execução do Merge Sort: $\Theta(n * \log_2 n)$ (tamanho do vetor x tempo) .

O somatório da equação 3.38 irá se comportar como uma multiplicação do valor do i máximo como termo dentro dele, pois irá ser uma soma de parcelas iguais, ficando da seguinte forma:

$$(\log_2 n) * b * n \tag{3.40}$$

O somatório da equação 3.39 tiramos a constante b para fora, e desenvolvemos o valor do somatório restante:

$$c * \sum_{i=0}^{(\log_2 n)-1} 2^i \tag{3.41}$$

$$c * (n-1) \tag{3.42}$$

Juntando tudo teremos:

$$T(n) = (n-1) * a + n * (c_{33} + c_{35}) + (\log_2 n) * b * n + c * (n-1) \tag{3.43}$$

Sendo assim podemos concluir que o tempo de execução do Merge Sort é de ordem $\Theta(n * \log_2 n)$. Isso também pode ser observado no gráfico (Veja a figura 3.3) feito a partir de simulações produzidas pelo código apresentado anteriormente.

3.4 Quick-sort()

O Quick Sort é um algoritmo de ordenação baseado no princípio "Dividir para Conquistar". Ele seleciona um elemento como pivô e particiona a lista em duas sublistas, uma

contendo elementos menores que o pivô e outra contendo elementos maiores. Em seguida, o processo é aplicado recursivamente às sublistas até que toda a lista esteja ordenada. Ele é um algoritmo in-place. (Veja a figura 2.3)

O tempo de execução do Quick Sort no melhor caso (o pivô sempre divide o vetor no meio) e no caso médio (o pivô quase sempre divide o vetor no meio) é de $\Theta(n * \log_2 n)$, mas no pior caso (o pivô fica em uma das pontas, e não no meio) ele será $\Theta(n^2)$. Como vimos que o merge é da forma an + b, o partition também será, pois segue a mesma estrutura.

O caso base do quick será quando o vetor tiver uma posição.

$$T(1) = c_{22} + c_{24} \tag{3.44}$$

3.4.1 Melhor caso Quick-sort()

Então vamos analisar o caso quando existe a recursão, analisando a recorrência, ele irá se comportar como o algoritmo de busca binária. (Veja a equação 3.45)

$$T(n) = c_{22} + c_{24} + c_{25} + c_{27} + c_{28} + 2 * T^q(\frac{n-1}{2}) + T^p(n) \tag{3.45}$$

$$c_{22} + c_{24} + c_{25} + c_{27} + c_{28} = a \tag{3.46}$$

1ºVamos substituir "n"por "n - 1/2"na equação 3.45:

$$T(\frac{n-1}{2}) = a + 2 * T^q(\frac{n-3}{4}) + T^p(\frac{n-1}{2}) \tag{3.47}$$

2ºSubstituindo na equação 3.45 teremos:

$$T(n) = a + 2 * [a + 2 * T^q(\frac{n-3}{4}) + T^p(\frac{n-1}{2})] + T^p(n) \tag{3.48}$$

Desenvolvendo:

$$T(n) = 3a + 4 * T^q(\frac{n-3}{4}) + 2 * T^p(\frac{n-1}{2}) + T^p(n) \tag{3.49}$$

3ºVamos substituir "n"por "n - 3/4"na equação 3.45:

$$T(\frac{n-3}{4}) = a + 2 * T^q(\frac{n-7}{8}) + T^p(\frac{n-3}{4}) \tag{3.50}$$

4ºSubstituindo na equação 3.49 teremos:

$$T(n) = 3a + 4 * [a + 2 * T^q(\frac{n-7}{8}) + T^p(\frac{n-3}{4})] + 2 * T^p(\frac{n-1}{2}) + T^p(n) \tag{3.51}$$

Desenvolvendo:

$$T(n) = 7a + 8 * T^q(\frac{n-7}{8}) + 4 * T^p(\frac{n-3}{4}) + 2 * T^p(\frac{n-1}{2}) + T^p(n) \tag{3.52}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = (2^x - 1) * a + 2^x * T^q(\frac{n - (2^x - 1)}{2^x}) + 2^{x-1} * T^p(\frac{n - (2^{x-1} - 1)}{2^{x-1}}) ... + 2^0 * T^p(\frac{n - (2^0 - 1)}{2^0}) \tag{3.53}$$

$$T(n) = (2^x - 1) * a + 2^x * T^q(\frac{n - (2^x - 1)}{2^x}) + \sum_{i=0}^{x-1} 2^i * T^p(\frac{n - (2^i - 1)}{2^i}) \tag{3.54}$$

Agora vamos igualar a recorrência ao caso base, então teremos:

$$\frac{n - (2^x - 1)}{2^x} = 1 \tag{3.55}$$

$$n - 2^x + 1 = 2^x \tag{3.56}$$

$$2^x + 2^x = n + 1 \tag{3.57}$$

A soma de duas potências com mesma base e mesmo expoente pode ser feita como uma multiplicação entre o valor da base vezes a potência, sabendo disso teremos:

$$2 * 2^x = n + 1 \tag{3.58}$$

$$2^x = \frac{n+1}{2} \tag{3.59}$$

$$\log_2 2^x = \log_2 \frac{n+1}{2} \tag{3.60}$$

$$x = \log_2 \frac{n+1}{2} \tag{3.61}$$

Substituindo na equação 3.54 teremos:

$$T(n) = (2^{\log_2 \frac{n+1}{2}} - 1) * a + 2^{\log_2 \frac{n+1}{2}} * T^q(\frac{n - (2^{\log_2 \frac{n+1}{2}} - 1)}{2^{\log_2 \frac{n+1}{2}}}) + \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * T^p(\frac{n - (2^i - 1)}{2^i}) \tag{3.62}$$

Simplificando teremos:

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * T^q(1) + \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * T^p(\frac{n - (2^i - 1)}{2^i}) \tag{3.63}$$

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * (c_{22} + c_{24}) + \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * T^p(\frac{n - (2^i - 1)}{2^i}) \tag{3.64}$$

Desenvolvendo o somatório, lembrando que o T^p é igual an + b, usaremos a forma bn + c pois já usamos "a"nessa equação:

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * (b * (\frac{n-(2^i-1)}{2^i}) + c) \tag{3.65}$$

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} b * (n-(2^i-1)) + c * 2^i \tag{3.66}$$

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} b * n - b * 2^i + b + c * 2^i \tag{3.67}$$

Podemos dividir em:

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} b * n \tag{3.68}$$

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} -b * 2^i \tag{3.69}$$

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} b \tag{3.70}$$

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} c * 2^i \tag{3.71}$$

Como já foi observado no somatório da equação 3.38 o somatório 3.68 irá se comportar como uma multiplicação do valor do i máximo como termo dentro dele, pois irá ser uma soma de parcelas iguais, ficando da seguinte forma:

$$(\log_2 \frac{n+1}{2}) * b * n \tag{3.72}$$

O mesmo vale para o somatório da equação 3.70:

$$(\log_2 \frac{n+1}{2}) * b \tag{3.73}$$

Agora para o somatório da equação 3.69 a constante b sai para fora, e desenvolvemos o somatório restante:

$$c * \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i \tag{3.74}$$

$$c * (\frac{n-1}{2}) \tag{3.75}$$

Mesma coisa para a equação 3.71:

$$-b * \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i \tag{3.76}$$

$$-b * (\frac{n-1}{2}) \tag{3.77}$$

Juntando tudo teremos:

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * (c_{22} + c_{24}) + (\log_2 \frac{n+1}{2}) * b * n + c * (\frac{n-1}{2}) - b * (\frac{n-1}{2}) \tag{3.78}$$

Sendo assim podemos concluir que o tempo de execução do quick sort no melhor caso é de ordem $\Theta(n * \log_2 n)$.

3.4.2 Pior caso Quick-sort()

No pior caso o pivô sempre ficará em uma das pontas, resultando na seguinte equação 3.80:

$$T(n) = c_{22} + c_{24} + c_{25} + c_{27} + c_{28} + T^q(n-1) + T^q(0) + T^p(n) \tag{3.79}$$

$$T(n) = a + T^q(n-1) + T^p(n) \tag{3.80}$$

1ºVamos substituir "n"por "n - 1"na equação 3.80:

$$T(n-1) = a + T^q(n-2) + T^p(n-1) \tag{3.81}$$

2ºSubstituindo na equação 3.80 teremos:

$$T(n) = 2 * a + T^q(n-2) + T^p(n-1) + T^p(n) \tag{3.82}$$

3ºVamos substituir "n"por "n - 2"na equação 3.80:

$$T(n-2) = a + T^q(n-3) + T^p(n-2) \tag{3.83}$$

4ºSubstituindo na equação 3.83 teremos:

$$T(n) = 3 * a + T^q(n-3) + T^p(n-2) + T^p(n-1) + T^p(n) \tag{3.84}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = x * a + T^q(n-x) + T^p(n-x-1) + T^p(n-x-2) + ... + T^p(n) \tag{3.85}$$

$$T(n) = x * a + T^q(n-x) + \sum_{i=0}^{x-1} T^p(n-i) \tag{3.86}$$

Agora vamos igualar a recorrência ao caso base, então teremos:

$$n - x = 1 \tag{3.87}$$

$$x = n - 1 \tag{3.88}$$

Substituindo na equação 3.86 teremos:

$$T(n) = (n-1) * a + T^q(n - (n-1)) + \sum_{i=0}^{(n-1)-1} T^p(n-i) \tag{3.89}$$

Simplificando teremos:

$$T(n) = (n-1) * a + T^q(1) + \sum_{i=0}^{n} T^p(n-i) \tag{3.90}$$

$$T(n) = (n-1) * a + (c_{22} + c_{24}) + \sum_{i=0}^{n} T^p(n-i) \tag{3.91}$$

Desenvolvendo o somátorio, lembrando que T^p é igual an + b, iremos chamar de bn + c pois já usamos o "a":

$$\sum_{i=0}^{n} b * (n-i) + c \tag{3.92}$$

$$\sum_{i=0}^{n} b * n - b * i + c \tag{3.93}$$

Separando teremos:

$$\sum_{i=0}^{n} b * n \tag{3.94}$$

$$\sum_{i=0}^{n} -b * i \tag{3.95}$$

$$\sum_{i=0}^{n} c \tag{3.96}$$

Desenvolvendo o somatório 3.94:

$$n * (b * n) \tag{3.97}$$

$$n * b + n^2 \tag{3.98}$$

Desenvolvendo o somatório 3.95:

$$-b * (\sum_{i=0}^{n} i) \tag{3.99}$$

$$-b * (\frac{n}{2} * (n+1)) \tag{3.100}$$

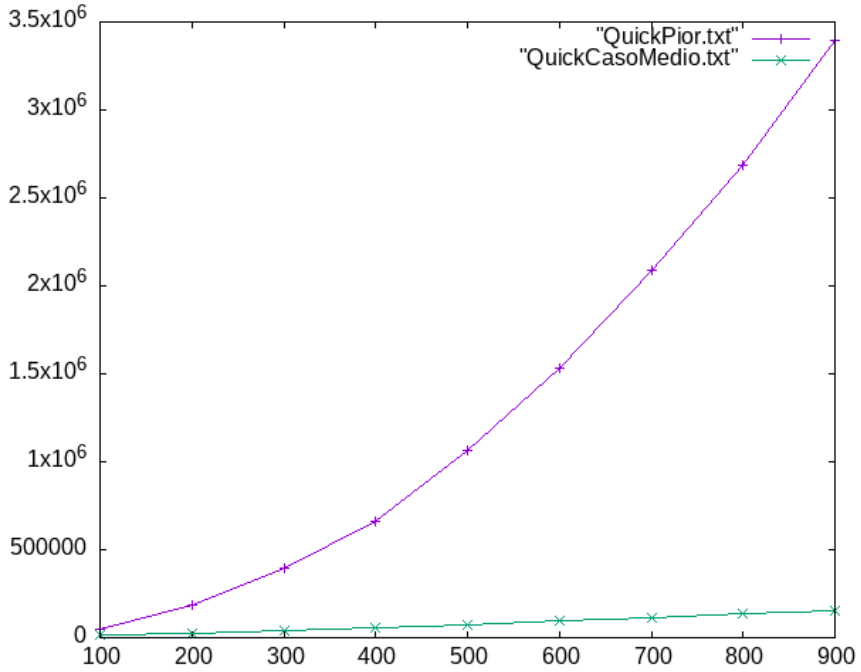

Figura 3.4: Tempo de execução do Quick Sort no caso esperado ($\Theta(n * \log_2 n)$) e pior caso ($\Theta(n^2)$). (tamanho do vetor x tempo) .

$$-b * (\frac{n^2 + n}{2}) \tag{3.101}$$

Desenvolvendo o somatório 3.96:

$$n * c \tag{3.102}$$

Juntando tudo teremos:

$$T(n) = (n-1) * a + (c_{22} + c_{24}) + n * b + n^2 - b * (\frac{n^2 + n}{2}) + n * c \tag{3.103}$$

Sendo assim podemos concluir que o tempo de execução do Quick Sort no pior caso é de $\Theta(n^2)$.

3.4.3 Caso esperado Quick-sort()

O caso esperado do quick sort estará mais perto do seu melhor caso, sendo $\Theta(n * \log_2 n)$. Isso pode ser verificado na figura 3.4. Note como o pior caso cresce mais rápido do que o caso esperado.

3.5 Distribution-sort()

O Distribution Sort se baseia na contagem do número de ocorrências de cada elemento em uma lista. Ele cria um vetor auxiliar para a contagem, e a partir dele faz a ordenação.

Ele não é um algoritmo in-place. (Veja figura 2.5)

O tempo de execução do Distribution Sort sempre será linear, mas diferentemente dos algoritmos visto antes, ele não depende só de "n", mas também de uma variável "k", que faz referência a diferença entre o maior número no vetor, e o menor, sendo então $\Theta(n + k)$. Como a equação ficará bem grande, para melhor entendimento irei dividir em partes.

1ºPegando as primeiras linhas constantes.

$$c_2 + c_3 + c_4 + c_6 + c_7 \tag{3.104}$$

2ºA função max e min possui um laço de repetição que pecorre o vetor, logo elas são an + b. Então teremos:

$$2 * (a * n + b) \tag{3.105}$$

$$2 * a * n + 2 * b \tag{3.106}$$

3ºPegando o for da linha 9. Obs: "k"representa "b - s + 2", ou seja, o tamanho do vetor auxiliar que foi criado.

$$c_9 * (k + 1) + c_{10} * k \tag{3.107}$$

Desenvolvendo ficaremos com:

$$c_9 + c_9 * k + c_{10} * k \tag{3.108}$$

4ºPegando o for da linha 13.

$$c_{13} * (n + 1) + c_{14} * n \tag{3.109}$$

Desenvolvendo ficaremos com:

$$c_{13} + c_{13} * n + c_{14} * n \tag{3.110}$$

5ºPegando o for da linha 17.

$$c_{17} * k + c_{18} * (k - 1) \tag{3.111}$$

Desenvolvendo ficaremos com:

$$c_{17} * k + c_{18} * k - c_{18} \tag{3.112}$$

6ºPegando o for da linha 21. (Veja equação x.x)

$$c_{21} * (n + 1) + (c_{22} + c_{23} + c_{24}) * n \tag{3.113}$$

Desenvolvendo ficaremos com:

$$c_{21} + c_{21} * n + (c_{22} + c_{23} + c_{24}) * n \tag{3.114}$$

7ºPegando o for da linha 27.

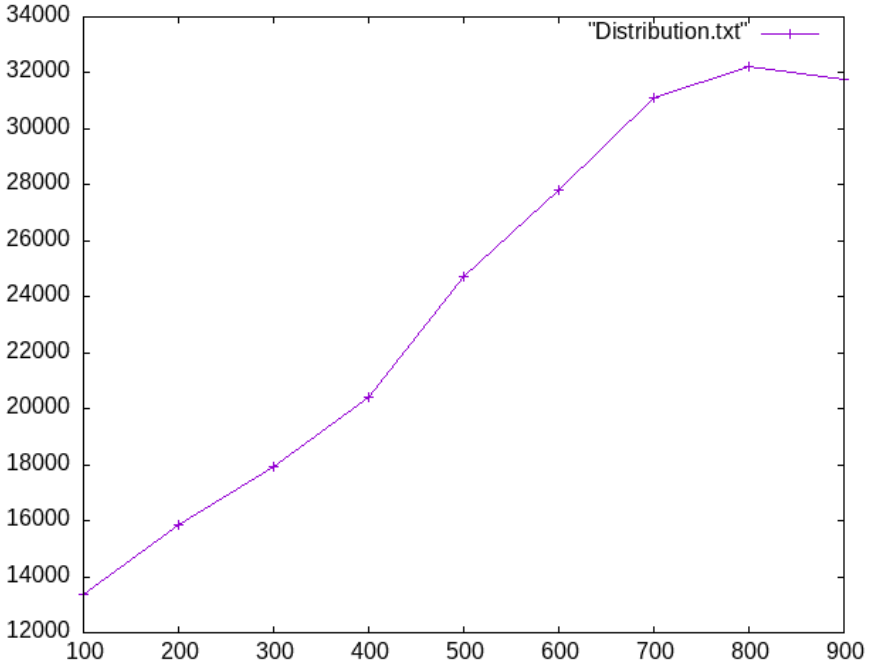

Figura 3.5: Tempo de execução do Distribution Sort: $\Theta(n+k)$ (tamanho do vetor x tempo) .

$$c_{27} * (n+1) + c_{28} * n \tag{3.115}$$

Desenvolvendo ficaremos com:

$$c_{27} + c_{27} * n + c_{28} * n \tag{3.116}$$

8ºPegando as última linhas constantes.

$$c_{31} + c_{32} \tag{3.117}$$

9ºAgora iremos considerar a soma das constantes solta como "c". Logo juntando tudo ficaremos com : (Veja equação 3.118)

$$T(n,k) = c+2*a*n+c_9*k+c_{10}*k+c_{13}*n+c_{14}*n+c_{17}*k+c_{18}*k+c_{21}*n+(c_{22}+c_{23}+c_{24})*n+c_{27}*n+c_{28}*n \tag{3.118}$$

Botando em evidência para melhor visualização:

$$T(n,k) = c+(2*a+c_{13}+c_{14}+c_{21}+c_{22}+c_{23}+c_{24}+c_{27}+c_{28})*n+(c_9+c_{10}+c_{17}+c_{18})*k \tag{3.119}$$

Sendo assim podemos concluir que o tempo de execução do Distribution Sort é de ordem $\Theta(n+k)$. Isso também pode ser observado no gráfico (Veja a figura 3.5) feito a partir de simulações produzidas pelo código 2.5.

4. Resultados

Aqui iremos comparar o tempo de execução dos algoritimos usando gráficos.

4.1 Selection x Insertion

Como vimos antes o tempo do selection e o do isertion é de ordem $\Theta(n^2)$. No gŕafico 4.1 podemos observar como eles crescem de forma quadrática, porém o selection ainda cresce mais rápido do que o insertion devido ao fato dele sempre executar seus dois laços de repetição onde um se encontra dentro do outro.

4.2 Insertion x Merge

Como vimos antes o tempo do isertion é de ordem $\Theta(n^2)$ e o do merge é de ordem $\Theta(n * \log_2 n)$. No gŕafico 4.2 podemos observar como o insertion cresce bem mais rápido do que o merge.

4.3 Merge x Quick

Como vimos antes o tempo do merge e o do quick é de ordem $\Theta(n * \log_2 n)$. No gŕafico 4.3 podemos observar como eles crescem bem perto um do outro, apesar do quick ter um desempenho melhor em relação do merge já que seu elemento que divide o vetor ao meio sempre estará na sua posição ordenado.

4.4 Quick x Distribution

Como vimos antes o tempo do quick é de ordem $\Theta(n * \log_2 n)$ e o do distribution é de ordem $\Theta(n + k)$. No gŕafico 4.4 podemos observar como o quick cresce bem mais rápido do que o distribution.

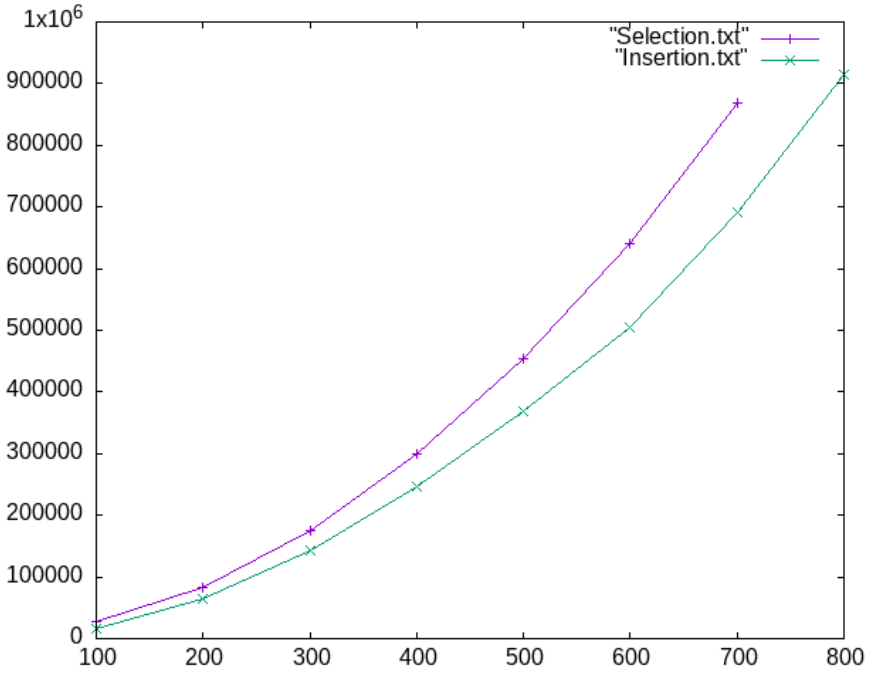

Figura 4.1: Selection x Insertion (tamanho do vetor x tempo) .

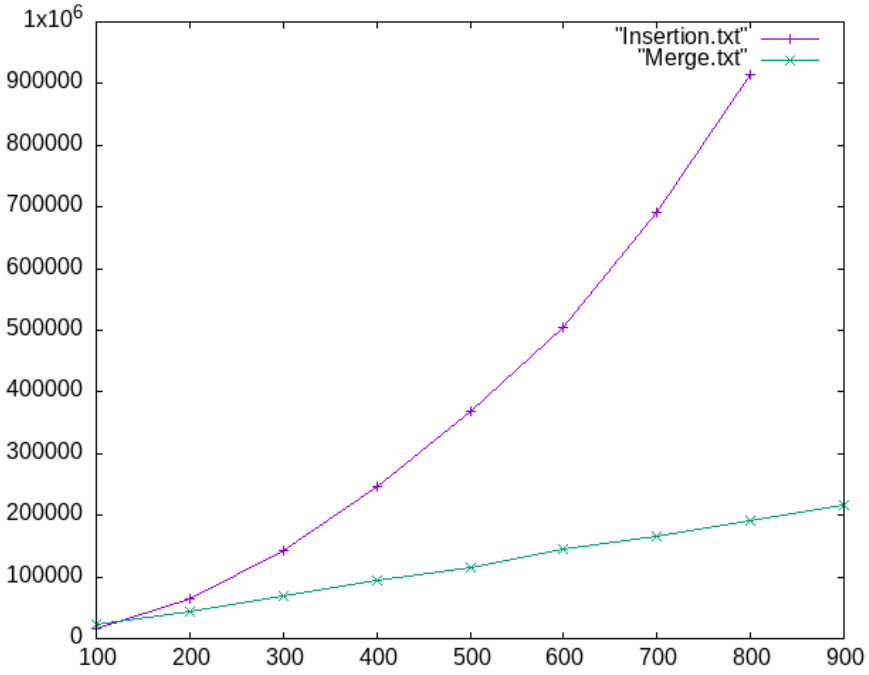

Figura 4.2: Tempo de execução do Insertion x Merge. (tamanho do vetor x tempo) .

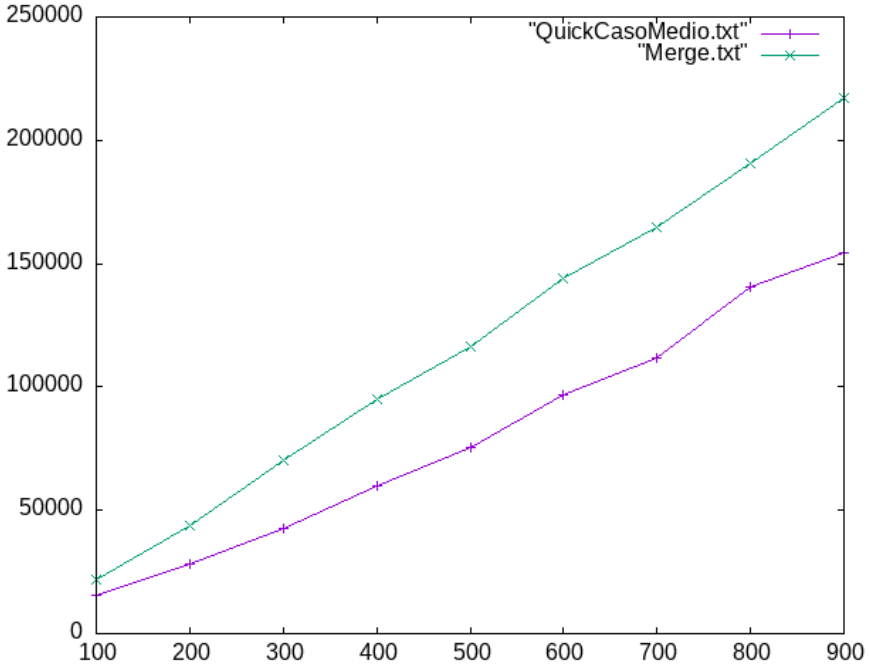

Figura 4.3: Tempo de execução do Merge x quick: $\Theta(n+k)$ (tamanho do vetor x tempo) .

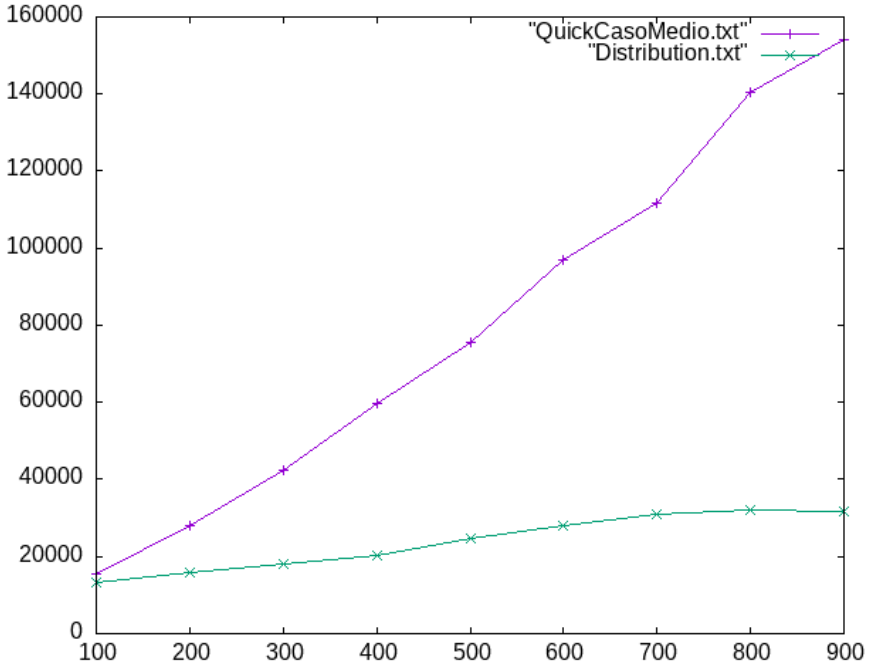

Figura 4.4: Tempo de execução do Quick x Distribution. (tamanho do vetor x tempo) .

5. Conclusão

Em conclusão, os algoritmos de ordenação desempenham um papel fundamental no tratamento de conjuntos de dados. Ao longo deste trabalho, foram explorados alguns algoritmos apresentados pelo professor em sala de aula.

Cada algoritmo possui características e complexidades próprias, e a escolha adequada depende do contexto específico e dos requisitos de desempenho. Algoritmos de ordenação mais simples, como o Selection Sort e o Insertion Sort, são amplamente utilizados por iniciantes na área de programação devido à sua facilidade de entendimento e implementação. No entanto, eles podem apresentar ineficiência em conjuntos de dados grandes. Por outro lado, algoritmos mais sofisticados, como o Merge Sort, Quick Sort e Distribution Sort, oferecem uma melhor eficiência em termos de tempo de execução, embora possam exigir mais recursos computacionais.

Portanto, compreender as características distintas de cada algoritmo nos permite selecionar a abordagem mais adequada para resolver problemas de ordenação, considerando a eficiência e a escalabilidade como critérios importantes. Ao dominar esses algoritmos e sua aplicação correta, torna-se possível desenvolver soluções otimizadas e eficientes.