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

Emanuel Alves de Medeiros

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, 16 de julho de 2024

Sumário

1. Introdução

A velocidade ordendando dados era e ainda é um desafio na programação desde muito tempo, e com isso, apareceram vários algoritmos de ordenação que possuem suas peculiaridades, alguns mais rápidos que outros, alguns mais rápidos até certos tamanhos de listas.

Ordenar listas pequenas não é um problema relevante, mas a coisa muda quando as listas ultrapassam um tamanho razoável. O trabalho a seguir irá realizar um estudo sobre os seguintes algoritmos: selection sort, insertion sort, merge sort, quick sort e sistribution sort. O estudo será feio no tocante aos algoritmos, o cálculo teórico do tempo de execução, a imprementação em Swift, a comparação entre os casos médios, melhores e piores, e também a comparação entre os algoritmos.

2. Códigos dos algoritmos

Todos os códigos feitos nesse trabalho foram levando em consideração os pseudocódigos que se do GitHub do professor, sendo escritos usando a linguagem Swift, realizando adaptações quando necessário.

2.1 Selection sort

O pseudocódigo do selection sort está na figura 2.1, e a implementação em swift está na figura 2.1. As principais adaptações foram:

O selection sort é um método simples para ordenar um conjunto de dados. Sua lógica consiste em encontrar a todo momento o menor elemento não ordenado e colocá-lo na posição correta. Ele é um algoritmo in-place, ou seja, ele ordena no próprio vetor, ou seja, não precisa ter um gasto extra com um vetor auxiliar ou algo do tipo.
1 - É necessário declarar algumas variáveis.

2 - Os laços que necessitam percorrer o vetor desde o primeiro elemento, devem começar em 0 ao invés de 1.

3- Nesse algoritmo, o priemiro laço percorre o vetor até a penúltima posição, que em swift é n-2, mas colocando ¡n-1, ele ignora o elemento da posição n-1, no fim das contas é a mesma coisa que o pseudocódigo.

4- O segundo for vai de i até o fim do vetor, no pseudocódigo ele ia até n, mas em Swift a última posição do vetor é n-1, colocando ¡n, ele ignora o n e percorre até n-1 inclusivo.

5- A função swap foi substituída por uma linha específica em swift.

```
algorithm sselection-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 sort.

```
func selectionSort(v: inout [Int], n: Int) {
    for i in 0..<n-1 {
        var m = i
        for j in (i+1)..<n {
            if v[m] > v[j] {
                m = j
            }
        }
        (v[i], v[m]) = (v[m], v[i])
    }
}
```

Figura 2.1: Código em Swift: Selection sort.

2.2 Insertion sort

O pseudocódigo do insertion sort está na figura 2.2, e a implementação em swift está na figura 2.2. As principais adaptações foram:

O insertion sort é um método simples que percorre o vetor que deve ser ordenado e altera a posição de cada elemento para a correta dentro da parte já ordenada do vetor. A cada iteração percorrendo o vetor, o elemento atual é comparado com os elementos da parte ordenada do vetor. O objetivo é encontrar a posição correta onde o elemento deve ser inserido. Ele é um algoritmo in-place.

1 - É necessário declarar algumas variáveis.

2 - O laço que precisa percorrer o vetor inteiro, devem começar em 0 e ir até n exclusivo, no caso ¡n.

3- A função swap foi substituída por uma linha específica em swift.

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 sort.

2.3 Merge sort

O pseudocódigo do merge sort está na figura 2.3, e a implementação em swift está na figura 2.3. As principais adaptações foram:

O merge sort segue o paradigma "Dividir para Conquistar". Ele divide a lista em

```
func insertionSort(v: inout [Int], n: Int) {
    for i in 1..<n {
        var j = i
        while j > 0 && v[j - 1] > v[j] {
            (v[j], v[j - 1]) = (v[j - 1], v[j])
            j -= 1
        }
    }
}
```

Figura 2.2: Código em Swift: Insertion sort.

sublistas menores de forma recursiva, as ordenam e combinam para obter a lista final ordenada. Ele não é um algoritmo in-place, ou seja, cria um vetor auxiliar. No fim, substitui o vetor original pelo auxiliar.

1 - É necessário declarar algumas variáveis.

2 - A função merge necessita de um vetor auxiliar w com o mesmo tamanho, como em cada iteração ele tem elemento, ele é adicionado no w, fazendo com que eles tenham o mesmo tamanho.

3 - Os dois laços percorrem o vetor do início, começando em 0.

```
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 sort.

```
func mergeSort(v: inout [Int], s: Int, e: Int) {
    if s < e {
        let m = (s+e)/2
        mergeSort(v: &v, s: s, e: m)
        mergeSort(v: &v, s: m+1, e: e)
        merge(v: &v, s: s, m: m, e: e)
    }
}

func merge(v: inout [Int], s: Int, m: Int, e: Int) {
    var p = s
    var q = m + 1
    var w: [Int] = []
    for _ in 0..<(e - s + 1) {
        if (q > e) || ((p <= m) && (v[p] < v[q])) {
            w.append(v[p])
            p += 1
        } else {
            w.append(v[q])
            q += 1
        }
    }
    for i in (0..<(e - s + 1)) {
        v[s + i] = w[i]
    }
}
```

Figura 2.3: Código em Swift: Merge sort.

2.4 Quick sort

O pseudocódigo do quick sort está na figura 2.4, e a implementação em swift está na figura 2.4. As principais adaptações foram:

O quick sort é também 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 nas sublistas até que tudo esteja ordenado. Ele é um algoritmo in-place, ou seja, não necessita de um vetor auxiliar.

1 - É preciso definir o tipo de retorno das funções que têm retorno e, Swift

algorithm quick-sort(v, s, e)
| if $s < e$ then
| | $p \leftarrow$ partition(v, s, e)
| | quick-sort$(v, s, p-1)$
| | quick-sort$(v, p+1, e)$

algorithm partition(v, s, e)
| $d \leftarrow s - 1$
| for i from s to $(e-1)$ do
| | if $v[i] \leq v[e]$ then
| | | $d \leftarrow d + 1$
| | | swap$(v[d], v[i])$
| swap$(v[d+1], v[e])$
| return $(d+1)$

Algoritmo 2.4: Pseudocódigo Quick sort.

2.5 Distribution sort

O pseudocódigo do distribution sort está na figura **??**, e a implementação em swift está na figura 2.5. As principais adaptações foram:

O distribution sort se baseia na contagem do número de ocorrências de cada elemento na lista. Ele não é um algoritmo in-place, ou seja, cria um vetor auxiliar para a contagem, e a partir dele faz a ordenação.

1 - É necessário declarar algumas variáveis.

2 - A função merge necessita de um vetor auxiliar w com o mesmo tamanho, como em cada iteração ele tem elemento, ele é adicionado no w, fazendo com que eles tenham o mesmo tamanho.

3 - O terceiro laço começa no segundo elemento, que em Swift é a posição 1, os outros começam na primeira posição, 0.

```
func quickSort(v: inout [Int], s: Int, e: Int) {
    if s < e {
        let p = partition(v: &v, s: s, e: e)
        quickSort(v: &v, s: s, e: p - 1)
        quickSort(v: &v, s: p + 1, e: e)
    }
}

func partition(v: inout [Int], s: Int, e: Int) -> Int {
    var d = s - 1
    for i in s..<(e) {
        if v[i] <= v[e] {
            d += 1
            (v[d], v[i]) = (v[i], v[d])
        }
    }
    (v[d + 1], v[e]) = (v[e], v[d + 1])
    return d + 1
}
```

Figura 2.4: Código em Swift: Quick sort.

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

Algoritmo 2.5: Pseudocódigo Distribution sort.

```
func distributionSort(v: inout [Int], n: Int) {
    let s = v.min()!
    let b = v.max()!
    var c: [Int] = []
    var w: [Int] = []
    for _ in 0..<(b - s + 2) {
        c.append(0)
    }
    for i in 0..<n {
        c[v[i] - s + 1] = c[v[i] - s + 1] + 1
        w.append(0)
    }
    for i in 1..<(b - s + 2) {
        c[i] += c[i - 1]
    }
    for i in 0..<n {
        let d = v[i] - s + 1
        w[c[d] - 1] = v[i]
        c[d] -= 1
    }
    for i in 0..<n {
        v[i] = w[i]
    }
}
```

Figura 2.5: Código em Swift: Distribution Sort.

3. Tempo de execução

3.1 Selection sort

A análise teórica do Selection sort leva em consideração o pseudocódigo da figura 2.1

O tempo de execução do Selection sort é de ordem $\Theta(n^2)$, isso se dá por causa da presença de um laço dentro do outro, o que faz com que aumentando pouco o tamanho do vetor, o tempo aumenta rapidamente, o que o torna um algoritmo pouco eficiente. O for dentro do outro só é um problema grave pela falta de uma condição de parada. Como o algoritmo sempre vai percorrer a lista inteira, e não há nada que interrompa esse laço fazendo com que ele "corte caminho", ele não tem um melhor ou pior caso, ele só terá o caso médio. Veja mais na equação 3.1

$$T(n) = c_1 * n + c_2(n-1) + \sum_{i=1}^{n}(i * c_3) + \sum_{i=1}^{n-1}(i * c_4) \tag{3.1}$$

Resolvendo o primeiro somatório:

$$\sum_{i=1}^{n}(i * c_3) = c_3 * \sum_{i=1}^{n}(i) \tag{3.2}$$

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

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

$$c_3 * (\frac{n^2+n}{2}) \tag{3.5}$$

Resolvendo o segundo somatório é a mesma coisa, mas como ele vai até n - 1, teremos que subtrair n do somatório:

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

$$c_4 * (\frac{n^2+n}{2} - n) \tag{3.7}$$

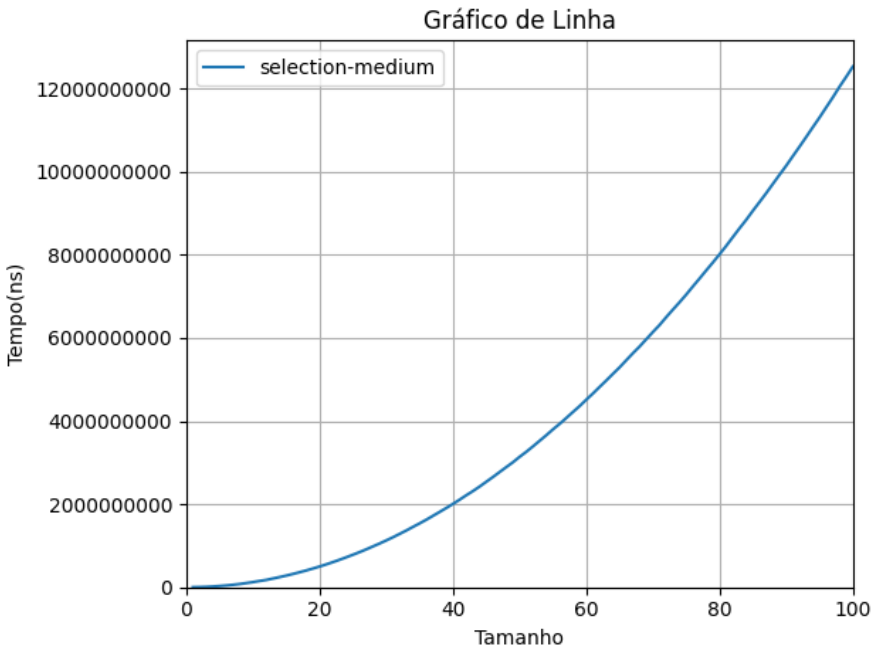

Figura 3.1: Tempo de execução do Selection (tamanho do vetor x tempo).

Podemos reescrever a equação geral como:

$$T(n) = c_1 * n + c_2(n-1) + c_3 * (\frac{n^2+n}{2}) + c_4 * (\frac{n^2+n}{2} - n) \tag{3.8}$$

Com essa equação podemos concluir que a ordem do tempo de execução do Selection sort é $\Theta(n^2)$. Podendo ser observado no gráfico (Veja a figura 3.1) produzido usando o código apresentado.

3.2 Insertion sort

O tempo de execução do Insertion sort, diferente do Selection possui melhor caso, pior caso, e o caso esperado, dependendo do vetor de entrada. O pior caso acontece quando o vetor estiver ordenado de forma decrescente, o caso esperado é quando o vetor está embaralhado, sendo de ordem $\Theta(n^2)$, e no melhor caso, quando o vetor já está ordenado, ele nunca entrará no while, sendo de ordem $\Theta(n)$ veja mais no gráfico geral 3.2.

3.2.1 Melhor caso Insertion sort

Equação quando do melhor caso: (Veja figura 3.9)

$$T(n) = c_1 * n + (c_2 + c_3) * (n-1) \tag{3.9}$$

Com essa equação podemos concluir que o tempo de execução do melhor caso é de ordem $\Theta(n)$.

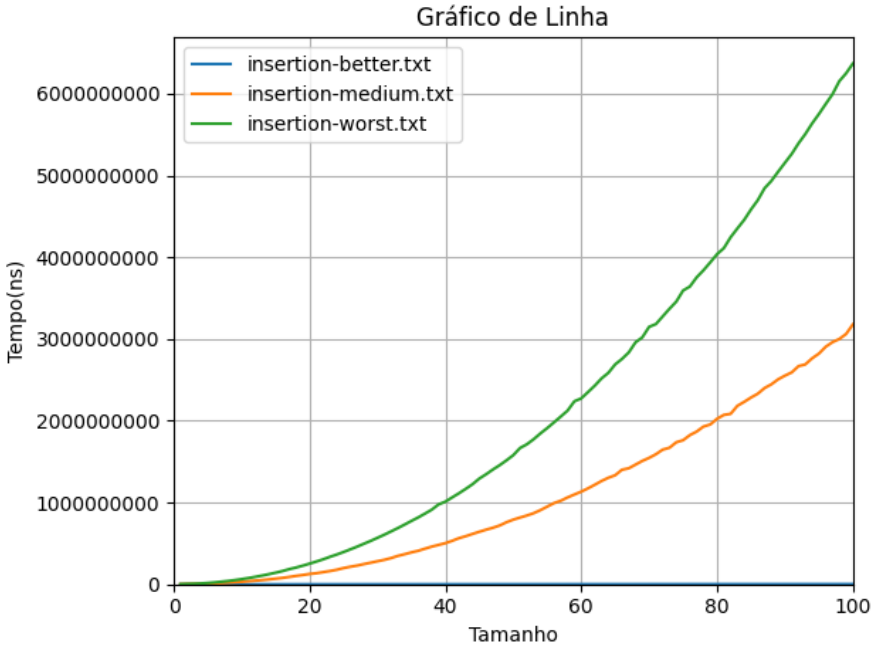

Figura 3.2: Tempo de execução do Insertion em todos os casos (tamanho do vetor x tempo) .

3.2.2 Pior caso Insertion sort

Equação do pior caso: (Veja figura 3.10)

$$T(n) = c_1 * n + c_2 * (n-1) + \sum_{i=1}^{n} i * c_3 + \sum_{i=1}^{n-1} i * (c_4 + c_5) \tag{3.10}$$

Desenvolvendo o primeiro somatório:

$$\sum_{i=1}^{n} i * c_3 \tag{3.11}$$

$$c_3 * \frac{n^2+n}{2} \tag{3.12}$$

Desenvolvendo o segundo:

$$\sum_{i=1}^{n-1} i * (c_4 + c_5) \tag{3.13}$$

$$(c_4 + c_5) * (\frac{n^2+n}{2} - n) \tag{3.14}$$

Equação final:

$$T(n) = c_1 * n + c_2 * (n-1) + c_3 * \frac{n^2+n}{2} + (c_4 + c_5) * (\frac{n^2+n}{2} - n) \tag{3.15}$$

Com essa equação podemos concluir que a ordem do tempo de execução do melhor caso do Insertion sort é $\Theta(n^2)$, observe melhor no gráfico 3.2.

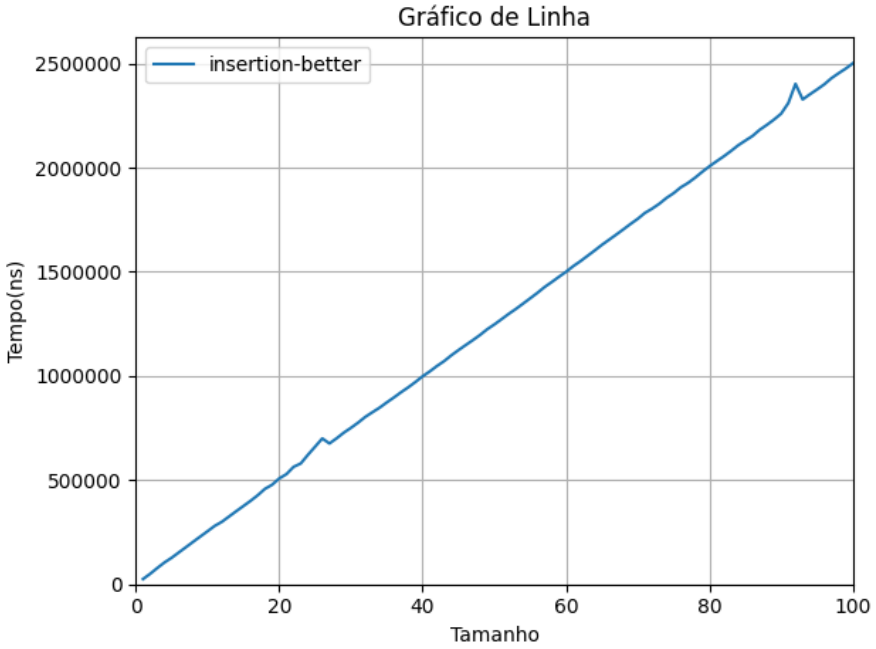

Figura 3.3: Tempo de execução do Insertion no melhor caso (tamanho do vetor x tempo) .

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 se dá pois a maioria das vezes os dois laços de repetição irão ser executados. Confira na figura 3.4, o crescimento dos 3 casos.

3.3 Merge sort

A ordem do tempo de execução do Merge sort é $\Theta(n * \log_2 n)$. Isso acontece graças ao fato dele dividir o vetor em subvetores até restarem só dois ou um elemento, então os ordena. Primeiro calcularemos o tempo de execução da função merge (Veja a equação 3.16).

$$T(n) = c_1 + c_2 + c_3(n+1) + c_9 * n + (c_5 + c_6) * n + c_{10}(n+1) + c_{11} * n \tag{3.16}$$

Com essa equação, podemos dizer que ele será linear, então podemos substituir como an + b no cálculo do Merge sort. Como ele é recursivo, temos que descobrir o caso base do nosso algortimo. O caso base é quando não tem recurção, que é quando nosso vetor só tiver uma posição (Veja a equação 3.17).

$$T(1) = c_1 \tag{3.17}$$

Vamos analisar o caso quando tem recursão, analisando a recorrência. (Veja a equação 3.18)

$$T(n) = c_1 + c_2 + T_{ms}(\frac{n}{2}) + T_{ms}(\frac{n}{2}) + T_m(n) \tag{3.18}$$

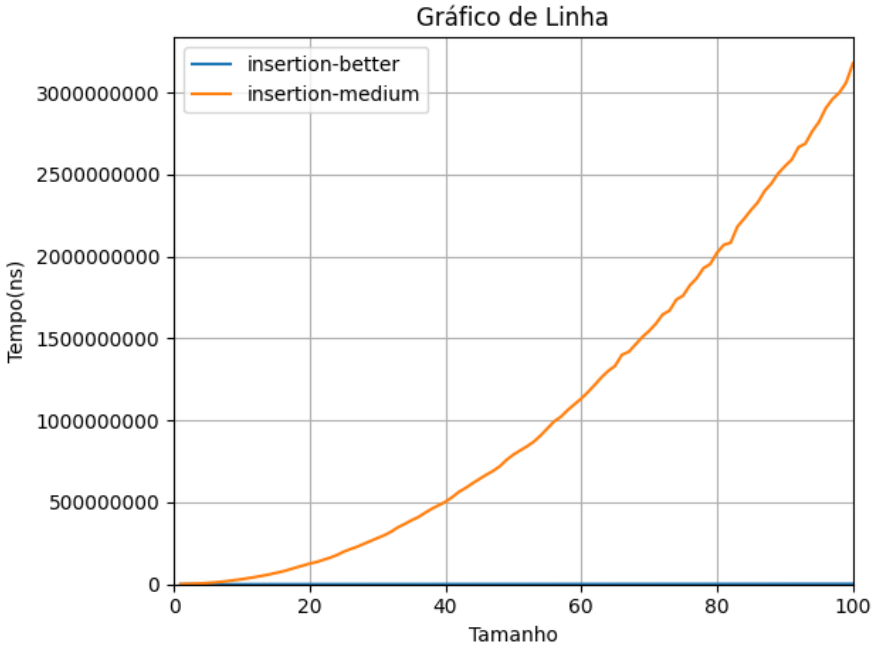

Figura 3.4: Tempo de execução do Insertion caso médio vs melhor (tamanho do vetor x tempo) .

$$T(n) = c_1 + c_2 + 2 * T_{ms}(\frac{n}{2}) + T_m(n) \tag{3.19}$$

Observe que como existem duas recursões, podemos simplificar juntando-os, também precisamos levar em consideração o tempo de execução do merge calculado anteriormente (Veja a equação 3.16). Vamos observar o comportamento da recorrência para achar um padrão. Para melhor visualização junte as contantes 3.20.

$$c_1 + c_2 = a \tag{3.20}$$

1ºSubstituindo "n"por "n/2"na equação 3.18:

$$T(\frac{n}{2}) = a + 2 * T_{ms}(\frac{n}{4}) + T_m(\frac{n}{2}) \tag{3.21}$$

2ºSubstituindo o resultado na equação 3.18:

$$T(n) = a + 2 * [a + 2 * T_{ms}(\frac{n}{4}) + T_m(\frac{n}{2})] + Tm(n) \tag{3.22}$$

$$T(n) = 3a + 4 * T_{ms}(\frac{n}{4}) + 2 * T_m(\frac{n}{2}) + T_m(n) \tag{3.23}$$

3ºSubstituindo "n"por "n/4"na equação 3.18:

$$T(\frac{n}{4}) = a + 2 * T_{ms}(\frac{n}{8}) + T_m(\frac{n}{4}) \tag{3.24}$$

4ºSubstituindo na equação 3.23:

$$T(n) = 3a + 4 * [a + 2 * T_{ms}(\frac{n}{8}) + T_m(\frac{n}{4})] + 2 * T_m(\frac{n}{2}) + T_m(n) \tag{3.25}$$

$$T(n) = 7a + 8 * T_{ms}(\frac{n}{8}) + 4 * T_m(\frac{n}{4}) + 2 * T_m(\frac{n}{2}) + T_m(n) \tag{3.26}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = (2^x - 1) * a + 2^x * T_{ms}(\frac{n}{2^x}) + 2^{x-1} * T_m(\frac{n}{2^{x-1}}) + 2^{x-2} * T_m(\frac{n}{2^{x-2}}) + ... + 2^0 * T_m(\frac{n}{2^0}) \tag{3.27}$$

$$T(n) = (2^x - 1) * a + 2^x * T_{ms}(\frac{n}{2^x}) + \sum_{i=0}^{x-1} 2^i * T_m(\frac{n}{2^i}) \tag{3.28}$$

Temos que saber quando a esse somatório acaba, logo, devemos igualar a recorrência ao caso base:

$$\frac{n}{2^x} = 1 \tag{3.29}$$

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

$$\log_2 2^x = \log_2 n \tag{3.31}$$

$$x = \log_2 n \tag{3.32}$$

Substituindo na equação 3.28:

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

$$T(n) = (n - 1) * a + n * T_{ms}(1) + \sum_{i=0}^{(\log_2 n)-1} 2^i * T_m(\frac{n}{2^i}) \tag{3.34}$$

$$T(n) = (n - 1) * a + n * c_1 + \sum_{i=0}^{(\log_2 n)-1} 2^i * T_m(\frac{n}{2^i}) \tag{3.35}$$

Desenvolvendo o somatório, lembrando que o T_m é igual an + b, mas para não conflitar o "a"que já usamos, usaremos bn + c.

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

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

$$\sum_{i=0}^{(\log_2 n)-1} b * n + 2^i * c \tag{3.38}$$

Separando em dois somatórios:

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

$$\sum_{i=0}^{(\log_2 n)-1} 2^i * c \tag{3.40}$$

O somatório da equação 3.39 se comporta como uma multiplicação do valor do i máximo com o termo dentro dele, pois irá ser uma soma de parcelas iguais, ficando assim:

$$(\log_2 n) * b * n \tag{3.41}$$

O somatório da equação 3.40 tiramos a constante b para fora, e desenvolvemos o restante:

$$c * \sum_{i=0}^{(\log_2 n)-1} 2^i \tag{3.42}$$

$$c * (n-1) \tag{3.43}$$

Juntando tudo:

$$T(n) = (n-1) * a + n * c_1 + (\log_2 n) * b * n + c * (n-1) \tag{3.44}$$

Com essa equação podemos concluir que a ordem do tempo de execução do Merge sort é $\Theta(n * \log_2 n)$. Podendo ser observado no gráfico (Veja a figura 3.5) produzido usando o código apresentado.

3.4 Quick sort

O Quick sort possui um melhor caso, onde o pivô sempre divide o vetor ao meio e caso médio onde pivô quase sempre divide o vetor no meio, ambos tendo ordem do tempo de execução de $\Theta(n * \log_2 n)$, mas no pior caso onde pivô fica em uma das pontas, não no meio, ele será $\Theta(n^2)$. Assim como o merge é an + b, o partition também será, pois também é linear.

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

$$T(1) = c_1 \tag{3.45}$$

3.4.1 Melhor caso Quick sort

Vamos analisar o caso quando existe a recursão e a recorrência. (Veja a equação 3.46)

$$T(n) = c_1 + c_2 + 2 * T_q(\frac{n-1}{2}) + T_p(n) \tag{3.46}$$

$$c_{22} + c_1 + c_2 = a \tag{3.47}$$

1ºSubstituindo "n"por "n - 1/2"na equação 3.46:

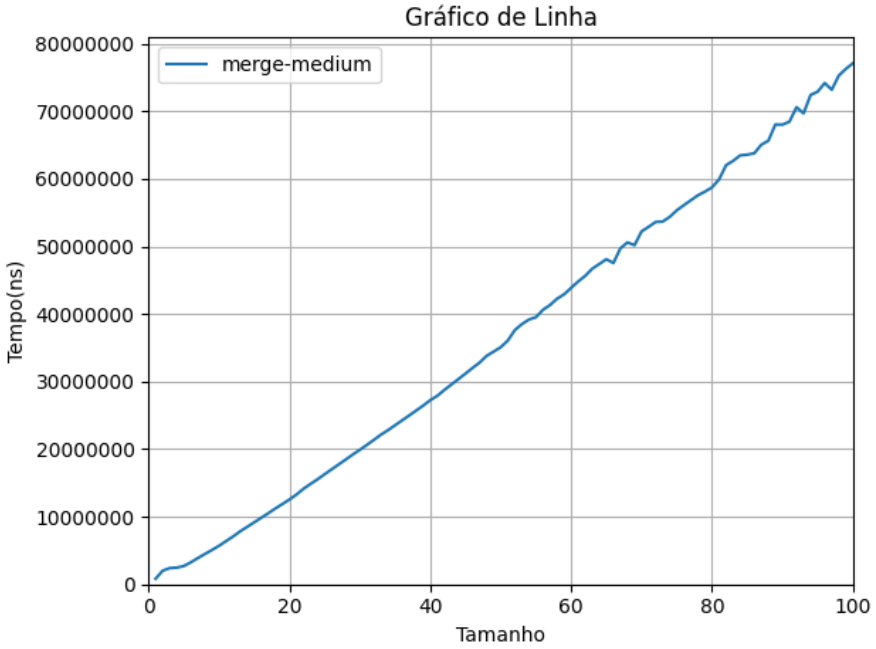

Figura 3.5: Tempo de execução do Merge: (tamanho do vetor x tempo) .

$$T(\frac{n-1}{2}) = a + 2 * T_q(\frac{n-3}{4}) + T_p(\frac{n-1}{2}) \tag{3.48}$$

2ºSubstituindo na equação 3.46:

$$T(n) = a + 2 * [a + 2 * T_q(\frac{n-3}{4}) + T_p(\frac{n-1}{2})] + T_p(n) \tag{3.49}$$

$$T(n) = 3a + 4 * T_q(\frac{n-3}{4}) + 2 * T_p(\frac{n-1}{2}) + T_p(n) \tag{3.50}$$

3ºSubstituindo "n"por "n - 3/4"na equação 3.46:

$$T(\frac{n-3}{4}) = a + 2 * T_q(\frac{n-7}{8}) + T_p(\frac{n-3}{4}) \tag{3.51}$$

4ºSubstituindo na equação 3.50:

$$T(n) = 3a + 4 * [a + 2 * T_q(\frac{n-7}{8}) + T_p(\frac{n-3}{4})] + 2 * T_p(\frac{n-1}{2}) + T_p(n) \tag{3.52}$$

$$T(n) = 7a + 8 * T_q(\frac{n-7}{8}) + 4 * T_p(\frac{n-3}{4}) + 2 * T_p(\frac{n-1}{2}) + T_p(n) \tag{3.53}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = (2^x - 1) * a + 2^x * T_q(\frac{n-(2^x-1)}{2^x}) + 2^{x-1} * T_p(\frac{n-(2^{x-1}-1)}{2^{x-1}}) ... + 2^0 * T_p(\frac{n-(2^0-1)}{2^0}) \tag{3.54}$$

$$T(n) = (2^x - 1) * a + 2^x * T_q(\frac{n - (2^x - 1)}{2^x}) + \sum_{i=0}^{x-1} 2^i * T_p(\frac{n - (2^i - 1)}{2^i}) \tag{3.55}$$

Temos que saber quando a esse somatório acaba, logo, devemos igualar a recorrência ao caso base:

$$\frac{n - (2^x - 1)}{2^x} = 1 \tag{3.56}$$

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

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

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:

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

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

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

$$x = \log_2 \frac{n+1}{2} \tag{3.62}$$

Substituindo na equação 3.55:

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

Simplificando teremos:

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * T_q(1) + \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * T_p(\frac{n - (2^i - 1)}{2^i}) \tag{3.64}$$

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * (c_{22} + c_{24}) + \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i * T_p(\frac{n - (2^i - 1)}{2^i}) \tag{3.65}$$

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

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

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

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

Podemos dividir em:

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

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

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

$$\sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} c * 2^i \tag{3.72}$$

Como já foi observado no somatório da equação 3.39 o somatório 3.69 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:

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

O mesmo vale para o somatório da equação 3.71:

$$(\log_2 \frac{n+1}{2}) * b \tag{3.74}$$

Para o somatório da equação 3.70 a constante b sai, e desenvolvemos o somatório restante:

$$c * \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i \tag{3.75}$$

$$c * (\frac{n-1}{2}) \tag{3.76}$$

Mesma coisa para a equação 3.72:

$$-b * \sum_{i=0}^{(\log_2 \frac{n+1}{2})-1} 2^i \tag{3.77}$$

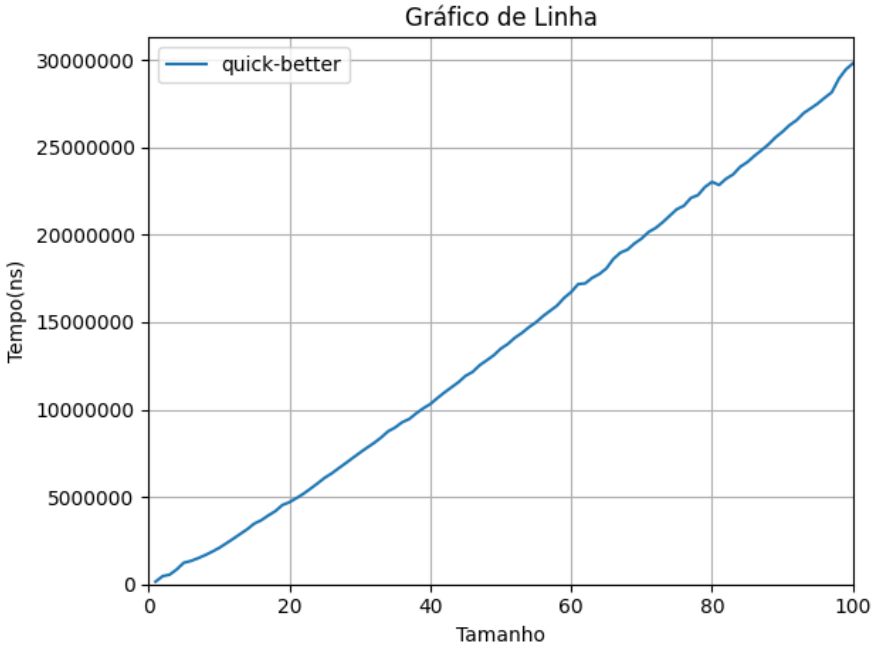

Figura 3.6: Tempo de execução do Merge: (tamanho do vetor x tempo) .

$$-b * (\frac{n-1}{2}) \tag{3.78}$$

Juntando tudo:

$$T(n) = (\frac{n+1}{2} - 1) * a + \frac{n+1}{2} * (c_{22} + c_{24}) + (\log_2 \frac{n+1}{2}) * b * n + c * (\frac{n-1}{2}) - b * (\frac{n-1}{2}) \tag{3.79}$$

Com essa equação podemos concluir que a ordem do tempo de execução do melhor caso é $\Theta(n * \log_2 n)$. Podendo ser observado no gráfico (Veja a figura 3.6) produzido usando o código apresentado.

3.4.2 Pior caso Quick sort

No pior caso o pivô sempre ficará em uma das pontas, resultando na equação 3.81:

$$T(n) = c_1 + c_2 + T_q(n-1) + T_q(0) + T_p(n) \tag{3.80}$$

$$T(n) = a + T_q(n-1) + T_p(n) \tag{3.81}$$

1ºSubstituindo "n"por "n - 1"na equação 3.81:

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

2ºSubstituindo na equação 3.81:

$$T(n) = 2 * a + T^q(n-2) + T^p(n-1) + T^p(n) \tag{3.83}$$

3ºSubstituindo "n"por "n - 2"na equação 3.81:

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

4ºSubstituindo na equação 3.84:

$$T(n) = 3 * a + T^q(n-3) + T^p(n-2) + T^p(n-1) + T^p(n) \tag{3.85}$$

Podemos notar que existe o seguinte padrão:

$$T(n) = x * a + T^q(n-x) + T^p(n-x-1) + T^p(n-x-2) + ... + T^p(n) \tag{3.86}$$

$$T(n) = x * a + T^q(n-x) + \sum_{i=0}^{x-1} T^p(n-i) \tag{3.87}$$

Temos que saber quando a esse somatório acaba, logo, devemos igualar a recorrência ao caso base:

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

$$x = n - 1 \tag{3.89}$$

Substituindo na equação 3.87:

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

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

$$T(n) = (n-1) * a + (c_{22} + c_{24}) + \sum_{i=0}^{n} T^p(n-i) \tag{3.92}$$

Desenvolvendo o somatório, lembrando que o T_m é igual an + b, mas para não conflitar o "a"que já usamos, usaremos bn + c.

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

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

Separando teremos:

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

$$\sum_{i=0}^{n} -b * i \tag{3.96}$$

$$\sum_{i=0}^{n} c \tag{3.97}$$

Desenvolvendo o somatório 3.95:

$$n * (b * n) \tag{3.98}$$

$$n * b + n^2 \tag{3.99}$$

Desenvolvendo o somatório 3.96:

$$-b * (\sum_{i=0}^{n} i) \tag{3.100}$$

$$-b * (\frac{n}{2} * (n + 1)) \tag{3.101}$$

$$-b * (\frac{n^2 + n}{2}) \tag{3.102}$$

Desenvolvendo o somatório 3.97:

$$n * c \tag{3.103}$$

Juntando tudo:

$$T(n) = (n - 1) * a + (c_{22} + c_{24}) + n * b + n^2 - b * (\frac{n^2 + n}{2}) + n * c \tag{3.104}$$

Com essa equação podemos concluir que a ordem do tempo de execução do pior caso é $\Theta(n^2)$.

3.4.3 Caso esperado Quick sort

O caso esperado do quick sort estará mais próximo do seu melhor caso, sendo $\Theta(n * \log_2 n)$. Podendo ser observado no gráfico (Veja a figura 3.7) produzido usando o código apresentado.

3.5 Distribution sort

A ordem do tempo de execução do Distribution sort sempre será linear, mas diferente dos algoritmos visto, ele não depende só de n, mas também de k, que é a diferença entre o maior número no vetor e o menor, sendo então de ordem $\Theta(n + k)$.

1ºPegando as primeiras linhas constantes.

Figura 3.7: Tempo de execução do Quick em todos os casos (tamanho do vetor x tempo) .

$$c_1 + c_2 = a \tag{3.105}$$

2ºA função max e min são lineares, logo elas são an + b.

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

$$2 * a * n + 2 * b \tag{3.107}$$

3ºPrimeiro for. Obs: k é b - s + 2, sendo o tamanho do vetor auxiliar.

$$c_3 * (k + 1) + c_4 * k \tag{3.108}$$

$$c_3 + c_3 * k + c_4 * k \tag{3.109}$$

4ºSegundo for.

$$c_5 * (n + 1) + c_6 * n \tag{3.110}$$

$$c_5 + c_5 * n + c_6 * n \tag{3.111}$$

5ºTerceiro for.

$$c_7 * k + c_8 * (k - 1) \tag{3.112}$$

$$c_7 * k + c_8 * k - c_8 \tag{3.113}$$

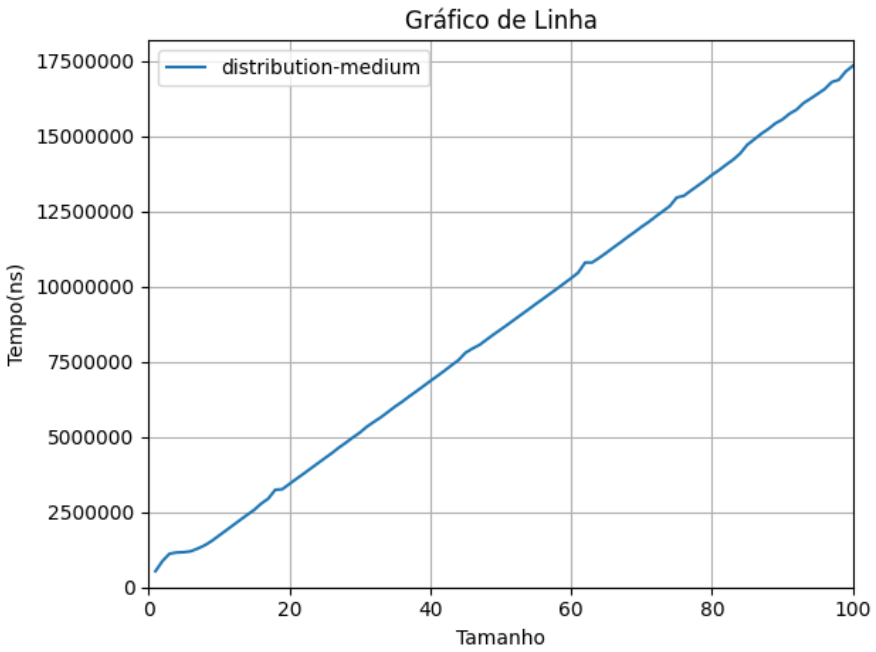

Figura 3.8: Tempo de execução do Distribution (tamanho do vetor x tempo).

6ºQuarto for.

$$c_9 * (n+1) + (c_{10} + c_{11} + c_{12}) * n \tag{3.114}$$

$$c_9 + c_9 * n + (c_{10} + c_{11} + c_{12}) * n \tag{3.115}$$

7ºQuinto for.

$$c_{13} * (n+1) + c_{14} * n \tag{3.116}$$

$$c_{13} + c_{13} * n + c_{14} * n \tag{3.117}$$

Juntando tudo:

$$T(n,k) = 2*a*n+c_3*k+c_4*k+c_5*n+c_6*n+c_7*k+c_8*k+c_9*n+(c_{10}+c_{11}+c_{12})*n+c_{13}*n+c_{14}*n \tag{3.118}$$

$$T(n,k) = c+(2*a+c_5+c_6+c_9+c_{10}+c_{11}+c_{12}+c_{13}+c_{14})*n+(c_3+c_4+c_7+c_8)*k \tag{3.119}$$

Com essa equação podemos concluir que a ordem do tempo de execução do Distribution sort é $\Theta(n+k)$. Podendo ser observado no gráfico (Veja a figura 3.8) produzido usando o código apresentado 2.5.

4. Comparação entre algoritmos

Aqui iremos comparar o tempo de execução dos algoritimos usando os gráficos e vamos eliminando os menos eficientes.

4.1 Todos vs todos

Observando o gráfico com todos os tempos 4.1, podemos perceber com mais clareza a superioridade em eficiência de certos algoritmos sobre outros, por exemplo, fica claro que o selection sort é de longe o pior de todos.

4.2 Insertion vs Merge vs Quick vs Distribution

Como podemos ver no gráfico 4.1, os mais lentos são o pior caso do Quick, e o Selection, removendo-os, sobra o gráfico 4.2

4.3 Merge vs Quick vs Distribution

Como podemos ver no gráfico 4.2, iremos eliminar o Insertion por causa que os piores tempos são seu pior e caso médio, fazendo essa alteração sobra o gráfico 4.3.

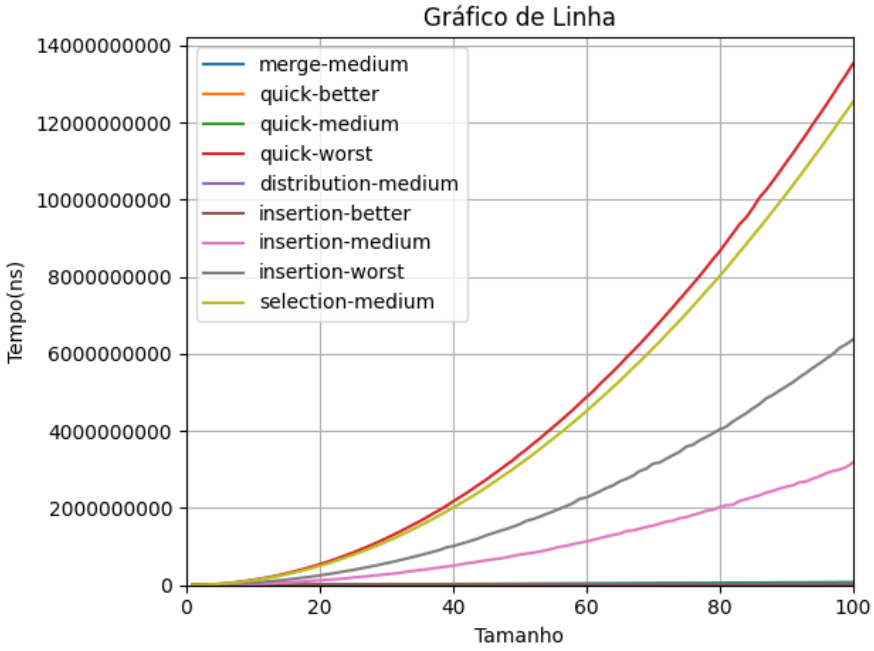

Figura 4.1: Todos vs todos (tamanho do vetor x tempo) .

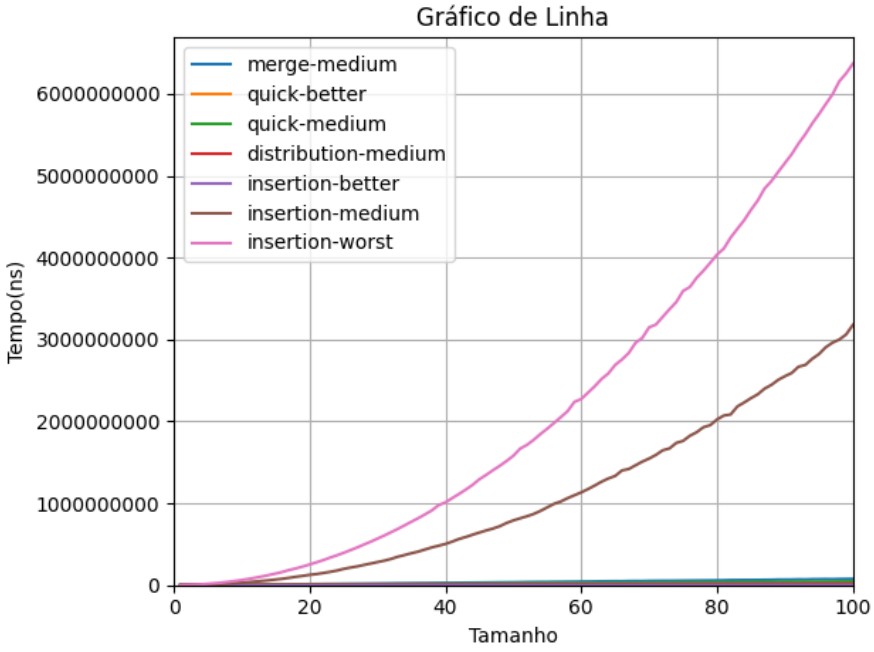

Figura 4.2: Insertion vs Merge vs Quick vs Distribution (tamanho do vetor x tempo).

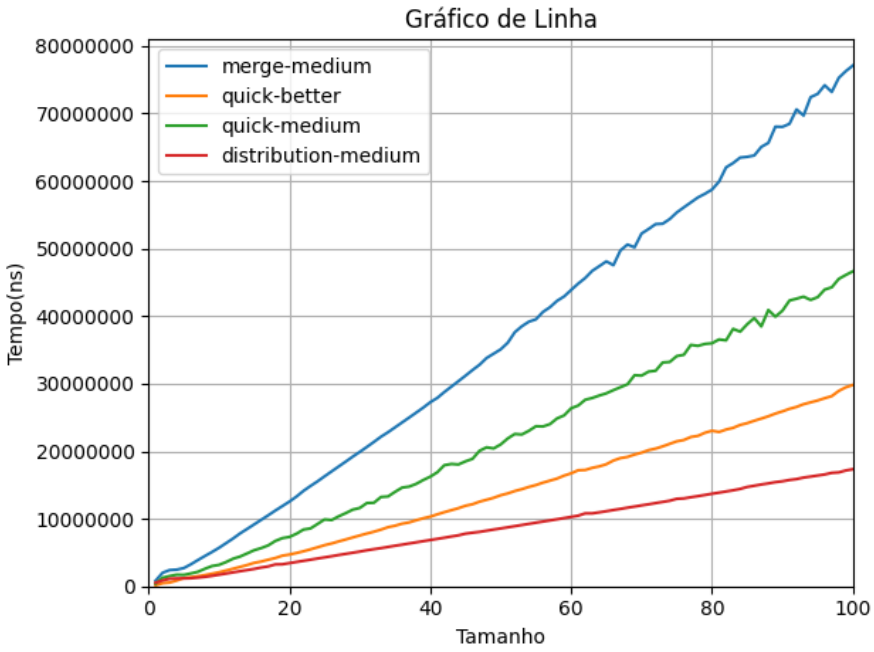

Figura 4.3: Tempo de execução do Merge vs Quick vs Distribution. (tamanho do vetor x tempo).

5. Conclusão

Ao longo deste trabalho, foram explorados os algoritmos de ordenação expostos pelo professor.

No fim das contas, da para perceber que os algoritmos possuem comportamentos e níveis de complexidade de implementação únicos, e a eficiência deles depende de como o vetor está organizado inicialmente, por exemplo, o Quick sort é muito rápido no seu melhor e médio caso, mas seu pior caso é péssimo.

Observando o gráfico 4.3, podemos ver com mais precisão a velocidade dos melhores algoritmos a longo prazo e concluimos que o mais rápido acaba sendo o distribution. Apesar de ter apenas o caso médio, ele performa bem, porém, vale lembrar que ele não é um algoritmo in-place, o que significa que é necessário mais memória da máquina.

Dito isso, o melhor algoritmo depende de cada caso, desde o nível das habilidades do programador até os recursos disponíveis na máquina alvo. Dito isso, aprender sobre cada um deles é bastante importante para conseguirmos implementar o melhor, da melhor forma dependendo do contexto.