

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE - UFRN
CENTRO DE ENSINO SUPERIOR DO SERIDÓ - CERES
BACHARELADO EM SISTEMAS DE INFORMAÇÃO - BSI

JHONATAS ISRAEL DA COSTA LAURENTINO

Algoritmos de Busca e Ordenação

Um Relatório Sobre Os Alguns dos Principais Algoritmos de Busca e
Ordenação Vistos em Estrutura de Dados

CAICÓ/RN
2019

JHONATAS ISRAEL DA COSTA LAURENTINO

Algoritmos de Busca e Ordenação

Um Relatório Sobre Os Alguns dos Principais Algoritmos de Busca e
Ordenação Vistos em Estrutura de Dados

Relatório entregue à matéria Estrutura de Dados, no curso de Bacharelado em Sistemas de Informação, matéria está ministrada pelo Prof. Dr. João Paulo de Souza Medeiros, na Universidade Federal do Rio Grande do Norte (UFRN).

CAICÓ/RN
2019

Resumo

O presente relatório tem como objetivo apresentar estudos coletados durante a primeira unidade da disciplina de Estrutura de Dado. Neste relatório, serão apresentados os algoritmos de ordenação **Insertion Sort**, **Merge Sort**, **Distribution Sort** e **Quick Sort**, além de alguns Algoritmos de Busca. Para implementação dos códigos foi utilizado o sistema operacional Linux Ubuntu, a linguagem de programação C, o compilador GCC, além da ferramenta GNUPLOT para criação dos gráficos.

Abstract

This report aims to present studies collected during the first unit of the Data Structure discipline. In this report, we will introduce the ordering algorithms **Insertion Sort**, **Merge Sort**, **Distribution Sort** and **Quick Sort**, as well as some Search Algorithms. To implement the codes was used the Linux operating system Ubuntu, the programming language C, the GCC compiler, besides the GNUPLOT tool for creating the graphics.

1 Introdução aos Algoritmos de Ordenação	5
1.1 Algoritmos Quadráticos	5
1.2 Algoritmos Lineares	5
1.3 Algoritmos Logarítmicos	5
2 Análise dos algoritmos de ordenação e busca	6
2.1 Big O, Big Omega e Big Theta	6
2.2 Complexidade de um Algoritmo	7
3 Algoritmos de Busca	8
3.1 Busca Linear	8
3.2 Busca Binária	11
3.3 Comparaçao	15
4 Algoritmos de Ordenação	16
4.1 Insertion Sort	16
4.2 Merge Sort	20
4.3 Quick Sort	24
4.4 Distribution Sort	30
4.5 Comparaçao	32
5. Conclusão	33
6. Referências	34

1 Introdução aos Algoritmos de Ordenação

Em computação, **Ordenação** é o ato de colocar os elementos de uma sequência de informações, ou dados, em uma relação de ordem predefinida. O termo técnico em inglês para ordenação é *sorting*, cuja tradução literal é "classificação". Por causa disso, grande maioria dos **Algoritmos de Ordenação (AO)** possuem uma terminação com *Sort*, como por exemplo, o *Insertion Sort*.

Um AO coloca os elementos de uma dada sequência em uma certa ordem, ou em outras palavras, como o próprio nome diz, efetua sua ordenação completa ou parcial, geralmente são utilizadas as ordens numérica ou lexicográfica.

Algumas ordens são facilmente definidas. Por exemplo, a ordem numérica, ou a ordem alfabética -- crescentes ou decrescentes. Contudo, existem ordens, especialmente de dados compostos, que podem ser não triviais de se estabelecer.

Mas pra que realizar a ordenação de uma sequência? Uma delas é a possibilidade de se acessar seus dados de modo mais eficiente.

Entre os mais importantes, podemos citar **Bubble Sort** (ou ordenação por flutuação), **Heap Sort** (ou ordenação por *heap*), **Insertion Sort** (ou ordenação por inserção), **Merge Sort** (ou ordenação por mistura) e o **QuickSort**. Existem diversos outros, que o aluno pode com dedicação pesquisar por si. Porém, neste relatório serão concentrados os estudos nos: **Insertion Sort**, **Merge Sort**, **Distribution Sort** e **Quick Sort**, assim como uma comparação de desempenho entre eles, e será utilizada a ferramenta **GNUPLOT** para criar os gráficos.

1.1 Algoritmos Quadráticos

Em matemática, uma função é de Ordem quadrática (ou ainda, que apresenta crescimento quadrático) quando os valores de seu resultado são proporcionais ao quadrado do valor do seu argumento (comumente representado por x). Na Notação O: $f(x) = O(x^2)$.

São exemplos de funções que apresentam crescimento quadrático toda equação polinomial de grau 2. Por exemplo: $f(x) = 3x^2 + 5x$

Diz-se também que um algoritmo é de ordem quadrática quando a função que descreve sua complexidade é quadrática. Nesse caso o algoritmo é classificado como $O(n^2)$, isso acontece por conta dos itens de dados serem processados aos pares, muitas vezes em uma repetição dentro da outra. Por exemplo, dado n igual à mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos.

1.2 Algoritmos Lineares

Em matemática, problemas de Programação Linear (PL) são problemas de otimização nos quais a função objetivo e as restrições são todas lineares. A PL é uma importante área da otimização por várias razões.

Muitos problemas práticos em pesquisa operacional podem ser expressos como problemas de programação linear. Certos casos especiais dessa natureza, tais como problemas de *network flow* e problemas de *multicommodity flow* são considerados importantes o suficiente para que se tenha gerado muita pesquisa em algoritmos especializados para suas soluções.

Vários algoritmos para outros tipos de problemas de otimização funcionam resolvendo problemas de PL como sub-problemas. Historicamente, ideias da programação linear inspiraram muitos dos conceitos centrais de teoria da otimização, tais como dualidade, decomposição, e a importância da convexidade e suas generalizações.

Na computação, os AOs que possuem complexidade linear são representados por $\Theta(n)$, ou seja, complexidade algorítmica em que um pequeno trabalho é realizado sobre cada elemento da entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho o tempo de execução dobra.

1.3 Algoritmos Logarítmicos

Na matemática, o logaritmo de um número é o expoente a que outro valor fixo, a base, deve ser elevado para produzir este número. Por exemplo, o logaritmo de 1000 na base 10 é 3 porque 10 ao cubo é 1000 ($1000 = 10 \times 10 \times 10 = 10^3$). De maneira geral, para quaisquer dois números reais b e x , onde b é positivo e $b \neq 1$,

$$y = b^x \Leftrightarrow x = \log_b(y)$$

O logaritmo da base 10 ($b = 10$) é chamado de logaritmo comum (ou decimal) e tem diversas aplicações na ciência e engenharia. O logaritmo natural (ou neperiano) tem a constante irracional e ($\approx 2,718$) como base e é utilizado na matemática pura, principalmente em cálculo diferencial. Ainda há o logaritmo binário, no qual se usa base 2 ($b = 2$), que é importante para a ciência da computação.

Na computação é representada por $O(\log_n)$. Complexidade algorítmica no qual algoritmo resolve um problema transformando-o em partes menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Por exemplo, quando n é um milhão, $\log_2 n$ é aproximadamente 20.

2 Análise dos algoritmos de ordenação e busca

Em termos mais práticos, é uma forma de julgarmos se o nosso algoritmo é eficiente, independente dos “recursos que o cercam” (como velocidade de processamento, quantidade de memória, latência de rede, etc).

Removendo todas as variáveis que podem influenciar no tempo de execução, focamos nossas atenções em como o algoritmo está escrito, em qual é a sua entrada, e se “ele por si” é a maneira mais eficiente para a resolução de um determinado problema.

Vale reforçar que a entrada é um fator de extrema importância no que tange a análise assintótica. A análise é “*input bound*”, ou seja, a entrada influenciará diretamente no resultado do estudo. Por exemplo, quando ordenamos um vetor de tamanho N , utilizando o algoritmo *Selection Sort*, teremos um tempo de execução de N^2 (já que o algoritmo pega um número, e compara com os demais números no vetor, repetindo essa operação até chegar ao fim do dado estruturado).

Ao fim da análise, podemos chegar a 3 conclusões diferentes: Melhor Caso, Médio Caso e Pior Caso, como poderemos entender melhor na seção 2.2 Complexidade de um Algoritmo, onde falará com mais detalhes sobre os casos citados.

2.1 Big O, Big Omega e Big Theta

Quem trabalha com desenvolvimento (ou até mesmo com computação em geral), já deve ter ouvido falar sobre o famigerado Big O Notation. Ele é uma notação assintótica muito famosa na análise de tempos de execução de algoritmos. O que pode ser uma surpresa é que ele não é a única notação que temos disponível:

- **O(n)**: Expressa o limite superior do tempo de execução de um algoritmo (pior cenário);
- **$\Omega(n)$** : Expressa o limite inferior do tempo de execução de um algoritmo (melhor cenário);
- **$\Theta(n)$** : Expressa limite superior e inferior do tempo de execução de um algoritmo (pior e melhor cenário).

Além da expressão linear, temos outras notações que descrevem diferentes tempos de execução:

- $O(1)$: Constante
- $O(\log n)$: Logarítmica
- $O(n)$: Linear
- $O(n \log n)$: “*Linear Rithmic*” (maior que linear, menor que quadrática)
- $O(n^2)$: Quadrática
- $O(n^3)$: Cúbica
- $n^{O(1)}$: Polinomial
- $2^{O(n)}$: Exponencial

De maneira simplista, não pode ser considerado como o número de operações que o algoritmo leva para chegar ao seu final. N está intimamente ligado com a entrada do seu algoritmo, onde quanto maior for o seu número, maior será o seu tempo de execução.

2.2 Complexidade de um Algoritmo

A complexidade do algoritmo é dada pela quantidade de vezes que o algoritmo será executado para resolver determinada tarefa. A complexidade é dividida em três níveis diferentes; o **Melhor Caso** onde a entrada mais favorável, que produz o menor número de passos da operação dominante, tem o **Caso Médio** que é o caso mais real, onde depende dos conhecimentos estatísticos sobre o problema, por fim, é obtido também o **Pior Caso**, nele a entrada é menos favorável e possui a maior produção de passos para executar a tarefa.

Pode-se utilizar o exemplo do *Insertion Sort* e que é bem comum com muitos algoritmos, nele a sua melhor entrada seria se o vetor estivesse com todos os valores colocados

em ordem crescente, enquanto seu pior será no caso do vetor não possuir o item no vetor ou então ele estiver ordenado em ordem inversa ao que está sendo pedido.

3 Algoritmos de Busca

Na computação, um **Algoritmo de Busca** (AB), em termos gerais é um algoritmo que toma um problema como entrada e retorna a solução para o problema, geralmente após resolver um número possível de soluções.

Uma solução, no aspecto de função intermediária, é um método o qual um algoritmo externo, ou mais abrangente, utilizará para solucionar um determinado problema. Esta solução é representada por elementos de um espaço de busca, definido por uma fórmula matemática ou um procedimento, tal como as raízes de uma equação com números inteiros variáveis, ou uma combinação dos dois, como os circuitos hamiltonianos de um grafo.

Já pelo aspecto de uma estrutura de dados, sendo o modelo de explanação inicial do assunto, um AB é projetado para encontrar um item com propriedades especificadas em uma coleção de itens. Os itens podem ser armazenadas individualmente, como registros em um banco de dados.

A maioria dos algoritmos estudados por cientistas da computação que resolvem problemas são ABs, neste relatório será mostrado o funcionamento de dois ABs o de Busca Linear e o de Busca Binária.

3.1 Busca Linear

O algoritmo de Busca Linear que também pode ser chamado de Busca Sequencial, percorre um vetor de forma sequencial até encontrar o valor especificado, portanto o tempo de execução cresce em proporção ao número de elementos no vetor sendo assim um algoritmo linear n .

Ele funciona da seguinte maneira, o **Algoritmo 3.1.1** exemplificado logo abaixo, possui como parâmetros três entradas, onde v é o vetor, n é o tamanho e x é valor que está sendo procurado. Como não é uma busca linear, ele não vai saber se o vetor v está ordenado, por causa disso é inserido o comando **for** e logo depois vai comparar o elemento x com o valor que está em $v[i]$, se for igual é porque o valor que está sendo buscado foi encontrado.

1	5	-2	3	4	v
---	---	----	---	---	-----

Figura 3.1.1 Vetor v

```
1 - int search(int *v, int n, int x){  
2 -     int i;  
3 -     for (i = 0; i < n; i++){  
4 -         if(v[i] == x){  
5 -             return i;  
6 -         }  
7 -     }  
8 -     return -1;
```

Algoritmo 3.1.1 Algoritmo de Busca Linear em C

A **Figura 3.1.2** mostra como seria a busca pelo elemento **3** em um vetor desordenado de **5** posições no vetor **v**.

$i = 0$	1	5	-2	3	4	v
$i = 1$	1	5	-2	3	4	v
$i = 2$	1	5	-2	3	4	v
$i = 3$	1	5	-2	3	4	v

Figura 3.1.2 Vetor **v** sendo utilizado como exemplo de busca pelo valor **3**

Como o valor **x** que está sendo procurado é o **3**, o algoritmo ficará sendo executado até que o valor de **v[i]** seja igual a **x**, quando isso acontecer ele irá encerrar o laço de repetição.

O **Gráfico 3.1.1** mostra o melhor caso do problema, onde ocorre quando o valor buscado está na primeira posição do vetor. Já o **Gráfico 3.1.2** mostra-nos o pior caso, que é quando o item buscado não está no vetor e o **Gráfico 3.1.3** mostra o caso médio do mesmo problema.

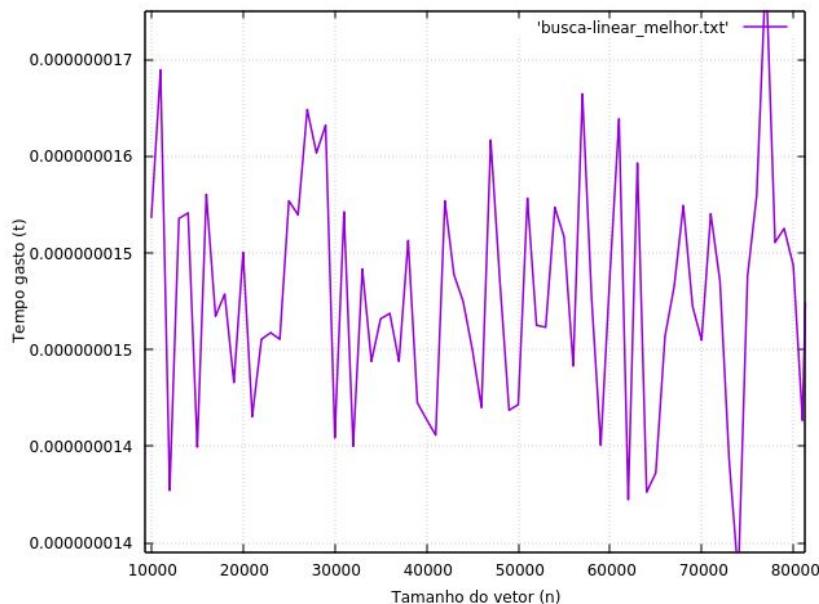


Gráfico 3.1.1 Gráfico do Melhor Caso no Algoritmo de Busca Linear

Sobre a análise da complexidade do algoritmo, como ele possui apenas uma estrutura de repetição, o pior caso acontece quando o número procurado é o último elemento do vetor ou então ele não está no vetor, realizando assim n comparações. Logo, o tempo de execução é $T(n)$

$= O(n)$. O seu cálculo é mostrado na **Figura 3.1.3** é a equação encontrada a respeito de tal acontecimento.

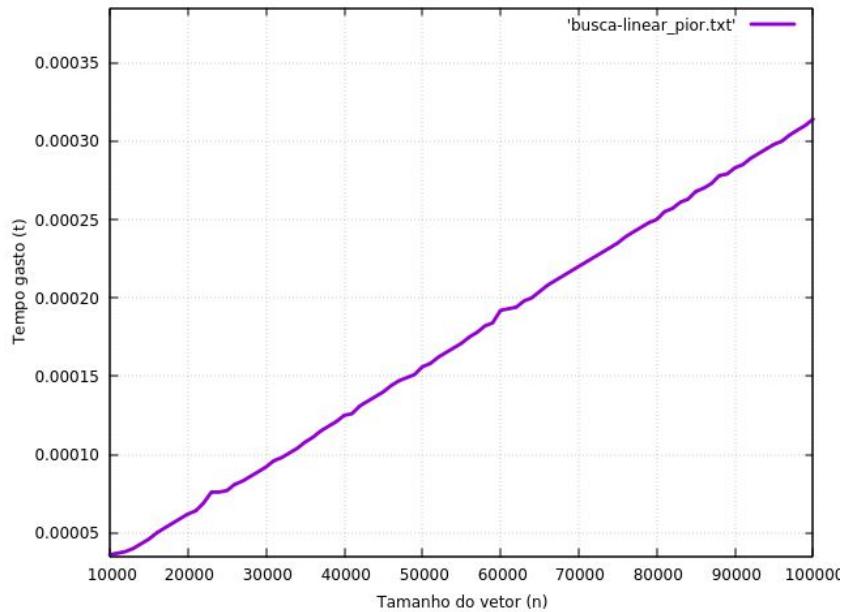


Gráfico 3.1.2 Gráfico do Pior Caso no Algoritmo de Busca Linear

$$\begin{aligned} T(n) &= C_1 + (n + 1)C_2 + nC_3 + C_5 = \\ &n(C_2 + C_3) + C_1 + C_2 + C_5 \end{aligned}$$

Figura 3.1.3 Equação do Pior Caso no Algoritmo de Busca Linear

Já o Melhor Caso ocorre quando o número procurado está na primeira posição do vetor, sendo comparado uma única vez e cujo tempo de execução é constante, ou seja, $T(n) = O(1)$.

$$T(n) = C_1 + C_2 + C_3 + C_4$$

Figura 3.1.4 Equação do Melhor Caso no Algoritmo de Busca Linear

Para o caso médio esperado do algoritmo de busca linear o vetor foi preenchido de forma aleatória, desta maneira o elemento pode estar em qualquer posição do vetor, variando o tempo de execução.

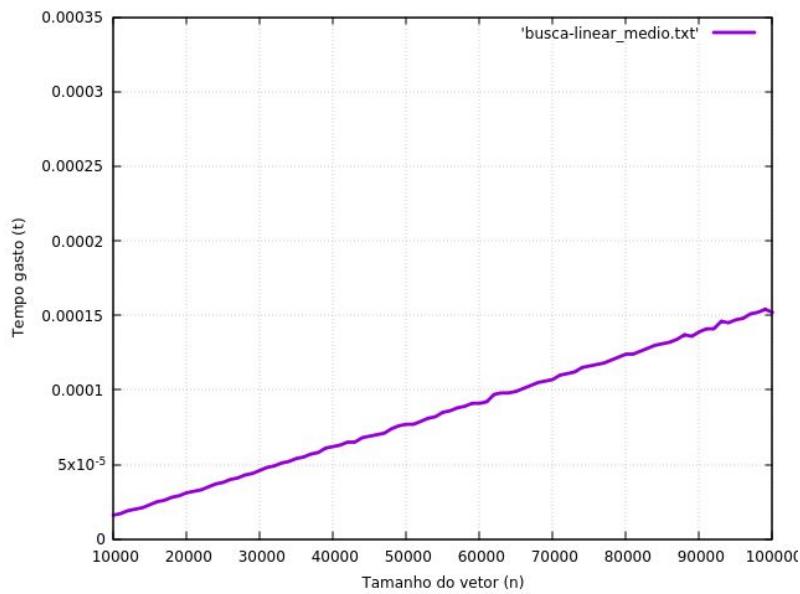


Gráfico 3.1.3 Gráfico do Caso Médio no Algoritmo de Busca Linear

Por fim, é possível observar no **Gráfico 3.1.4** a comparação entre os três casos do algoritmo de Busca Linear.

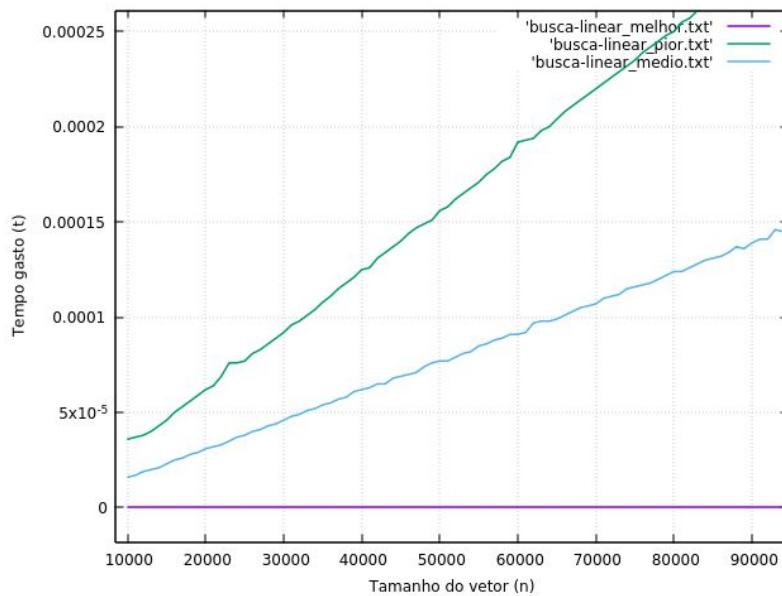


Gráfico 3.1.4 Gráfico de Comparação do Algoritmo de Busca Linear

3.2 Busca Binária

Chama-se busca binária o algoritmo de busca que segue o paradigma de divisão e conquista de ordem logarítmica \log .

```

1 - int search(int *v, int s, int e, int k){
2 -     int m = (s+e)/2;

```

```

3 -     if(s <= e){
4 -         if(v[m] == k){
5 -             return m;
6 -         }else if(v[m] > k){
7 -             search(v, s, m-1, k);
8 -         }else{
9 -             search(v, m+1, e, k);
10-        }
11-    }else{
12-        return -1;
13-    }
14- }
```

Algoritmo 3.2.1 Algoritmo de Busca Binaria em C

Ele é somente executado em vetores ordenados, nesse algoritmo o vetor é dividido ao meio e o número do meio é comparado com o valor procurado. Se estes forem iguais, a busca termina, caso contrário, se o valor for menor, o valor será procurado no vetor à esquerda ao do meio, se o valor for maior, ele irá buscar no lado direito ao meio do vetor.

Esse procedimento de divisão e comparação deve ocorrer até que o vetor de dados fique com apenas um elemento, ou até que o número procurado seja encontrado.

Assim como a Busca Linear seu melhor caso ocorre quando o elemento buscado é encontrado na primeira comparação, nesse caso, é possível equacionar a complexidade como mostrado na **Figura 3.2.1**.

$$T_b(n) = C_1 + C_2 + C_3 + C_4$$

Figura 3.2.1 Equação do Melhor Caso no Algoritmo de Busca Linear

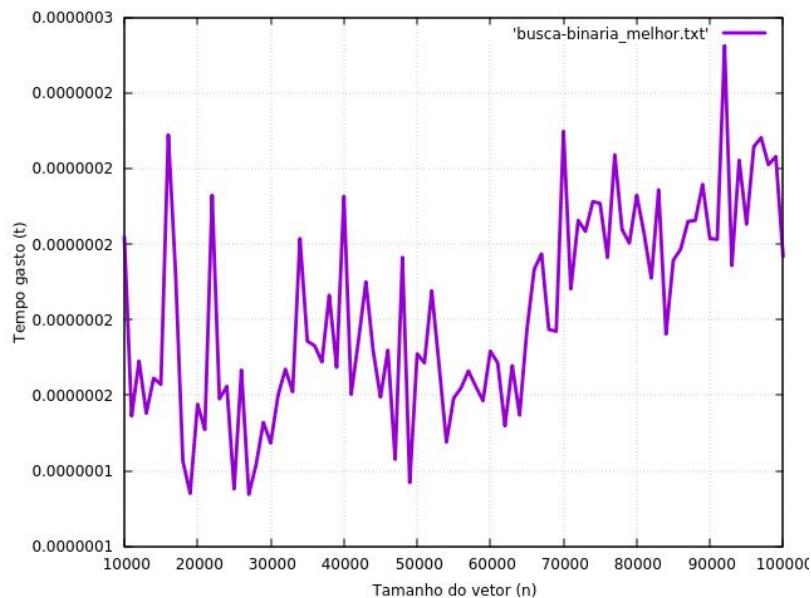


Gráfico 3.2.1 Gráfico de Melhor Caso do Algoritmo de Busca Linear

Embora a ordem seja constante, o tempo de execução varia muito por conta da interferência de outros processos externos rodando na máquina, como é visto no **Gráfico 3.2.1**. O pior caso da busca binária, assim como a linear, ocorre quando o elemento a ser procurado não está presente no vetor, assim, sua ordem é logarítmica e pode ser equacionada como mostrado na **Figura 3.2.2**, onde é possível perceber que é encontrada uma relação de recorrência, que tem como caso base um vetor com 0 posições.

$$T_w(n) = C_1 + C_2 + C_3 + C_{5,6,7,8} + T_w\left(\frac{n-1}{2}\right)$$

$$T_w(0) = C_1 + C_2 + C_3 + C_{10}$$

Por praticidade, será considerado

$$a = C_1 + C_2 + C_3 + C_{5,6,7,8}$$

Então temos

$$T_w(n) = a + T_w\left(\frac{n-1}{2}\right)$$

Calculando a próxima recorrência se tem

$$\begin{aligned} T_w\left(\frac{n-1}{2}\right) &= a + T_w\left(\frac{n-3}{4}\right) \\ T_w(n) &= a + \left[a + T_w\left(\frac{n-3}{4}\right) \right] = \\ &\quad 2a + T_w\left(\frac{n-3}{4}\right) \end{aligned}$$

Calculando a próxima relação se obtém

$$\begin{aligned} T_w\left(\frac{n-3}{4}\right) &= a + T_w\left(\frac{\frac{n-3}{4}-1}{2}\right) = \\ &\quad a + T_w\left(\frac{n-7}{8}\right) \end{aligned}$$

Substituindo na equação original

$$\begin{aligned} T_w(n) &= a + \left[a + \left[a + T_w\left(\frac{n-7}{8}\right) \right] \right] = \\ &\quad 3a + T_w\left(\frac{n-7}{8}\right) \end{aligned}$$

Neste ponto já é possível encontrar uma padrão nas trocas, então será utilizado a variável x para representá-las

$$(x) a + T_w\left(\frac{n - 2^x - 1}{2^x}\right)$$

Agora é necessário encontrar o valor de x que zere a equação para dar fim a recorrência, então

$$\frac{n - (2^x - 1)}{2^x} = 0$$

$$n - 2^x + 1 = 0$$

$$n + 1 = 2^x$$

$$\log_2(n + 1) = \log_2 2^x$$

$$x = \log_2(n + 1)$$

Substituindo na equação original

$$T_w(n) = \log_2(n + 1)a + C_1 + C_2 + C_9 + C_{10}$$

Figura 3.2.2 Equação do Pior Caso no Algoritmo de Busca Linear

Substituindo na equação original é possível, assim provar que sua ordem é logarítmica, o gráfico que mostra o tempo de execução do algoritmo pode ser visto na **Figura 3.2.4**.

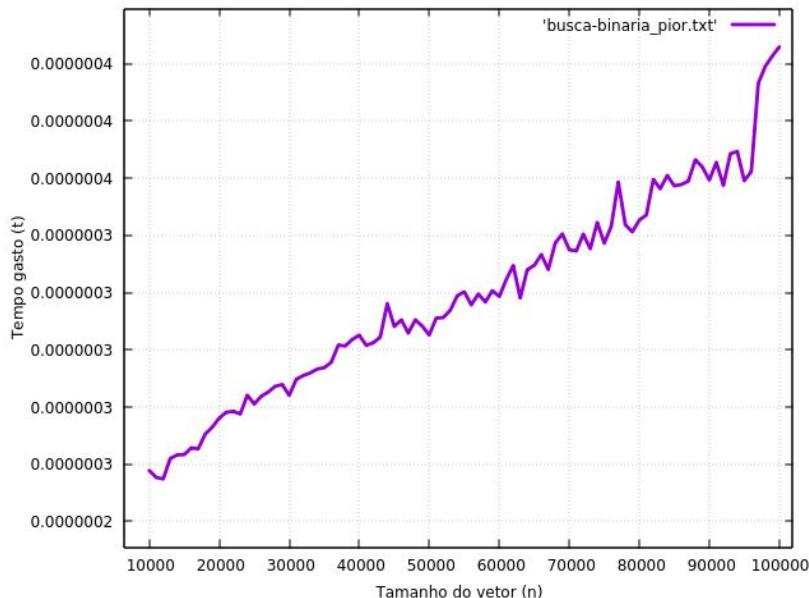


Gráfico 3.2.2 Gráfico de Pior Caso do Algoritmo de Busca Binária

O caso médio esperado do tempo de execução do algoritmo de Busca Binária foi calculado fazendo com que o elemento a ser procurado no vetor seja um número aleatório, presente no mesmo. O gráfico que mostra o tempo de execução neste caso pode ser visto na **Gráfico 3.2.3**.

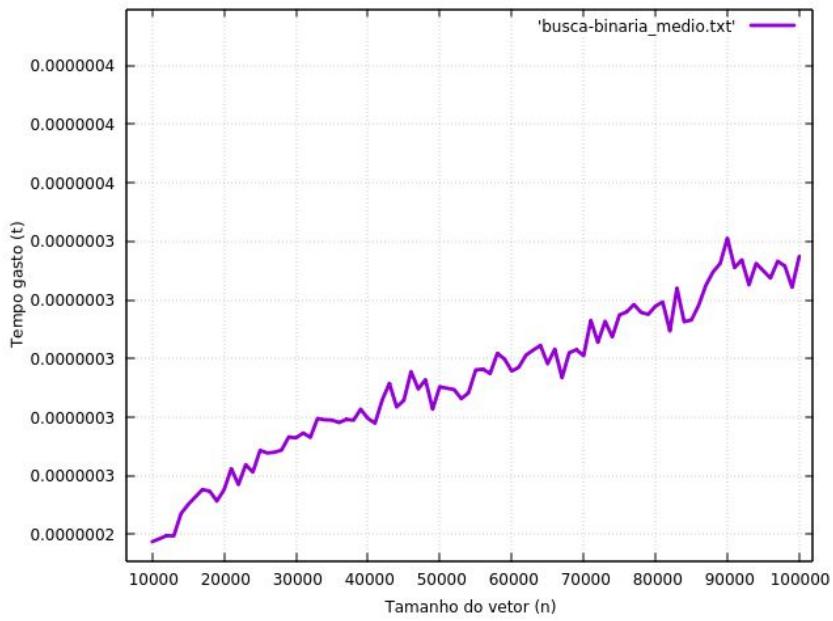


Gráfico 3.2.3 Gráfico de Médio Caso do Algoritmo de Busca Binária

O Gráfico 3.2.4 mostra a comparação entre os casos do algoritmo de Busca Binária

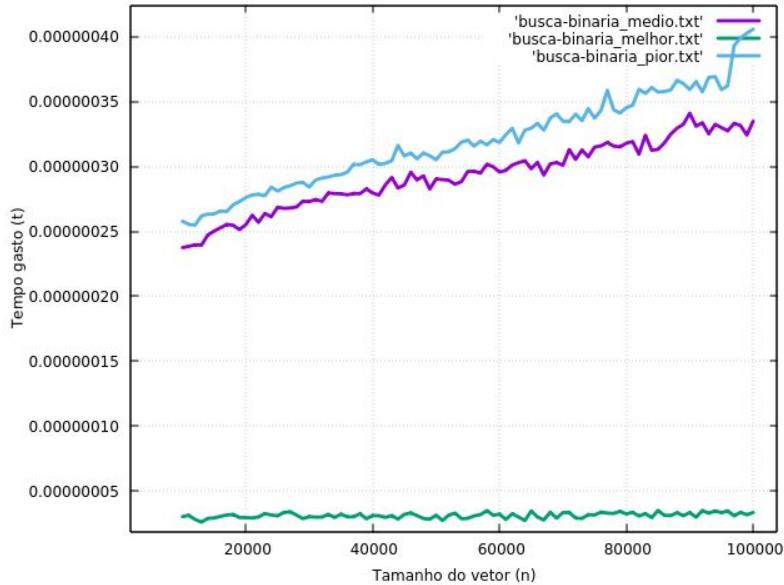


Gráfico 3.2.4 Gráfico da comparação dos casos no Algoritmo de Busca Binária

3.3 Comparação

Como visto nas seções anteriores, é possível entender que a complexidade da busca linear é n enquanto a da busca binária é \log_n . A busca linear em seu melhor caso leva vantagem sobre a busca binária, isso ocorre por conta que a quantidade de verificações é menor na busca linear. Ao contrário do melhor caso que é basicamente constante, o caso médio dos algoritmos já

adquire sua ordem, ou seja, n para linear e \log_n para logarítmica. Assim como no caso médio, o pior caso da busca binária tem tempo de execução muito inferior à busca linear.

4 Algoritmos de Ordenação

4.1 Insertion Sort

Considera que o primeiro elemento está ordenado (ou seja, na posição correta). A partir do segundo elemento, insere os demais elementos na posição apropriada entre aqueles já ordenados. O elemento é inserido na posição adequada movendo-se todos os elementos maiores para posição seguinte do vetor. Mais interessante que o Bubble Sort para popular um vetor.

O elemento da posição 0 (valor 50) é comparado com o elemento da posição 1 (valor 30). Como o objetivo é ordenar crescentemente, os conteúdos dos elementos das posições 0 e 1 devem ser trocados entre si.

0	1	2	3	4
50	30	40	20	10

0	1	2	3	4
30	50	40	20	10

0	1	2	3	4
30	40	50	20	10

Em seguida serão comparados os conteúdos dos elementos das posições 1 e 2 troca elementos das posições 2 e 3.

0	1	2	3	4
30	40	20	50	10

0	1	2	3	4
30	40	20	50	10

Elementos das posições 3 e 4 apesar do vetor não estar ordenado ainda, observe que o maior elemento ficou na última posição:

0	1	2	3	4
30	40	20	10	50
0	1	2	3	4
30	40	20	10	50

0	1	2	3	4
30	40	20	10	50
0	1	2	3	4
30	20	40	10	50

O processo recomeça, porém ocorrerá entre as posições 0 e 3 (o elemento da posição 4 já está ordenado). Não acontece a troca entre os elementos das posições 1 e 2:

0	1	2	3	4
30	20	40	10	50

0	1	2	3	4
30	20	10	40	50

0	1	2	3	4
30	20	10	40	50

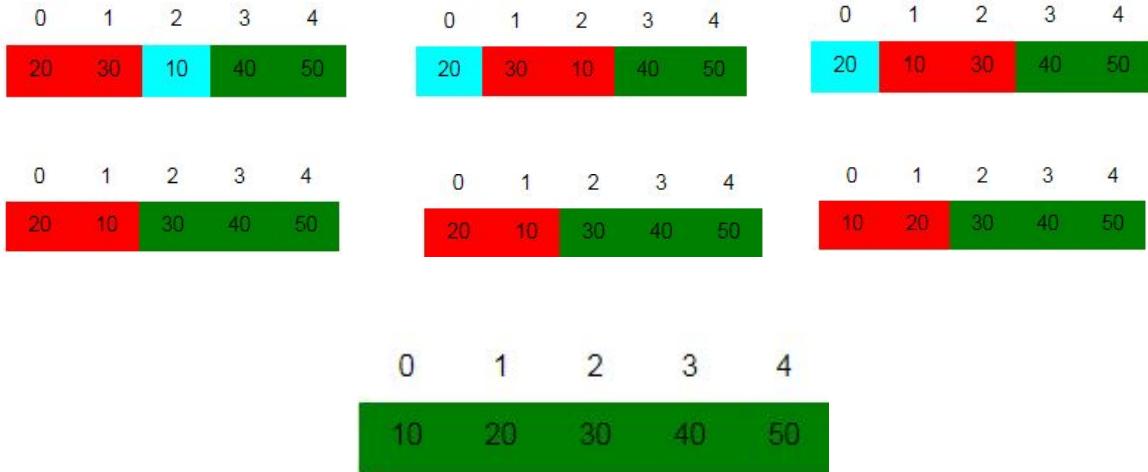


Figura 4.1.1 Procedimento de ordenação no *Insertion Sort*

Com isso, o processo de ordenação termina! Veja o algoritmo a seguir:

```

1 - void ordena (int *vetor, unsigned int n){
2 -     int i, j, atual;
3 -     for (i = 1; i < n; i++) {
4 -         atual = vetor[i];
5 -         for (j = i - 1; (j >= 0) && (atual < vetor[j]); j--) {
6 -             vetor[j+1] = vetor[j];
7 -         }
8 -         vetor[j+1] = atual;
9 -     }
10 - }
```

Algoritmo 4.1.1 Algoritmo de *Insertion Sort* em C

Seu melhor caso ocorre quando o vetor já se encontra ordenado, nesse caso o algoritmo fará o menor número de comparações, a equação que em seu melhor caso é dada por:

$$\begin{aligned}
T_b(n) &= C_1 + nC_2 + (n - 1)C_3 + (n - 1)C_4 + (n - 1)C_6 = \\
&C_1 + nC_2 + nC_3 - C_3 + nC_4 - C_4 + nC_6 - C_6 = \\
&n(C_2 + C_3 + C_4 + C_6) - C_3 - C_4 - C_6 + C_1
\end{aligned}$$

Figura 4.1.2 Equação do Melhor Caso no Algoritmo de Busca Linear

No **Gráfico 4.1.1** é possível visualizar seu tempo de execução em função do vetor de tamanho n que varia entre 1000 e 10000 posições, apesar dos ruídos causados por pequenas variações por conta de processos externos, sua complexidade é $\Theta(n)$, ou seja, linear.

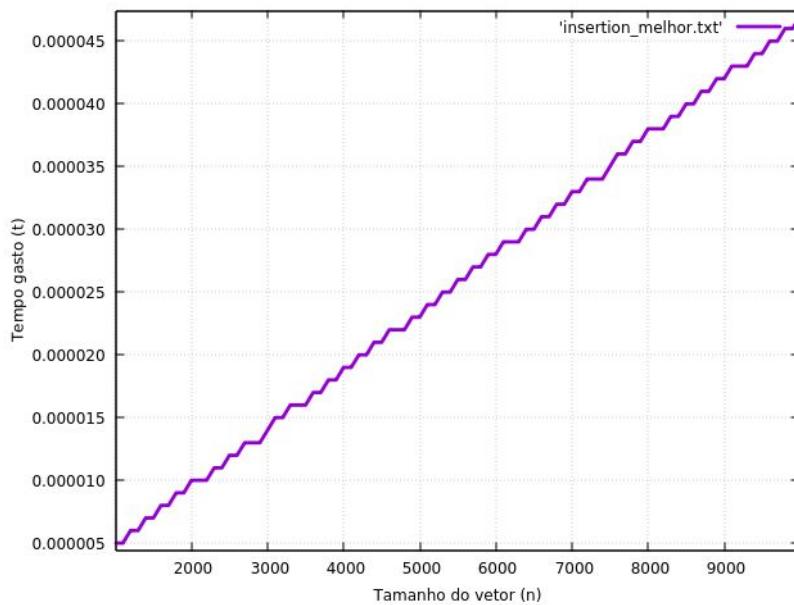


Gráfico 4.1.1 Gráfico de Melhor Caso do Algoritmo de Busca Binária

O caso médio esperado do *Insertion Sort* foi calculado de forma que o vetor de entrada foi preenchido de forma aleatória, ou seja, o algoritmo poderá ou não está ordenado, variando assim o tempo de execução.

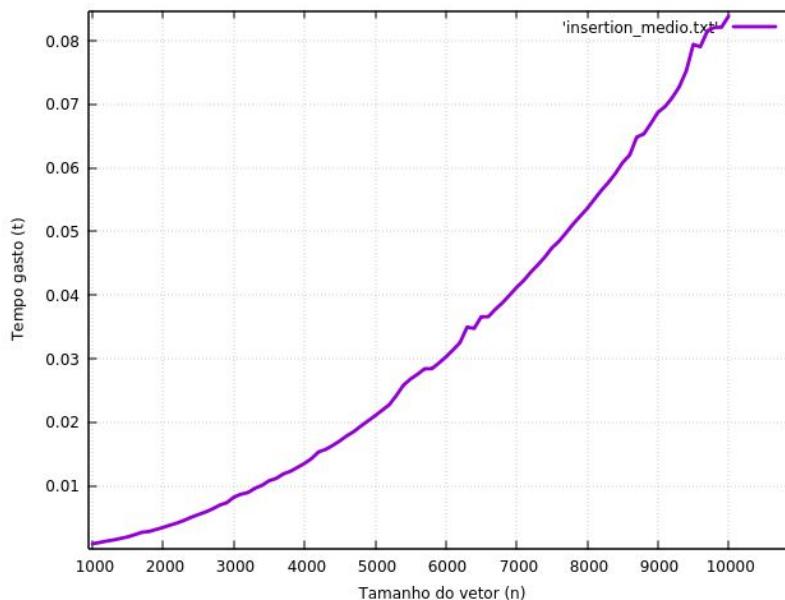


Gráfico 4.1.2 Gráfico do Caso Médio do Algoritmo de Busca Binária

No **Gráfico 4.1.2** é possível visualizar seu tempo de execução em função do vetor de tamanho n que varia entre 1000 e 10000 posições, e tem formato de uma parábola, mostrando assim claramente que seu caso médio tem complexidade quadrática.

O pior caso do *Insertion Sort* ocorre quando o vetor está ordenado em ordem inversa (decrescente), sendo assim, tornando-o de ordem $\Theta(n^2)$, a equação que define sua ordem é dada por:

$$\begin{aligned}
T_w(n) &= C_1 + nC_2 + (n-1)C_3 + C_4 \sum_{i=1}^{n-1} i + C_5 \sum_{i=1}^{n-1} i + (n-1)C_6 + (n-1)C_4 = \\
&= C_1 + nC_2 + nC_3 - C_3 + C_4 \left(\frac{n}{2}(n+1) - n \right) + C_5 + \left(\frac{n}{2}(n+1) - n \right) + C_6n - C_6 + C_4(n-1) = \\
&= C_1 + nC_2 + nC_3 - C_3 + C_4 \left(\frac{n^2+n}{2} - n \right) + C_5 \left(\frac{n^2+n}{2} - n \right) + C_6n - C_6 + C_4(n-1) = \\
&= n(C_2 + C_3 + C_6) + (C_4 + C_5) \left(\frac{n^2+n-2n}{2} \right) + (n-1)C_4 - C_6 + C_1
\end{aligned}$$

Figura 4.1.3 Equação do Pior Caso no Algoritmo *Insertion Sort*

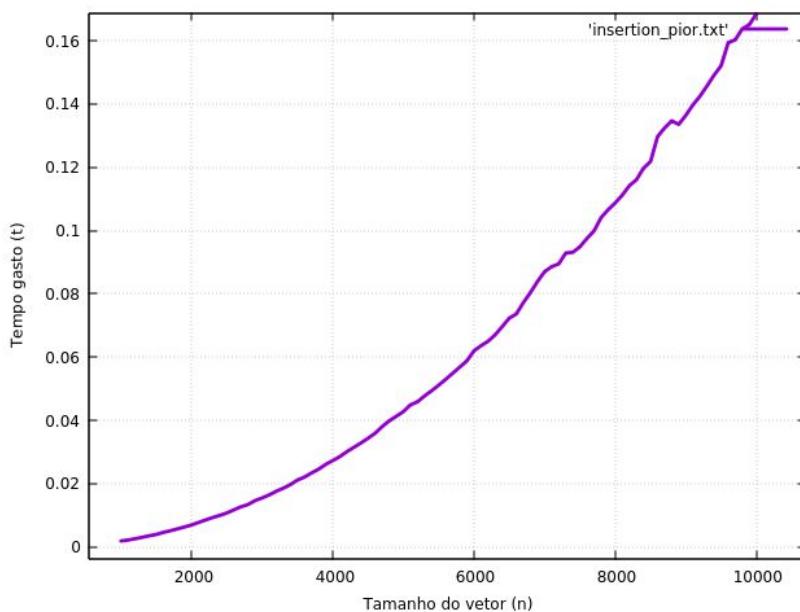


Gráfico 4.1.3 Gráfico do Pior Caso no Algoritmo de *Insertion Sort*

Como visto na equação apresentada graficamente na figura 4, assim como o caso médio, o pior caso do *Insertion Sort* têm complexidade $\Theta(n^2)$, embora seu tempo de execução seja superior.

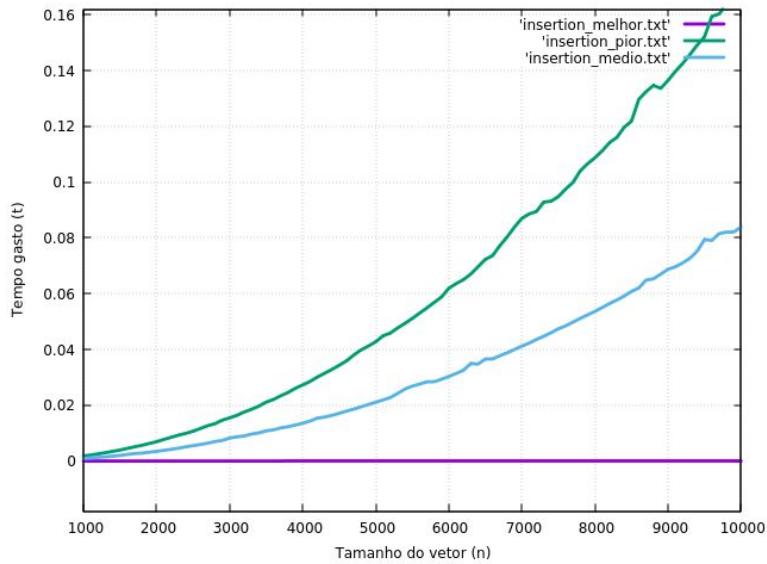


Gráfico 4.1.4 Gráfico de Comparaçāo entre os Algoritmos de Busca Binária

No **Gráfico 4.1.4** é possível visualizar que todos os casos estão de acordo com seu título. É válido lembrar que, em comparação ao médio e pior caso, o melhor caso tem tempo de execução em segundos muito baixo por ser de ordem linear, por isso não é perceptível sua variação.

4.2 Merge Sort

O algoritmo de ordenação *Merge Sort* usa uma técnica de divisão, ou seja, reparte o vetor sucessivamente ao meio até que reste apenas um elemento, onde a partir daí será ordenado e depois intercalado até chegar ao vetor original, porém ordenado. Para fazer a implementação deste algoritmo, é necessário ter conhecimento do tamanho do vetor, bem como seu valor de início e m, para que seja possível dividir o vetor em subvetores, como exemplificado na **Figura 4.2.1**, onde mostra todo o procedimento que ocorre na ordenação do vetor.

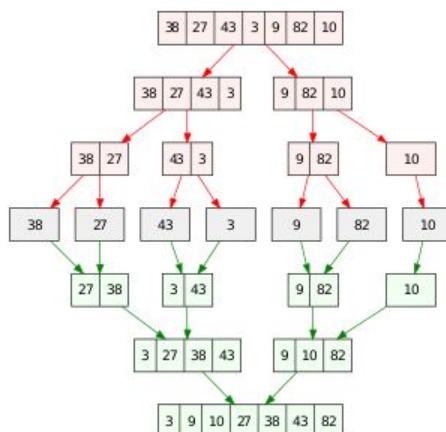


Figura 4.2.1 Procedimento de ordenação do Algoritmo Merge Sort

```

1 - void *merge(int *vetor, int começo, int meio, int fim) {
2 -   int i = começo;
3 -   int j = meio+1, k=0;
4 -   int *aux = (int *) malloc((fim-começo+1) * sizeof(int));
5 -   while (k <= fim-começo){
6 -     if ((vetor[i] < vetor[j] && i <= meio) || (j > fim)){
7 -       aux[k] = vetor[i];
8 -       i++;
9 -     }else{
10 -       aux[k] = vetor[j];
11 -       j++;
12 -     }
13 -     k++;
14 -   }
15 -   k = 0;
16 -   while (k <= fim-começo){
17 -     vetor[começo+k] = aux[k];
18 -     k++;
19 -   }
20 -   free(aux);
21 - }
```

Algoritmo 4.2.1 Algoritmo de *Merge Sort* em C

Esse processamento é possível devido o método de recursão onde o problema inicial é dividido em sub-problemas, transformando-se em problemas simples de serem resolvidos. O algoritmo é composto por duas funções, uma para ordenação mostrado no **Algoritmo 4.2.1** e outra responsável por dividir o vetor em sub-vetores **Algoritmo 4.2.2**.

```

1 - void *mergeSort(int *vetor, int começo, int fim){
2 -   if (começo < fim) {
3 -     int meio = (fim+começo)/2;
4 -     mergeSort(vetor, começo, meio);
5 -     mergeSort(vetor, meio+1, fim);
6 -     merge(vetor, começo, meio, fim);
7 -   }
8 - }
```

Algoritmo 4.2.2 Algoritmo de *Merge Sort* em C

Para obter o tempo de execução do *Merge Sort* primeiramente é necessário ter conhecimento do tempo de execução da função responsável pela divisão do vetor. Não existe casos necessariamente, pois independente da situação dos dados no vetor, o algoritmo irá sempre dividir e intercalar os dados. Logo, o tempo é dado pela altura da árvore de recursão, ou seja, pela quantidade sub-vetores que serão gerados que é dado em $\Theta(\log n)$, e pela quantidade de operações em cada nível da árvore (em cada vetor) que é dado por $\Theta(n)$. De maneira analítica é possível observar $\Theta(n \log n)$. É possível também chegar a essa que a complexidade do

algoritmo é conclusão de forma mais precisa equacionando o problema como mostrado na **Figura 4.2.2.**

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

Por fins de simplicidade, considera-se:

$$a = C_1 + C_2 + C_3 + C_4 + C_5$$

Como visto, é obtida uma relação de recorrência onde é encerrada quando o vetor alcança um valor unitário, ou seja, o algoritmo irá executar somente a linha C1, sendo assim:

$$T(1) = C_1$$

Prosseguindo com a relação de recorrência

$$T^{ms}\left(\frac{n}{2}\right) = a + 2T^{ms}\left(\frac{\frac{n}{2}}{2}\right) + T^m\left(\frac{n}{2}\right) =$$

$$a + 2T^{ms}\left(\frac{n}{4}\right) = T^m\left(\frac{n}{2}\right)$$

$$T^{ms}(n) = a + 2 \left[a + 2T^{ms}\left(\frac{n}{4}\right) + T^m\left(\frac{n}{2}\right) \right] + T^m(n) =$$

$$3a + 4T^{ms}\left(\frac{n}{4}\right) + 2T^m\left(\frac{n}{2}\right) + T^m(n)$$

$$T^{ms}\left(\frac{n}{4}\right) = a + 2T^{ms}\left(\frac{\frac{n}{4}}{2}\right) + T^m\left(\frac{n}{4}\right)$$

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

$$7a + 8T^{ms}\left(\frac{n}{8}\right) + 4T^m\left(\frac{n}{4}\right) + 2T^m\left(\frac{n}{2}\right) + T^m(n)$$

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

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

$$n = 2^x$$

$$\log_2 n = x$$

$$\begin{aligned}
T^{ms}(n) &= (2^{\log_2 n} - 1)a + 2^{\log_2 n} T^{ms}\left(\frac{n}{2^{\log_2 n}}\right) + \sum_{i=0}^{\log_2 n - 1} 2^i T^m\left(\frac{n}{2^i}\right) = \\
T^{ms}(n) &= (2^{\log_2 n} - 1)a + n T^{ms}(1) + \sum_{i=0}^{\log_2 n - 1} 2^i T^m\left(\frac{n}{2^i}\right) \\
T^{ms}(n) &= (n - 1)a + nC_1 + \sum_{i=0}^{\log_2 n - 1} 2^i T^m\left(\frac{n}{2^i}\right) \\
T^{ms}(n) &= (n - 1)a + nC_1 + \sum_{i=0}^{\log_2 n - 1} 2^i(b\left(\frac{n}{2^i}\right) + c) = \\
T^{ms}(n) &= (n - 1)a + nC_1 + \sum_{i=0}^{\log_2 n - 1} 2^i b\left(\frac{n}{2^i}\right) + 2^i c = \\
T^{ms}(n) &= (n - 1)a + nC_1 + \sum_{i=0}^{\log_2 n - 1} (bn + 2^i c)
\end{aligned}$$

Figura 4.2.2 Equação do Algoritmo *Merge Sort*

Ainda é possível expandir o somatório, entretanto, neste ponto já é possível observar que a complexidade do algoritmo é $n \log n$.

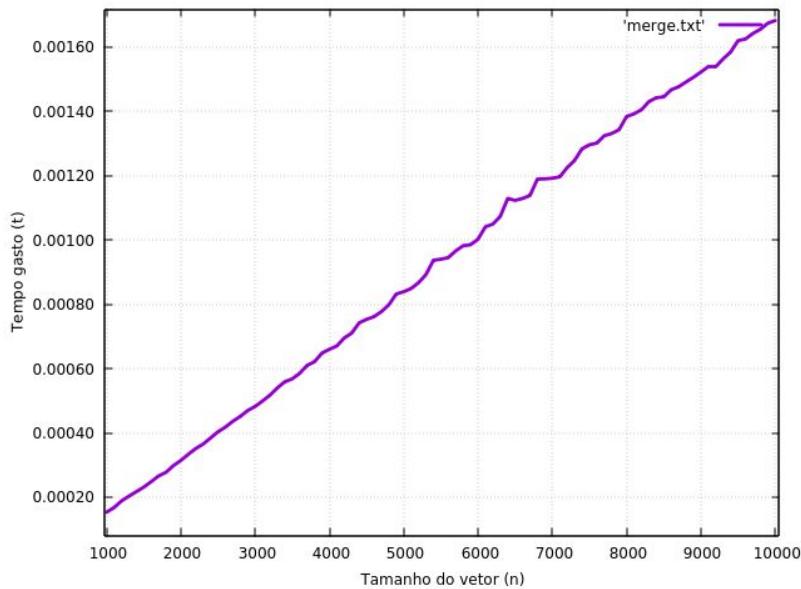


Gráfico 4.2.1 Gráfico do Algoritmo de *Merge Sort*

No **Gráfico 4.2.1** é possível observar o tempo de execução do algoritmo *Merge Sort* em função do número de elementos em um vetor n que varia entre 1000 e 10000. O tempo de execução se assemelha em algo linear, já que é um logaritmo multiplicando n .

4.3 Quick Sort

O *Quick Sort* é considerado algoritmo mais eficiente na ordenação por comparação, nele o vetor original vai sendo dividido em dois por meio de um procedimento recursivo. Essa divisão ocorre até que o vetor que com apenas um elemento, porém, ao contrário do que acontece no *Merge Sort*, os elementos que são comparados já são colocados de forma ordenada, não tendo assim a necessidade de outra função para fazer esse procedimento. Para isso é escolhido um elemento do vetor como sendo o primeiro valor a ser comparado com os demais, geralmente chamado de pivô. Após a partição, todos os elementos a esquerda do pivô são menores e os a direita maiores, fazendo com que o pivô já que na sua posição certa. A partir daí começa as comparações do pivô e os elementos do início e do final do vetor. O *Quick Sort* tem complexidade $\Theta(n^2)$ no pior caso e $\Theta(n \log n)$ em seu melhor e médio caso.

```
01 - void quickSort(int *v, int s, int e){  
02 -     int p;  
03 -     if(s < e){  
04 -         p = partition(v,s,e);  
05 -         quickSort(v,s, p-1);  
06 -         quickSort(v, p+1, e);  
07 -     }  
08 - }
```

Algoritmo 4.3.1 Algoritmo de *Quick Sort* em C

```
01 - int partition(int *v, int s, int e){  
02 -     int l = s, i , aux;  
03 -     for(i=s; i<e; i++){  
04 -         if(v[i]<v[e]){  
05 -             aux= v[i];  
06 -             v[i] = v[l];  
07 -             v[l] = aux;  
08 -             l++;  
09 -         }  
10 -     }  
11 -     aux = v[e];  
12 -     v[e] = v[l];  
13 -     v[l] = aux;  
14 -     return l;  
15 - }
```

Algoritmo 4.3.2 Algoritmo de *Quick Sort* em C

O melhor caso do algoritmo Quick Sort ocorre quando o vetor é preenchido de tal maneira que o pivô selecionado nas partições realizadas seja o elemento médio do vetor auxiliar, fazendo com que o algoritmo seja executado com maior rapidez. A equação que define a complexidade de seu melhor caso pode ser dada por:

$$T_b(n) = C_1 + C_2 + C_3 + C_4 + C_5 + 2T_b\left(\frac{n-1}{2}\right) + T^p(n)$$

Por fins de praticidade, a partir deste ponto considera-se

$$a = C_1 + C_2 + C_3 + C_4 + C_5$$

$$T_b(n) = a + 2T_b\left(\frac{n-1}{2}\right) + T^p(n)$$

É possível observar que é obtida uma relação de recorrência, onde para se chegar no caso base, o vetor deve ser unitário ou nulo, nesse caso somente a linha C1 será executada, então:

$$T_b(0) = C_1$$

$$T_b(1) = C_1$$

Iniciando a próxima equação da recorrência para tentar reconhecer um padrão

$$T_b\left(\frac{n-1}{n}\right) = a + 2T_b\left(\frac{n-3}{4}\right) + T^p\left(\frac{n-1}{2}\right)$$

$$T_b(n) = a + 2 \left[a + 2T_b\left(\frac{n-3}{4}\right) + T^p\left(\frac{n-1}{2}\right) \right] + T^p(n) =$$

$$3a + 4T_b\left(\frac{n-3}{4}\right) + 2T^p\left(\frac{n-2}{2}\right) + T^p(n)$$

Iniciando a próxima equação da recorrência

$$T_b\left(\frac{n-3}{4}\right) = a + 2T_b\left(\frac{\frac{n-3}{n}-1}{2}\right) + T^p\left(\frac{n-3}{4}\right)$$

Substituindo na equação original

$$\begin{aligned} T_b(n) &= 3a + 4 \left[a + 2T_b\left(\frac{n-7}{8}\right) + T^p\left(\frac{n-3}{4}\right) \right] + 2T^p\left(\frac{n-1}{2}\right) + T^p(n) = \\ &= 7a + 8T_b\left(\frac{n-7}{8}\right) + 4T^p\left(\frac{n-3}{4}\right) + 2T^p\left(\frac{n-1}{2}\right) + T^p(n) \end{aligned}$$

Neste ponto já é possível observar o padrão da recorrência, definindo que x substituirá os valores que variam, se tem

$$\begin{aligned} T_b(n) &= (2^x - 1)a + 2^x T_b\left(\frac{n - (2^x - 1)}{2^x}\right) + \sum_{i=0}^{x-1} 2^i T^p\left(\frac{n - (2^i - 1)}{2^i}\right) \\ &= \left(\frac{n+1}{2} - 1\right)a + \frac{n+1}{2} T_b\left(\frac{n - (\frac{n+1}{2} - 1)}{\frac{n+1}{2}}\right) + \sum_{i=0}^{\log_2(n+1)-2} 2^i T^p\left(\frac{n - (2^i - 1)}{2^i}\right) = \end{aligned}$$

$$\left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)T_b(1) + \sum_{i=0}^{\log_2(n+1)-2} 2^i T^p \left(\frac{n - (2^i - 1)}{2^i}\right) =$$

$$\left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)C_1 + \sum_{i=0}^{\log_2(n+1)-2} 2^i T^p \left(\frac{n - (2^i - 1)}{2^i}\right)$$

Agora tratando a outra recorrência, sabe-se que a função T^p é linear, ou seja, tem o formato $ax + b$, então, pode-se considerar

$$2^i T^p \left(\frac{n - (2^i - 1)}{2^i}\right) = 2^i k \left(\frac{n - (2^i - 1)}{2^i}\right) + y$$

$$T_b(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)C_1 + \sum_{i=0}^{\log_2(n+1)-2} 2^i \left(k \left(\frac{n - (2^i - 1)}{2^i}\right) + y\right) =$$

$$T_b(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)C_1 + \sum_{i=0}^{\log_2(n+1)-2} \left(2^i k \left(\frac{n - (2^i - 1)}{2^i}\right) + 2^i y\right) =$$

$$T_b(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)C_1 + \sum_{i=0}^{\log_2(n+1)-2} \left(2^i k \left(\frac{n+1}{2^i} - 1\right) + 2^i y\right) =$$

$$T_b(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)C_1 + \sum_{i=0}^{\log_2(n+1)-2} (k(n+1) - 2^i k) + 2^i y$$

Figura 4.3.1 Equação do Algoritmo *Quick Sort*

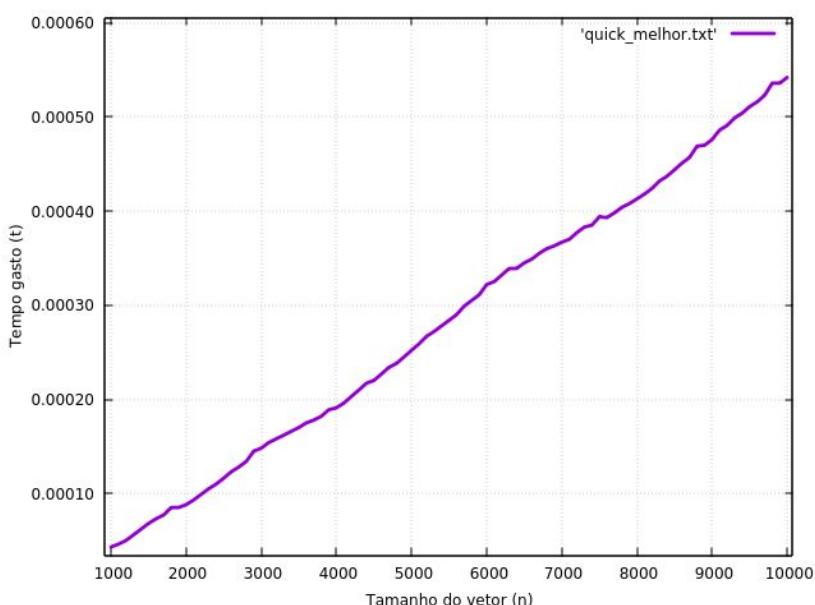


Gráfico 4.3.1 Gráfico do Melhor Caso no Algoritmo de Quick Sort

No **Gráfico 4.3.1** é possível visualizar o tempo de execução do algoritmo *Quick sort* em seu melhor caso em função de um vetor variando entre 1000 e 10000 elementos. O pior caso do algoritmo *Quick Sort* se dá quando o vetor está ordenado de forma crescente, já que o pivô escolhido é o último elemento, irá criar vetores auxiliares de tamanhos $n - 1$ e 0, fazendo com que as comparações sejam feitas para todas as posições do vetor. Se o pivô é muito próximo de um lado da sequência, ou seja, o primeiro, então o tempo de execução é alto, pois a primeira partição exige $n-1$ comparações e apenas rearranja o pivô. A equação que define o pior caso se dá por

$$T_w(n) = 2C_1 + C_2 + C_3 + C_4 + C_5 + T_w(n-1) + T^p(n)$$

Por fins de praticidade, será considerado

$$a = 2C_1 + C_2 + C_3 + C_4$$

É possível observar que é obtida uma relação de recorrência, onde para se chegar no caso base, o vetor deve ser unitário ou nulo, neste caso somente a linha C1 será executada, então

$$T_b(0) = C_1$$

$$T_b(1) = C_1$$

Iniciando a próxima equação da recorrência, se tem

$$T_w(n-1) = a + T_w(n-2) + T^p(n-1)$$

Substituindo na equação original se tem

$$\begin{aligned} T_w(n) &= a + [a + T_w(n-2) + T^p(n-1)] T^p(n) = \\ &= 2a + T_w(n-2) + T^p(n) + T^p(n-1) \end{aligned}$$

Iniciando a próxima equação da recorrência, se tem

$$T_w(n-2) = a + T_w(n-3) + T^p(n-2)$$

Substituindo na equação original se tem

$$\begin{aligned} 2a + [a + T_w(n-3) + T^p(n-2)] + T^p(n) + T^p(n-1) &= \\ 3a + T_w(n-3) + \sum_{i=0}^{n-2} T^p(n-i) &= \\ (n-1)a + T_w(1) + \sum_{i=0}^{n-2} z(n-i) + y &= \\ n - a - a + C_1 + (n-1)y + z \sum_{i=2}^n i &= \end{aligned}$$

$$\begin{aligned}
n(a+y) - y - a + C_1 + z \left(\frac{n}{2} (n+1) - 1 \right) &= \\
n(a+y) - y + a + C_1 - z + z \frac{n}{2} (n+1) &= \\
n \left(a + y + \frac{3}{z} \right) - y - a + C_1 - z + \frac{zn^2}{2} &= \\
\frac{zn^2}{2} + \left(a + y + \frac{z}{2} \right) n + C_1 - y - a - z
\end{aligned}$$

Figura 4.3.1 Equação do Algoritmo *Quick Sort*

Neste ponto já é possível observar que o pior caso do algoritmo *Quick Sort* conta com termos quadráticos, ou seja, sua complexidade é $\Theta(n^2)$.

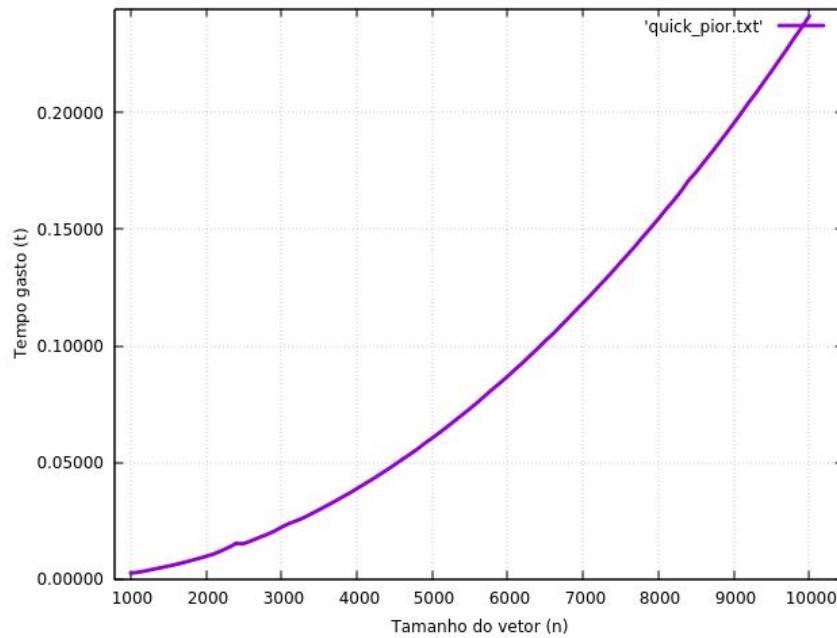


Gráfico 4.3.2 Gráfico do Pior Caso no Algoritmo de *Quick Sort*

No **Gráfico 4.3.2** temos a parábola formada pelo tempo de execução do *Quick Sort* em função de um vetor que varia entre 1000 e 10000 posições. Para analisar seu caso médio esperado foi preenchido o vetor de forma aleatória, mantendo o pivô como o último elemento, sua complexidade assim como o melhor caso se dá por $\Theta(n \log n)$.

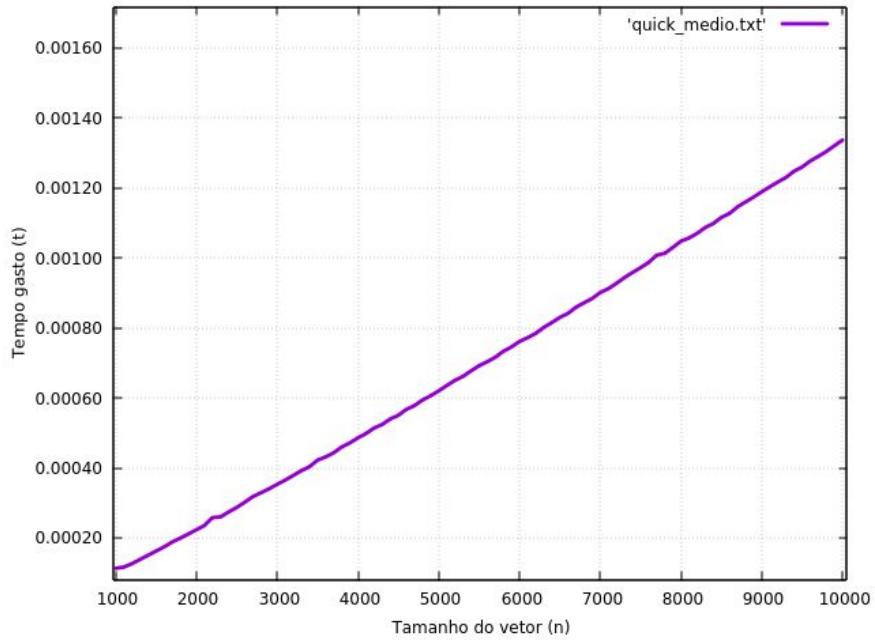


Gráfico 4.3.3 Gráfico do Caso Médio no Algoritmo de *Quick Sort*

No **Gráfico 4.3.4** é possível analisar todos os casos do *Quick Sort* graficamente, entretanto, em comparação ao melhor e médio caso o pior caso tem um tempo de execução muito maior, por se tratar de um algoritmo quadrático.

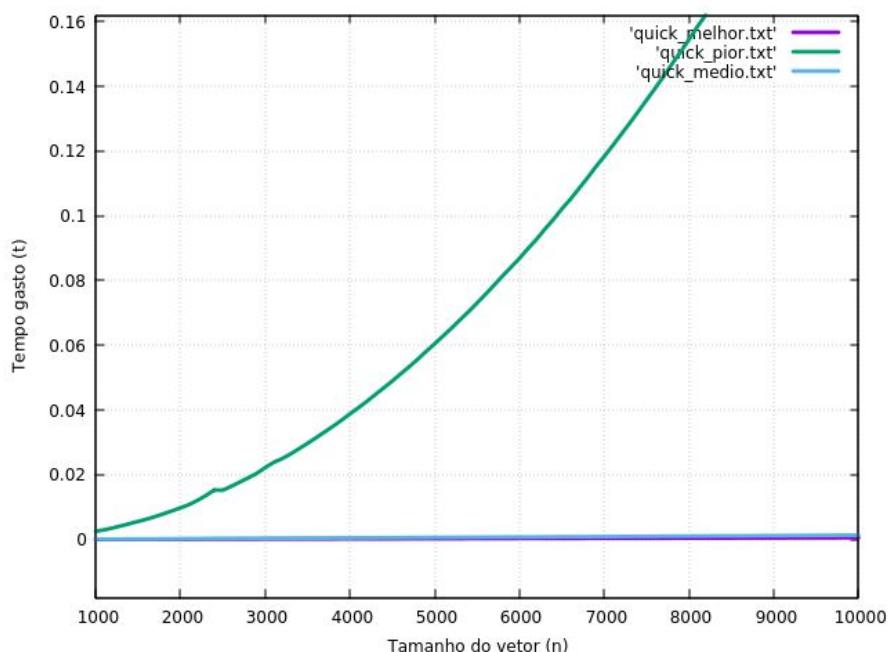


Gráfico 4.3.4 Gráfico de comparação do Algoritmo de *Quick Sort*

No **Gráfico 4.3.5** é possível visualizar de forma mais clara a comparação entre os casos médios e melhor caso do *Quick Sort*.

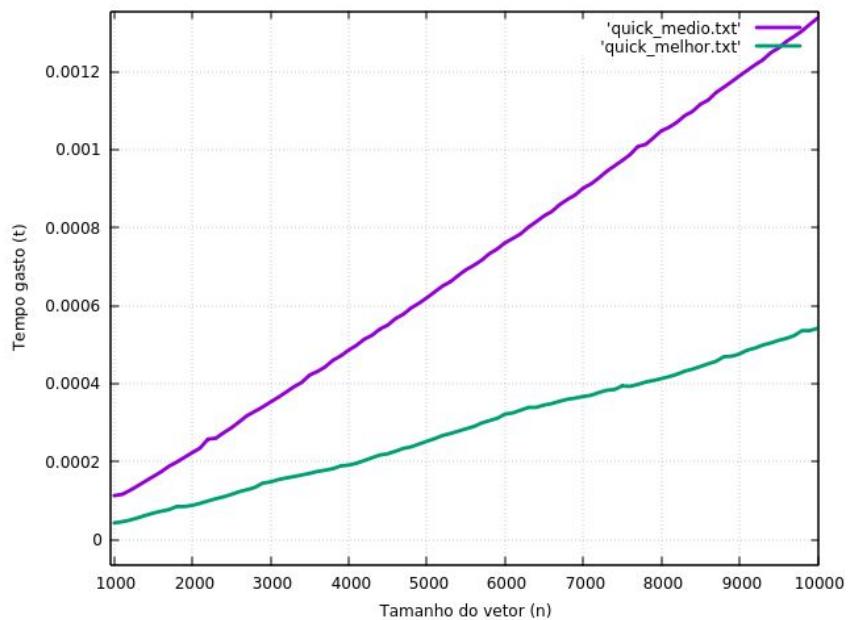


Gráfico 4.3.5 Gráfico de comparação entre o Melhor e os Casos Médios do Algoritmo de *Quick Sort*

4.4 Distribution Sort

O Distribution Sort é um dos, se não o, algoritmo de ordenação mais rápido, sendo de ordem estável $\Theta(n + k)$, entretanto, sua execução requer grande uso de memória em determinados casos por criar vetores auxiliares com grande tamanho. Utiliza uma técnica de encontrar o maior e o menor elemento presente no vetor, e então encontrar sua real posição através da criação e manipulação de vetores auxiliares (por isso o grande gasto de memória). Seu tempo de execução depende tanto do tamanho do vetor, quanto do vetor auxiliar, que vai ser de tamanho do maior elemento menos o tamanho do menor elemento.

```

01 - int *distribution (int *v, int n){
02 -     int l = min(v,n);
03 -     int b = max(v,n);
04 -     int k = b-l+1;
05 -     int i, j;
06 -     int *w = new_0(k);
07 -     int *y = new_0(n);
08 -     for (i = 0; i < n; i++){
09 -         w[v[i]-l]++;
10 -     }
11 -     for (j = 1; j <= b-l; j++){
12 -         w[j] = w[j] + w[j-1];
13 -     }
14 -     for (i = 0; i <= n-1; i++){
15 -         j = w[v[i]-l];
16 -         y[j-1] = v[i];
17 -         w[v[i]-l]--;

```

```

18 - }
19 - return y;
20 - free(y);
21 - free(w);
22 -}

```

Algoritmo 4.4.1 Algoritmo de *Distribution Sort* em C

Para entender melhor a equação da complexidade do *Distribution Sort*, consideremos

$$a = C_1 + C_2 + C_3 + C_4 + C_5 + C_6$$

e

$$k = x - y$$

Sendo x o maior elemento presente no vetor e y o menor elemento. Então temos

$$\begin{aligned} T(n) &= a + (n+1)C_7 + nC_8 + (k+1)C_9 + kC_{10} + (n+1)C_{11} + n(C_{12} + C_{13} + C_{14}) = \\ &a + nC_7 + C_7 + nC_8 + kC_9 + C_9 + kC_{10} + nC_{11} + C_{11} + n(C_{12} + C_{13} + C_{14}) = \\ &k(C_9 + C_{10}) + n(C_7 + C_8 + C_{11} + C_{12} + C_{13} + C_{14}) + C_7 + C_9 + C_{11} + a \end{aligned}$$

Figura 4.4.1 Equação do Algoritmo *Distribution Sort*

Observando a **Figura 4.4.1** é possível provar a ordem, que embora seja linear, depende não somente de n . A grande desvantagem do *Distribution Sort* é que o uso de memória gasto cresce em função de k . A Figura 19 mostra seu tempo de execução em função de um vetor que varia entre 1000 e 10000 posições.

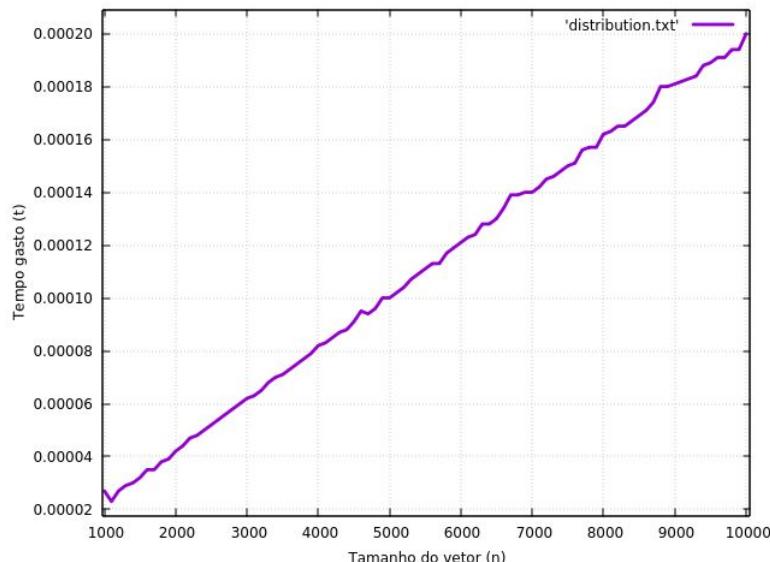


Gráfico 4.4.1 Gráfico de tempo de execução do Distribution Sort

4.5 Comparação

Vemos que, dentre todos os algoritmos quadráticos mostrados neste trabalho, o Insertion em seu caso médio é o que possui menor tempo de execução, e até mesmo em seu pior caso tem tempo de execução inferior aos outros algoritmos mostrados. Como mostrado no **Quadro 4.5.1** o Insertion é um algoritmo $\Theta(n)$ no seu melhor caso e $\Theta(n^2)$ no médio e no pior.

Embora seja o melhor dos algoritmos de ordem quadrática, o *Insertion Sort* tem tempo de execução muito superior comparado a outros algoritmos de ordem n e $n \log n$, fazendo assim, com que aconteça uma grande diferença entre ele e os outros algoritmos.

O Quick Sort em seu melhor caso, foi o algoritmo mais rápido, mas é necessário realizar uma ressalva pois seu grande uso de memória pode torná-lo perigoso de se usar. O *Merge Sort* tem o posto de pior algoritmo entre os com ordem linear.

Algoritmo	BEST	AVERAGE	WORST
Insertion Sort	n	n^2	n^2
Merge Sort	$n \log n$	$n \log n$	$n \log n$
Quick Sort	$n \log n$	$n \log n$	n^2

Quadro 4.5.1 Quadro de comparação entre os tempos de execução dos algoritmos

5. Conclusão

Com as informações coletadas durante o desenvolvimento deste trabalho é possível determinar qual o melhor algoritmo dada determinada situação.

O *Insertion Sort* é o melhor dos algoritmos de ordem quadrática dentre os apresentados neste trabalho, em comparação com outros algoritmos de ordem linear tem tempo de execução muito alto. O merge sort, de ordem $n \log n$ tem tempo de execução inferior a todos os algoritmos quadráticos mostrados neste trabalho, entretanto, utiliza memória auxiliar por conta de sua pilha de execução e criação de vetores auxiliares.

O quick sort, também de complexidade $n \log n$, tem em seu caso médio tempo de execução menor que o merge sort e todos os algoritmos quadráticos, não utiliza vetores auxiliares e seu gasto de memória é em função à sua pilha de execução, é em quase todos os casos o melhor algoritmo a se utilizar, tornando-o assim, o melhor algoritmo entre os outros que foram demonstrados.

Em relação aos algoritmos de busca apresentados, exceto no melhor caso de ambos, a busca binária leva vantagem sobre a busca linear em relação ao tempo de execução, em contrapartida tem um custo de memória maior pois é em função do tamanho da pilha de execução gerada. Então, se memória não for um problema e o elemento buscado não estiver na primeira posição do vetor (melhor caso de ambos) a melhor opção para se utilizar é a busca binária; caso o elemento esteja na primeira posição do vetor e/ou o consumo de memória por um problema, a busca linear é mais recomendada.

6. Referências

FEOFILOFF, Paulo. Projeto de Algoritmos: Busca binária (versão simplificada). Disponível em: <<http://www.ime.usp.br/~pf/algoritmos/aulas/bubi2.html>>. Acesso em:

FEOFILOFF, Paulo - Merge Sort: <http://www.ime.usp.br/~pf/mac0122-2002/aulas/mergesort.html>

RICARTE, Ivan L. M. Busca binária. Disponível em:

<<http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node20.html>>. Acesso em:

RICARTE, Ivan L. M. Algoritmos de busca. Disponível em:

<<http://www.icmc.usp.br/~sce182/lestbus.html>>. Acesso em:

Wikipédia, a enciclopédia livre. Pesquisa Binária. Disponível em:

<http://pt.wikipedia.org/wiki/Busca_Bin%C3%A1ria>. Acesso em:

PIMENTEL,Graça ; MIGHIM, Rosane. Algoritmos de Busca em Lista Estática Sequencial.

Disponível em:<<http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node19.html>>.

M. BEDER, Delano - *Algoritmos de Ordenação: Merge Sort:*

<http://www.ic.unicamp.br/~luciano/ACH2002/notasdeaula/11-mergeSort.pdf>

<https://www.profissionaisti.com.br/2016/10/analise-de-algoritmos-analise-assintotica/>

<http://www.dcc.fc.up.pt/~pbv/aulas/progimp/teoricas/teorica16.html>

https://pt.wikiversity.org/wiki/Introdu%C3%A7%C3%A3o_%C3%A0s_Estruturas_de_Dados/

https://pt.wikipedia.org/wiki/Algoritmos_de_Ordena%C3%A7%C3%A3o

https://pt.wikipedia.org/wiki/Ordem_quadr%C3%A1tica

https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_linear

https://pt.wikipedia.org/wiki/Algoritmo_de_busca