1 Introdução aos algoritmos de ordenação

Em computação, um algoritmo de ordenação colocar 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. Dado o algoritmo, é possível calcular sua complexidade, assim como seu tempo de execução, e, partir de uma análise criteriosa decidir se o mesmo é o melhor a ser usado em dada circunstância. Geralmente, a complexidade dos algoritmos de ordenação são quadráticos ou lineares.

1.1 Algoritmos Quadráticos

Algoritmos de ordenação com complexidade quadrática são representados por Θ 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

Algoritmos de ordenação com complexidade linear são representados por Θ(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

Algoritmos de ordenação com complexidade logarítmica são representados por (logn), ou seja, é uma 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 igual a um milhão, log2n é aproximadamente 20.

2 Análise dos algoritmos de ordenação e busca

2.1 Elementos da análise assintótica

2.1.1. Notação O

Existem várias formas de análise assintótica, e uma das mais conhecidas é a **O** ou **BIG-O**. Ela representa o custo (seja de tempo ou espaço) do algoritmo no pior caso possível de todas as

entradas de tamanho **n**, ou seja, ele analisa o limite superior de entrada, desse modo, podemos dizer que o comportamento do nosso algoritmo não pode nunca ultrapassar certo limite.

2.1.2. Notação Ω

2.1.3 Notação Θ

2.1.4. Notação ω

2.1.5. Notação o

Neste relatório, serão apresentados os algoritmos de ordenação **insertion sort, merge sort, counting sort, quick sort, distribution sort e bubble sort**, assim como uma comparação de desempenho entre eles, utilizando a ferramenta GNUPLOT para criar os gráficos.

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

3 Algoritmos de Busca

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.

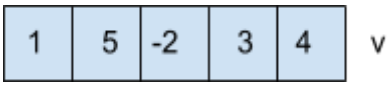


Figura 3.1.1 Vetor v

```
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; //elemento encontrado
6 -       }
7 -   }
8 -   return -1; //elemento não encontrado
9 - }
```

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.**

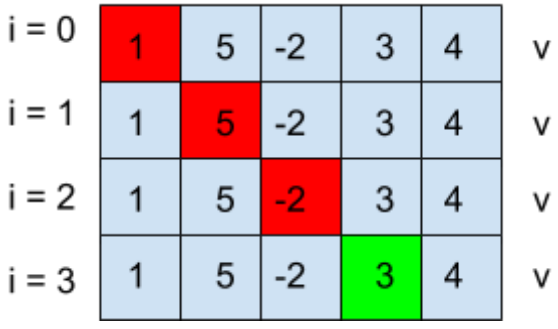


Figura 3.1.2 Vetor v sendo utilizado como exemplo de busca pelo valor 3

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.

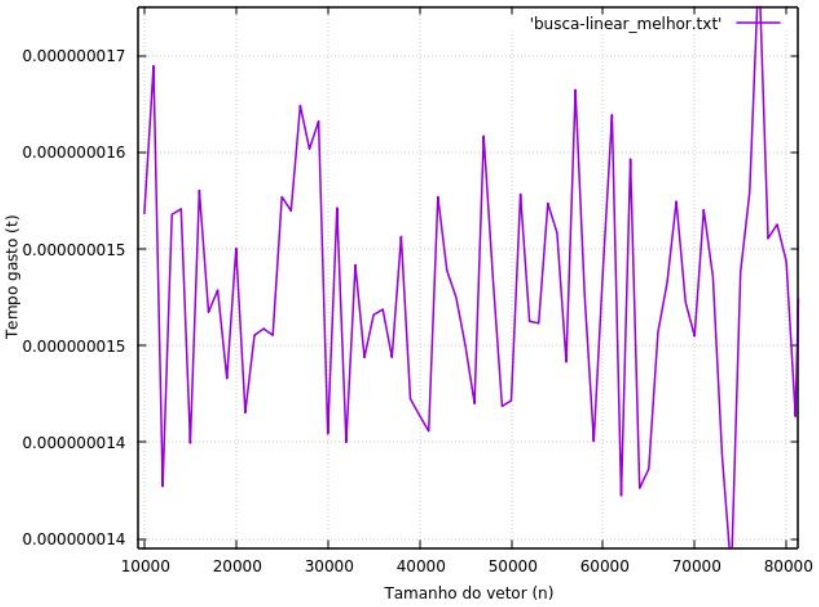


Gráfico 3.1.1 Gráfico do Melhor Caso no Algoritmo de Busca Linear

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.

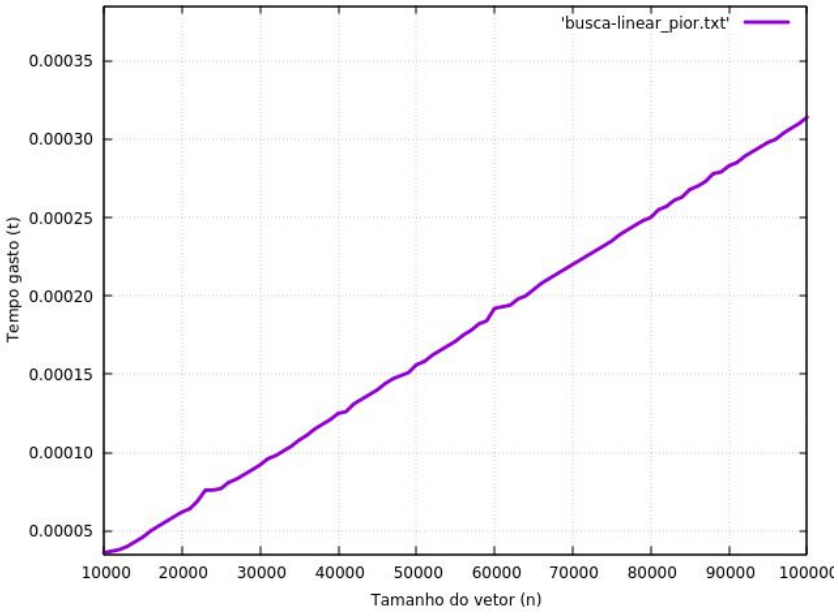


Gráfico 3.1.2 Gráfico do Pior Caso no Algoritmo de Busca Linear

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).

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.

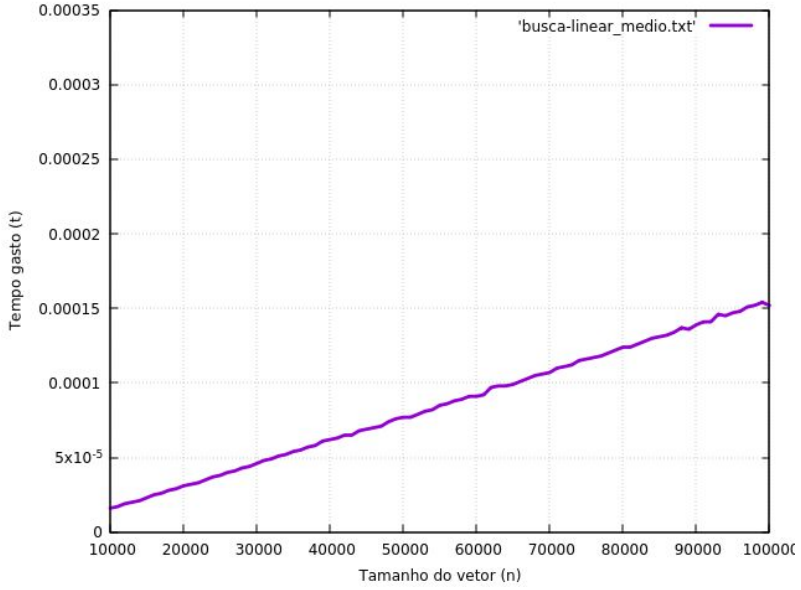


Gráfico 3.1.3 Gráfico do Caso Médio no Algoritmo de Busca Linear

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

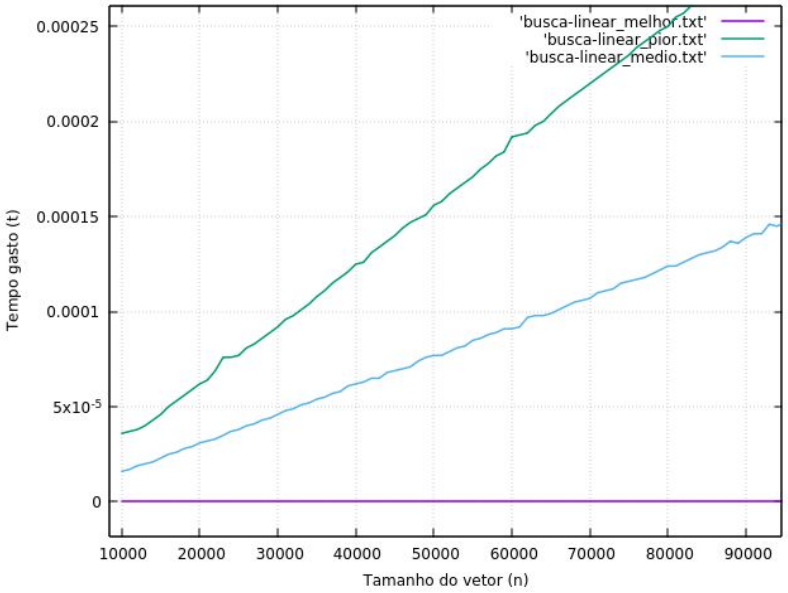


Gráfico 3.1.4 Gráfico de Comparação do Algoritmo de Busca Linear

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.**

Figura 3.2.1 Equação do Melhor Caso no Algoritmo de Busca Linear

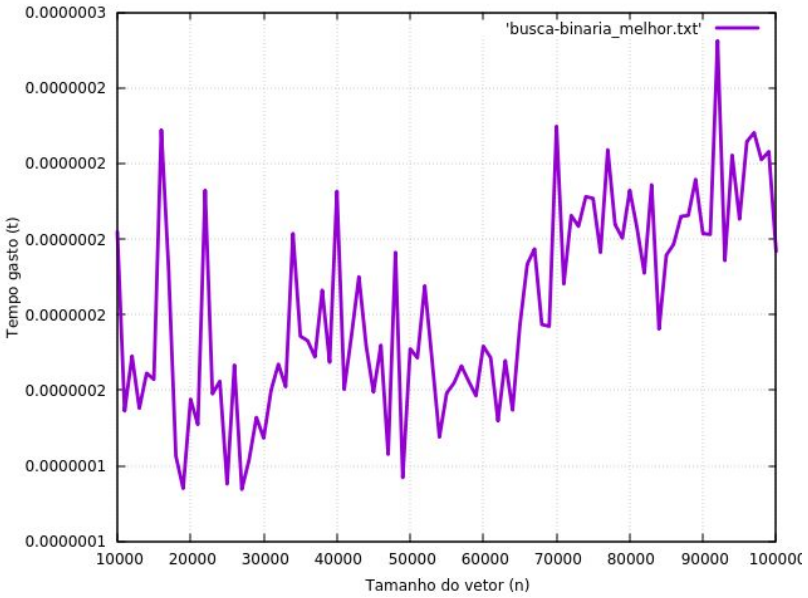


Gráfico 3.2.1 Gráfico de Melhor Caso do Algoritmo de Busca Linear

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.

Figura 3.2.2 Equação do Pior Caso no Algoritmo de Busca Linear

Neste ponto já é possível encontrar uma padrão nas trocas, então será utilizado a variável *x* para representá-las, e depois de realizar tal ação é é necessário encontrar o valor de x que zere a equação para dar fim a recorrência, como é mostrado na **Figura 3.2.3.**

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.**

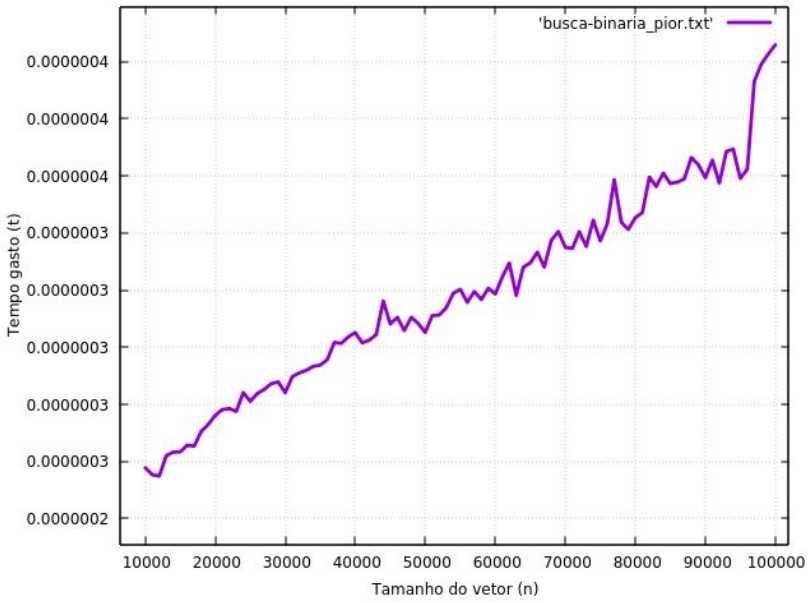


Gráfico 3.2.2 Gráfico de Pior Caso do Algoritmo de Busca Binária

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.**

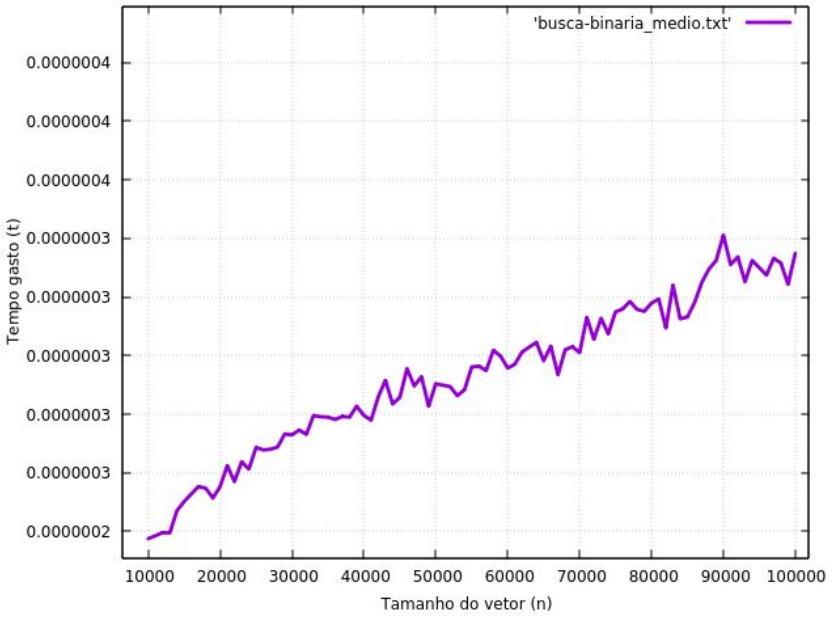


Gráfico 3.2.3 Gráfico de Médio Caso do Algoritmo de Busca Binária

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

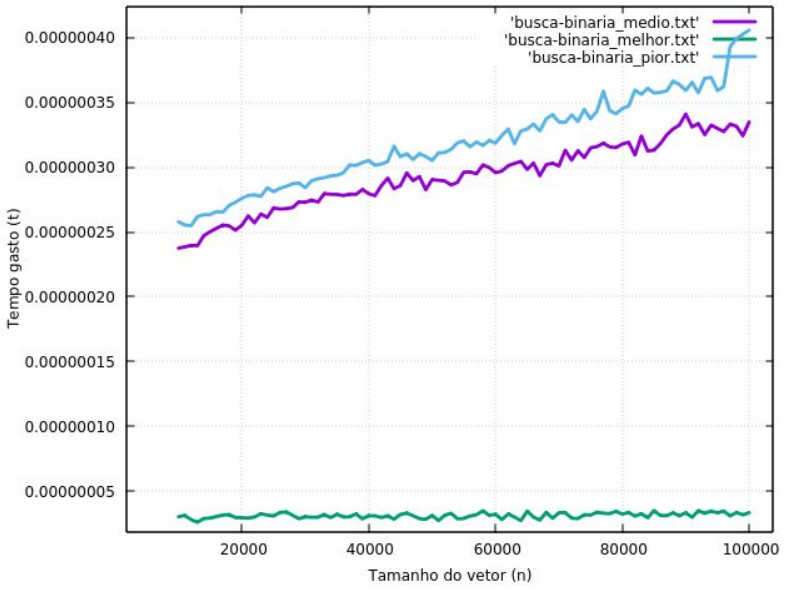
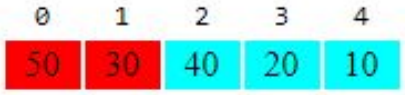


Gráfico 3.2.4 Gráfico da comparação dos casos no Algoritmo de Busca Binária

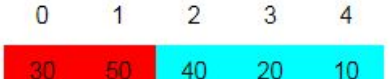
4 Algoritmos de Ordenação

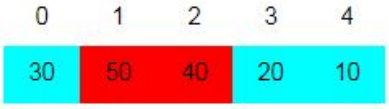
4.1 Insertion Sort

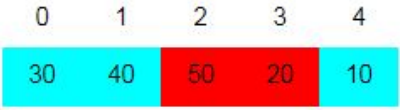
Considera que o primeiro elemento está ordenado (ou seja, na posição correta). A partir do segundo elemento, insere os demais elementos na posição apropriada entre aqueles já ordenados. O elemento é inserido na posição adequada movendo-se todos os elementos maiores para posição seguinte do vetor. Mais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



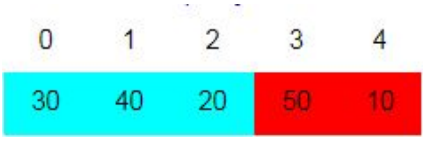
O elemento da posição 0 (valor 50) é comparado com o elemento da posição 1 (valor 30). Como o objetivo é ordenar crescentemente, os conteúdos dos elementos da posições 0 e 1 devem ser trocados entre si.



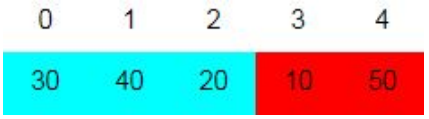


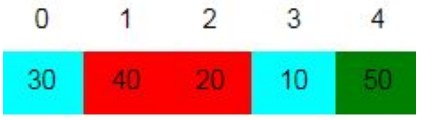


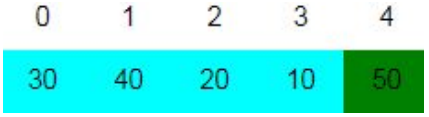
Em seguida serão comparados os conteúdos dos elementos das posições 1 e 2 troca elementos das posições 2 e 3.

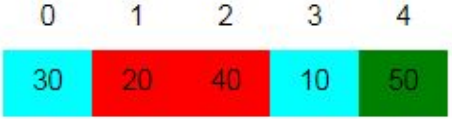


Elementos das posições 3 e 4 apesar do vetor não estar ordenado ainda, observe que o maior elemento ficou na última posição:

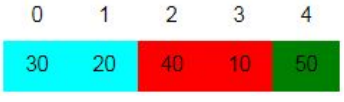


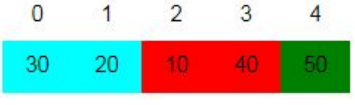


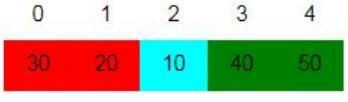




O processo recomeça, porém ocorrerá entre as posições 0 e 3 (o elemento da posição 4 já está ordenado). Não acontece a troca entre os elementos das posições 1 e 2:







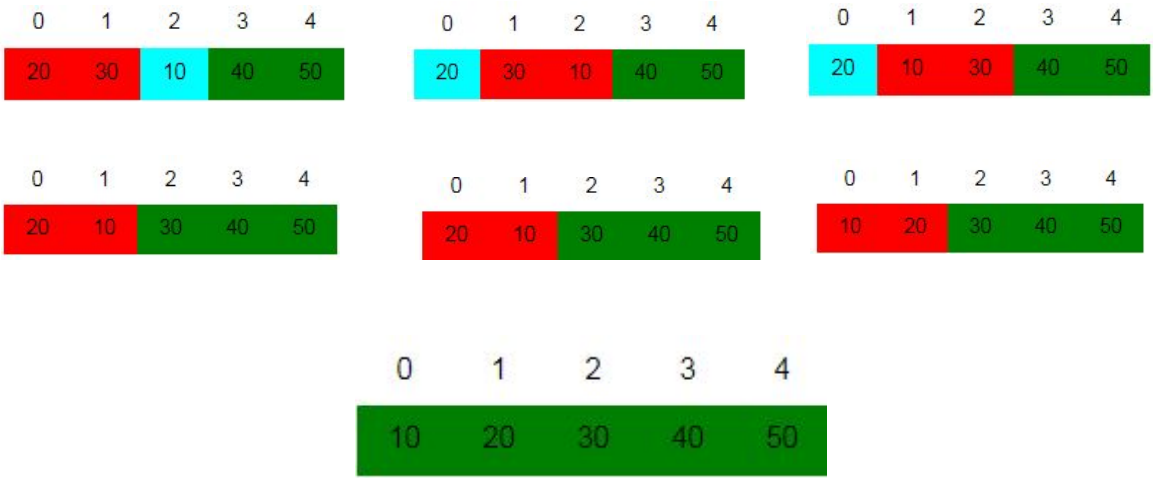


Figura 4.1.1 Procedimento de ordenação no Insertion Sort

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:

$$T_b(n) = C_1 + nC_2 + (n-1)C_3 + (n-1)C_4 + (n-1)C_6$$
$$= C_1 + nC_2 + nC_3 - C_3 + nC_4 - C_4 + nC_6 - C_6$$
$$= n(C_2 + C_3 + C_4 + C_6) - C_3 - C_4 - C_6 + C_1$$

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 Θ (n), ou seja, linear.

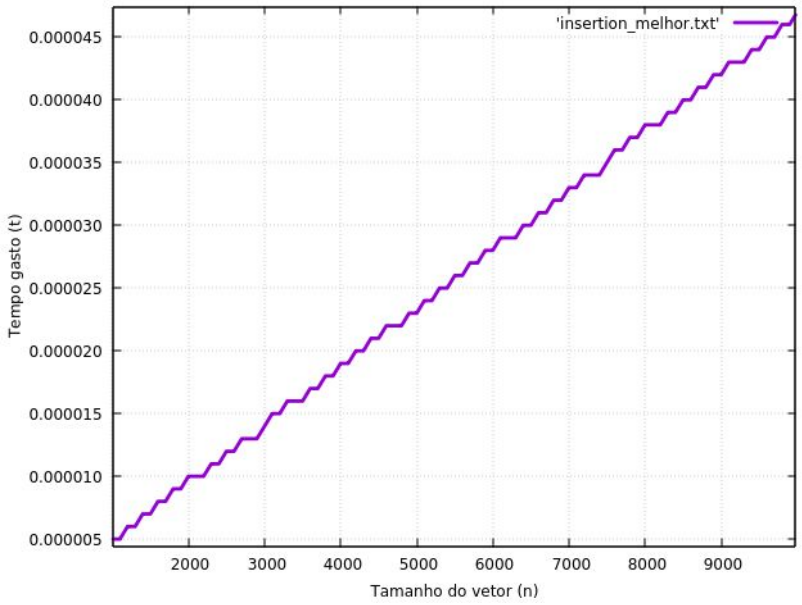


Gráfico 4.1.1 Gráfico de Melhor Caso do Algoritmo de Busca Binária

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.

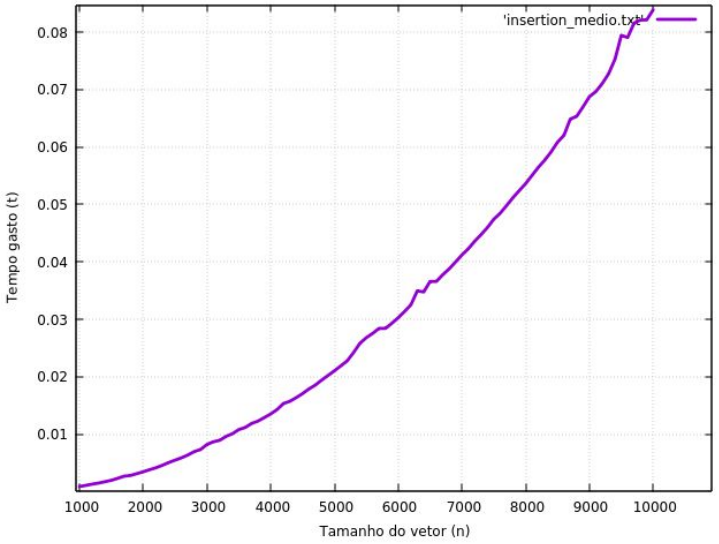


Gráfico 4.1.2 Gráfico do Caso Médio do Algoritmo de Busca Binária

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:

$$t_w(n) = C_1 + nC_2 + (n-1)C_3 + C_4 \sum_{i=1}^{n-1} i + C_5 \sum_{i=1}^{n-1} i + (n-1)C_6 + (n-1)C_4$$

$$= C_1 + nC_2 + nC_3 - C_3 + C_4\left(\frac{n}{2}(n+1) - n\right) + C_5\left(\frac{n}{2}(n+1) - n\right) + C_6 n - C_6 + C_4(n-1)$$

$$= C_1 + nC_2 + nC_3 - C_3 + C_4\left(\frac{n^2+n}{2} - n\right) + C_5\left(\frac{n^2+n}{2} - n\right) + C_6 n - C_6 + C_4(n-1)$$

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

Figura 4.1.3 Equação do Pior Caso no Algoritmo Insertion Sort

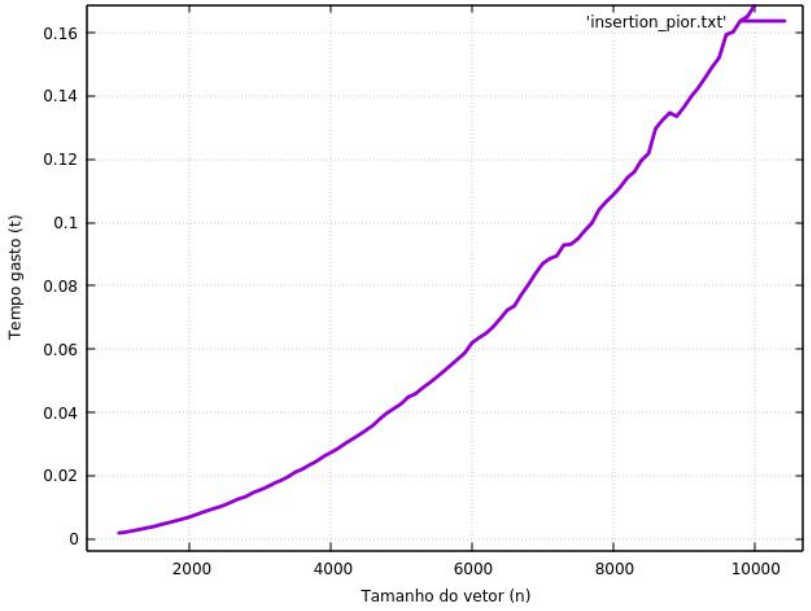


Gráfico 4.1.3 Gráfico do Pior Caso no Algoritmo de Busca Binária

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.

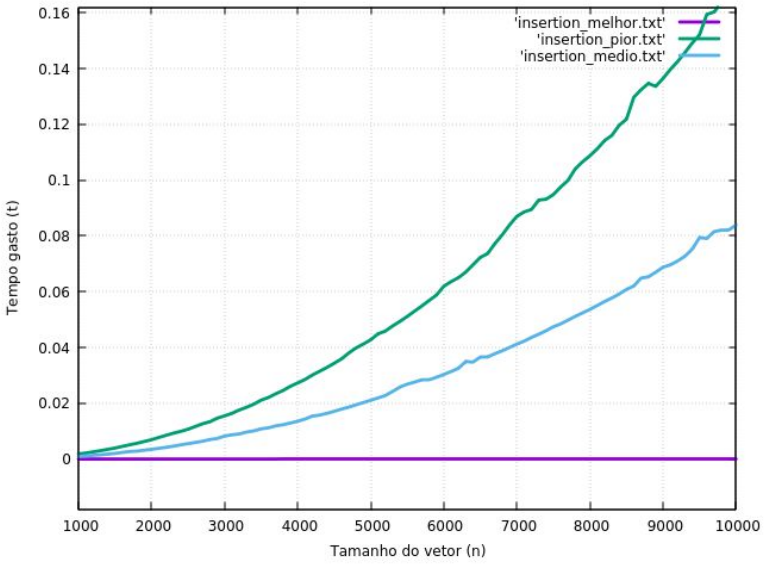


Gráfico 4.1.4 Gráfico de Comparação entre os Algoritmos de Busca Binária

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.

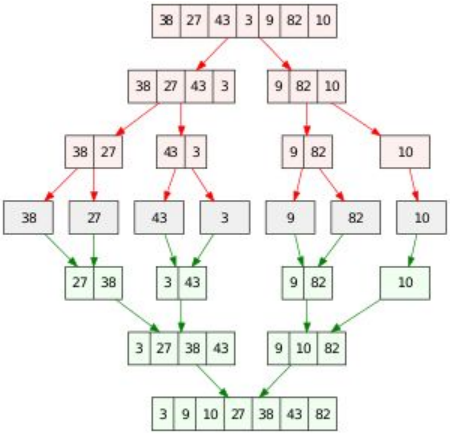


Figura 4.2.1 Procedimento de ordenação do Algoritmo Merge Sort

```
1 - void *merge(int *vetor, int comeco, int meio, int fim) {
2 -   int i = comeco;
3 -   int j = meio+1, k=0;
4 -   int *aux = (int *) malloc((fim-comeco+1) * sizeof(int));
5 -   while (k <= fim-comeco){
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-comeco){
17-         vetor[comeco+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 comeco, int fim){
2 -  if (comeco < fim) {
3 -      int meio = (fim+comeco)/2;
4 -      mergeSort(vetor, comeco, meio);
5 -      mergeSort(vetor, meio+1, fim);
6 -      merge(vetor, comeco, 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 $\Theta(logn)$, 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 $\Theta(nlogn)$. 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.**

Figura 4.2.2 Equação do Algoritmo Merge Sort

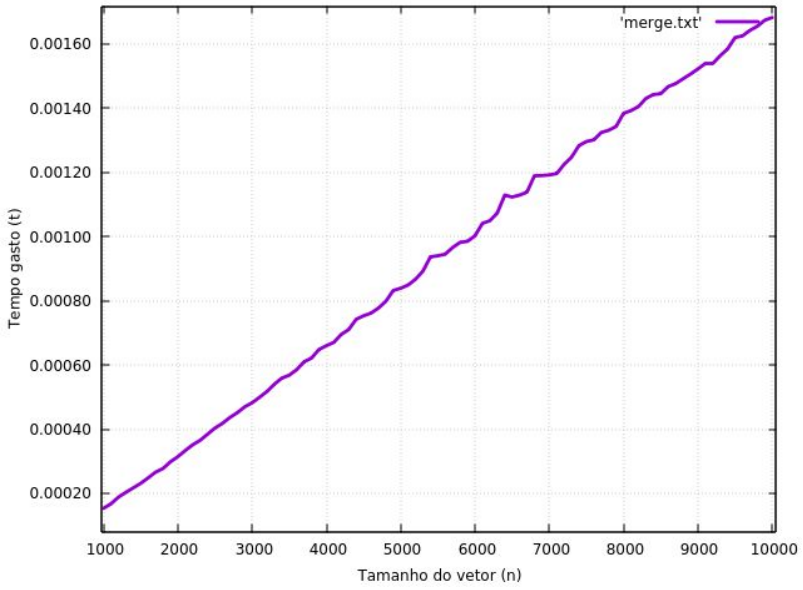


Gráfico 4.2.1 Gráfico do Algoritmo de Merge Sort

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 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 m do vetor. O Quick Sort tem complexidade $\Theta(n^2)$ no pior caso e $\Theta(nlogn)$ 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

4.4 Distribution Sort

Algoritmo	Tempo de execução no pior caso	Tempo de execução no melhor caso	Tempo de execução médio
ordenação por seleção (selection sort)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Ordenação por combinação (merge sort)	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

5. Referências

FEOFILOFF, Paulo. Projeto de Algoritmos: Busca binaria (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. Algoritimos de busca.Disponível em: <http://www.icmc.usp.br/~sce182/lestbus.html>. Acesso em:

Wikipedia, 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