UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – UFRN

CENTRO DE ENSINO SUPERIOR DO SERIDO – CERES

DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA – DCT

BACHARELADO EM SISTEMAS DE INFORMACAO

Prof. DSc. JOAO PAULO DE SOUZA MEDEIROS

MATEUS MEDEIROS DE ARAÚJO

TRABALHO DE IMPLEMENTAÇÃO UNIDADE I

Caicó – RN

Abril - 2018

Resumo

O presente relatório tem como objetivo apresentar estudos coletados durante a primeira unidade da disciplina de estrutura de dados ministrada pelo professor DSc. João Paulo de Souza Medeiros na Universidade Federal do Rio Grande no Norte. Serão apresentados gráficos e informações acerca de alguns algoritmos de ordenação, sendo eles couting sort, insertion sort, merge sort, quick sort, distribution sort e bubble sort. Além de outros algoritmos de busca. Para implementação dos códigos foi utilizado o sistema operacional Linux Mint, a linguagem de programação C, o compilador GCC. A ferramenta GNUPLOT foi utilizada para criação dos gráficos.

Abstract

This report aims to present studies collected during the first unit of the discipline of data structure taught by the teacher DSc. João Paulo de Souza Medeiros at the Federal University of Rio Grande do Norte. Graphs and information about some sort algorithms will be presented, being couting sort, insertion sort, merge sort, quick sort, distribution sort and bubble sort. In addition to other search algorithms. For the implementation of the codes the Linux Mint operating system, the C programming language, the GCC compiler was used. The GNUPLOT tool was used to create the graphics.

SUMÁRIO

1 Introdução aos algoritmos de ordenação

Em computação, um algoritmo de ordenação coloca 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 $\Theta\left(n^2\right)$, 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 $\Theta(n)$, 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

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, $log_2 n$ é aproximadamente 20.

2 Análise dos algoritmos de ordenação

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

2.1 Insertion sort

O insertion sort é um algoritmo de ordenação por inserção que percorre o vetor a partir da comparação do seu segundo elemento com o seu anterior e conforme

avança os elementos mais à esquerda vão ficando ordenados, pois nas comparações, busca-se o elemento menor que o de partida, invertendo suas posições. É um algoritmo de ordem quadrática, com excessão ao seu melhor caso, e, por muitos é considerado o melhor algoritmo dessa ordem de classificação.

```c
void ordena (int *vetor, unsigned int n){
    int i, j, atual;
    for (i = 1; i < n; i++) {
        atual = vetor[i];

        for (j = i - 1; (j >= 0) && (atual < vetor[j]); j--) {
            vetor[j+1] = vetor[j];
        }
        vetor[j+1] = atual;
    }
}
```

Figure 1: Código em C do algoritmo insertion sort

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

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

lembrando que C representa o custo de tempo referente à linha, como visto na figura 1.

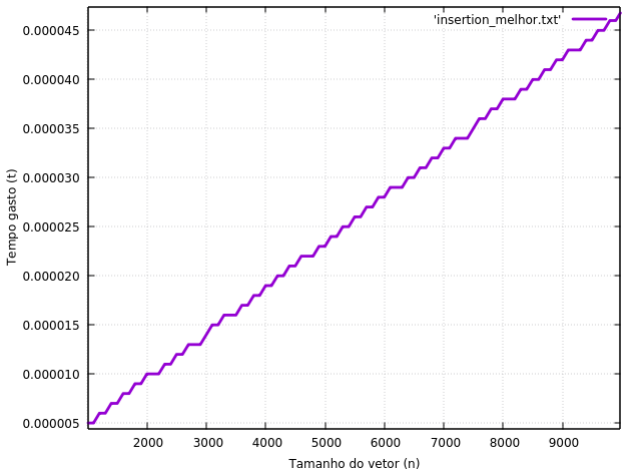


Figure 2: Gráfico de tempo de execução do insertion sort em seu melhor caso

Na figura 2 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.

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 esta ordenado, variando assim o tempo de execução.

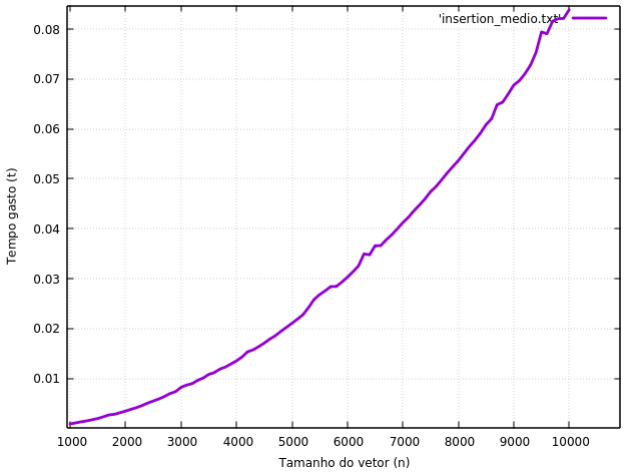


Figure 3: Gráfico de tempo de execução do insertion sort em seu caso médio

Na figura 3 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 $\Theta\left(n^2\right)$, a equação que define sua ordem é dada por

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

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

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

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

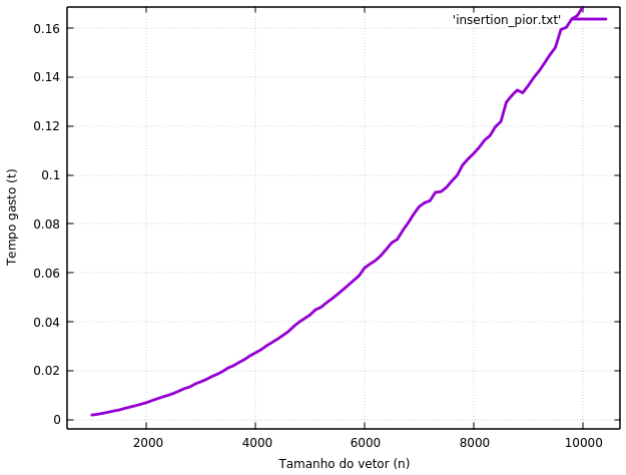


Figure 4: Gráfico de tempo de execução do insertion sort em seu pior caso

Como visto na equação apresentada graficamente na figura 4, assim como o caso médio, o pior caso do insertion sort tem complexidade $\Theta\left(n^2\right)$, embora seu tempo de execução seja superior.

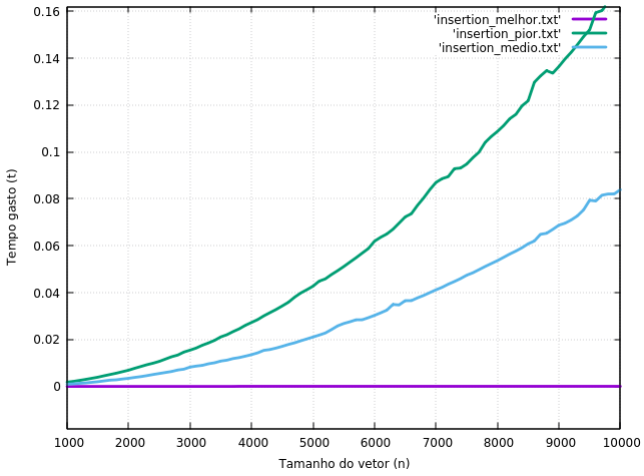


Figure 5: Gráfico comparativo entre os casos do insertion sort

No gráfico da figura 5 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.

2.2 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 algorítmo, é necessário ter conhecimento do tamanho do vetor, bem como seu valor de inicio e fim, para que seja possível dividir o vetor em subvetores.

```
void *merge(int *vetor, int comeco, int meio, int fim) {
    int i = comeco;
    int j = meio+1, k=0;;
    int *aux = (int *) malloc((fim-comeco+1) * sizeof(int));

    while (k <= fim-comeco){
        if ((vetor[i] < vetor[j] && i <= meio) || (j > fim)){
            aux[k] = vetor[i];
            i++;
        }else{
            aux[k] = vetor[j];
            j++;
        }
        k++;
    }
    k = 0;

    while (k <= fim-comeco){
        vetor[comeco+k] = aux[k];
        k++;
    }
    free(aux);
}
```

Figure 6: Código em C do algoritmo merge sort

Esse processamento é possível devido o método de recursão onde o problema inicial é divido em sub-problemas, transformando-se em problemas simples de serem resolvidos. O algoritmo é composto por duas funções, uma para ordenação e outra responsável por dividir o vetor em sub-vetores. 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 que a complexidade do algoritmo é $\Theta(nlogn)$. É possível também chegar a essa conclusão de forma mais precisa equacionando o problema:

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

por fins de simplicidadade, 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 C_1, sendo assim

$$T(1) = C_1$$

prosseguindo com a relação de recorrência

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

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

então temos que

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

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

calculando mais uma relação de recorrência, temos

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

substituindo os termos, ficamos com

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

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

a partir deste ponto, já é possível perceber um padrão para a criação da fórmula fechada, sendo x a variável que muda em cada relação, então

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

para encerrar a relação de recorrência é necessário que o vetor tenha tamanho unitário, então

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

$$n = 2^x$$

$$log_2 n = x$$

substituindo na equação se obtém

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

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

então

$$T^{ms}(n) = (n-1)\,a + nC_1 + \sum_{i=0}^{log_2 n - 1} 2^i T^m\left(\frac{n}{2^i}\right)$$

a função T^m tem complexidade $\Theta(n)$, então por simplificação, consideramos no formato $bx + c$, então

$$T^{ms}(n) = (n-1)\,a + nC_1 + \sum_{i=0}^{log_2 n - 1} 2^i (b\left(\frac{n}{2^i}\right) + c) =$$

expandindo se obtém

$$T^{ms}(n) = (n-1)\,a + nC_1 + \sum_{i=0}^{log_2 n - 1} 2^i b\left(\frac{n}{2^i}\right) + 2^i c =$$

$$T^{ms}(n) = (n-1)\,a + nC_1 + \sum_{i=0}^{log_2 n - 1} (bn + 2^i c)$$

ainda é possível expandir o somatório, entretando, neste ponto já é possível observar que a complexidade do algoritmo é $nlogn$.

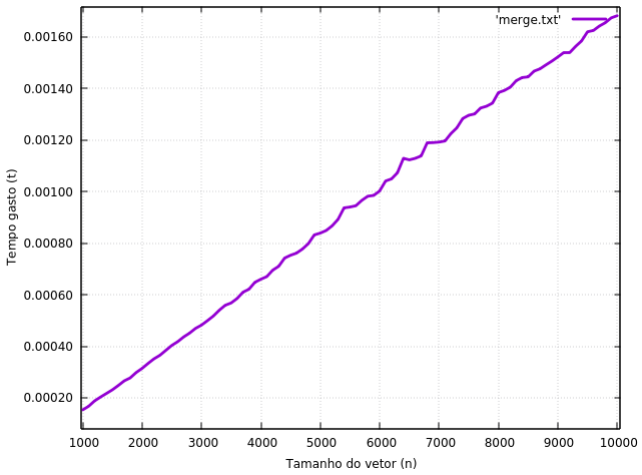


Figure 7: Gráfico de tempo de execução do merge sort

Na figura 7 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 à 10000. O tempo de execução se asemelha em algo linear, já que é um um logarítimo multiplicando n.

2.3 Quick sort

O Quicksort 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 fique 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 fique na sua posição certa. A partir daí começa as comparações do pivô e os elementos do início e do fim do vetor. O quick sort tem complexidade $\Theta\left(n^2\right)$ no pior caso e $\Theta\left(nlogn\right)$ em seu melhor e médio caso.

```c
void quickSort(int *v, int s, int e){
  int p;
  if(s < e){
    p = partition(v,s,e);
    quickSort(v,s, p-1);
    quickSort(v,p+1, e);
  }
}

int partition(int *v, int s, int e){
  int l = s, i , aux;
  for(i=s; i<e; i++){
    if(v[i]<v[e]){
      aux= v[i];
      v[i] = v[l];
      v[l] = aux;
      l++;
    }
  }
  aux = v[e];
  v[e] = v[l];
  v[l] = aux;
  return l;
}
```

Figure 8: Código em C do algoritmo quick sort

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

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

então

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

é possível observar que é obtida uma relação de recorrência, onde para se chegar no caso base, o vetor deve ser unitário ou nulo, nesses caso somente a linha C_1 será executada, então

$$T_b(0) = C_1$$
$$T_b(1) = C_1$$

iniciando a próxima equação da recorrência para tentar reconhecer um padrão

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

substituindo na equação original

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

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

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

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

$$7a + 8T_b\left(\frac{n-7}{8}\right) + 4T^p\left(\frac{n-3}{4}\right) + 2T^p\left(\frac{n-1}{2}\right) + T^p(n)$$

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

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

neste ponto é necessário igualar a variante igualar o termo $T_b\left(\frac{n-(2^x-1)}{2}\right) = 1$ para descobir o valor de x que encerra essa recorrência, então

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

fazendo um processo de isolamento se obtém

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

substituindo na equação, se tem

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

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

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

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

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

substituindo se tem

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

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

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

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

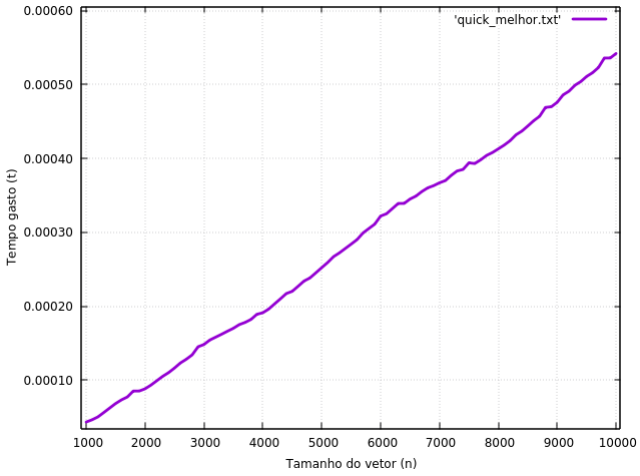


Figure 9: Gráfico de tempo de execução do melhor caso do algoritmo quick sort em seu melhor caso

na figura 9 é possível visualizar graficamente 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 da 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

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

portanto,

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

é possível observar que é obtida uma relação de recorrência, onde para se chegar no caso base, o vetor deve ser unitário ou nulo, nesses caso somente a linha C_1 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

$$T_w(n) = a + [a + T_w(n-2) + T^p(n-1)]\, T^p(n) =$$

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

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

$$2a + [a + T_w(n-3) + T^p(n-2)] + T^p(n) + T^p(n-1) =$$

$$3a + T_w(n-3) + \sum_{i=0}^{n-2} T^p(n-i) =$$

$$(n-1)\, a + T_w(1) + \sum_{i=0}^{n-2} z(n-i) + y =$$

$$n - a - a + C_1 + (n-1)\, y + z \sum_{i=2}^{n} i =$$

$$n\left(a+y\right)-y-a+C_1+z\left(\frac{n}{2}\left(n+1\right)-1\right)=$$

$$n\left(a+y\right)-y+a+C_1-z+z\frac{n}{2}\left(n+1\right)=$$

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

$$\frac{zn^2}{2}+\left(a+y+\frac{z}{2}\right)n+C_1-y-a-z$$

neste ponto já é possível observar que o pior caso do algoritmo quick sort conta com termos quadráticos, ou seja, sua complexidade é $\Theta\left(n^2\right)$.

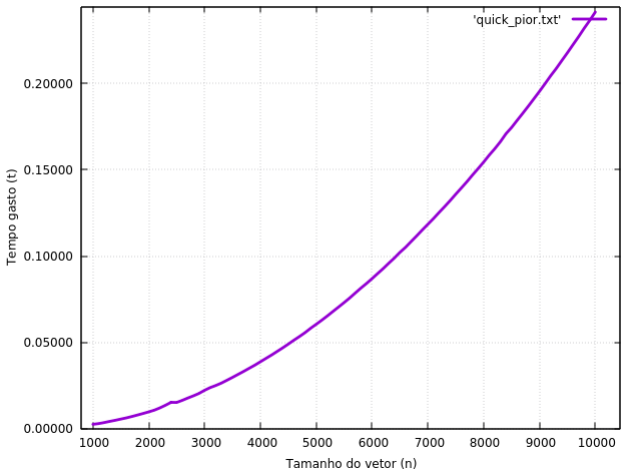


Figure 10: Gráfico de tempo de execução do algoritmo quick sort em seu pior caso

na figura 10 vemos 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 $\Theta\left(nlogn\right)$.

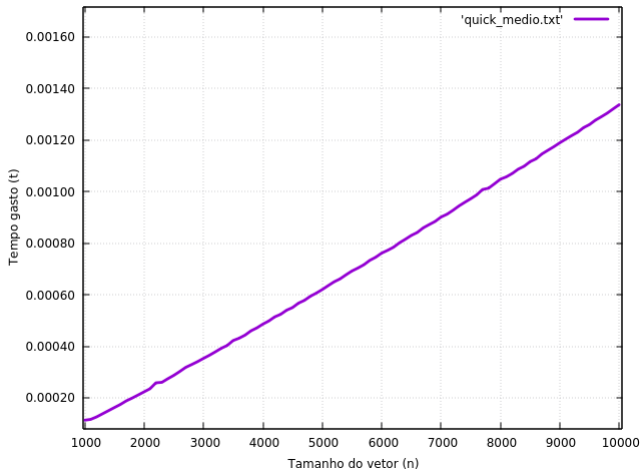


Figure 11: Gráfico de tempo de execução do algoritmo quick sort em seu caso médio

Na figura 12 é possível analisar todos os casos do quick sort graficamente, entretando, em comparação ao melhor e médio caso o pior caso tem um tempo de execução muito maior, por se tratar de um algorítmo quadrático.

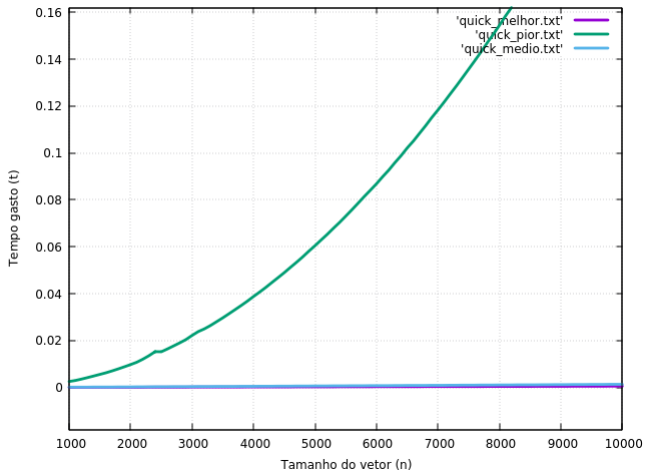


Figure 12: Gráfico comparativo entre os casos do quick sort

Na figura 13 é possível visualizar de forma mais clara a comparação entre o caso médio e melhor caso do quick sort.

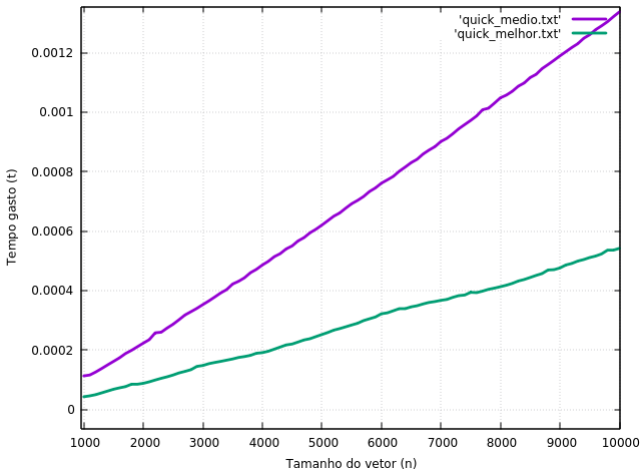


Figure 13: Gráfico comparativo entre o melhor e médio caso do quick sort

2.4 Bubble sort

O algoritmo bubble sort é um dos mais simples dos algoritmos de ordenação, a ideia é percorrer o vetor diversas vezes, verificando a cada passagem se o proximo elemento do vetor é ou não maior com o qual está sendo comparado, caso for, fazer a troca. Sua complexidade em geral é $\Theta\left(n^2\right)$.

```
void *bubble (int *v, int n){
 int aux, i, j;
 for(i = 0; i < n; i++){
   for(j =0; j < n; j++){
     if(v[i] < v[j]){
       aux = v[i];
       v[i] = v[j];
       v[j] = aux;
     }
   }
 }

}
```

Figure 14: Código em C do algoritmo bubble sort

É possível equacionar a complexidade o bubble sort da seguinte maneira

$$T_w\left(n\right) = C_1 + \left(n+1\right)C_2 + n\left(n+1\right)C_3 + n^2C_4 + (\frac{n^2-n}{2})C_{5,6,7} =$$

$$C_1 + nC_2 + C_2 + nC_3 + n^2C_3 + n^2C_4 + (\frac{n^2-n}{2})C_{5,6,7} =$$

$$n^2\left(C_3 + C_4\right) + (\frac{n^2-n}{2})C_{5,6,7} + n\left(C_2 + C_3\right) + C_1 + C_2$$

é interessante notar como claramente a equação toma forma de algo quadrático $ax^2 + bx + c$.

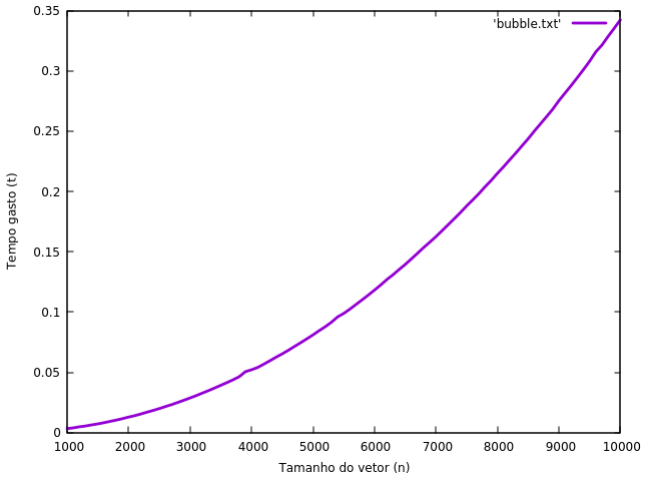


Figure 15: Gráfico de tempo de execução do melhor caso do algoritmo bubble sort em seu pior caso

Na figura 15 é possível visualizar graficamente que seu tempo de execução em função de um vetor tamanho variando entre 1000 e 10000 posições é $\Theta\left(n^2\right)$. É possível fazer uma pequena alteração no bubble sort para que o mesmo tenha ordem linear quando o vetor já estiver ordenado, para isso adiciona-se uma linha para verificar se houve alguma troca na primeira execução do laço interno, caso não tenha, significa que o vetor já está ordenado, podendo assim encerrar a execução.

2.5 Couting sort

O couting sort é um algoritmo de ordenação estável não baseado em troca de posições, e sim em contagem de posições e realocamento, sua complexidade é $\Theta\left(n^2\right)$.

```c
int *couting (int *v, int n){
   int *w = new_0(n), i, j, k;
   int *c = new_0(n);

   for(i = 0; i < n; i++){
      for(j = 0; j < n; j++){
         if(v[j] < v[i]){
            c[i]++;
         }
      }
   }
   for(k = 0; k < n; k++ ){
      w[c[k]] = v[k];
   }
   return w;
}
```

Figure 16: Código em C do algoritmo couting sort

a função new_0 chamada no algoritmo da figura 16 tem como finalidade preencher um vetor auxiliar com zeros. A equação que define a complexidade do couting sort, que não tem melhor e pior caso definido, pode ser dada por

$$T(n) = C_1+C_2+(n+1)C_3+n(n+1)C_4+n^2C_5+\left(\frac{n^2-n}{2}\right)C_6+(n+1)C_7+nC_8+C_9 =$$

$$C_1+C_2+nC_3+C_3+(n^2+n)C_4+n^2C_5+\frac{C_6}{2}n(n-1)+C_7+nC_7+nC_8+C_9 =$$

$$C_1+C_2+nC_3+C_3+n^2C_4+nC_4+n^2C_5+\frac{C_6}{2}n(n-1)+C_7+nC_7+nC_8+C_9 =$$

$$n^2(C_4+C_5)+n\left(C_3+C_4+C_7+C_8+\frac{C_6}{2}(n-1)\right)+C_1+C_2+C_3+C_7+C_9$$

como visto, o maior termo é quadrático, confirmando assim sua complexidade $\Theta(n^2)$.

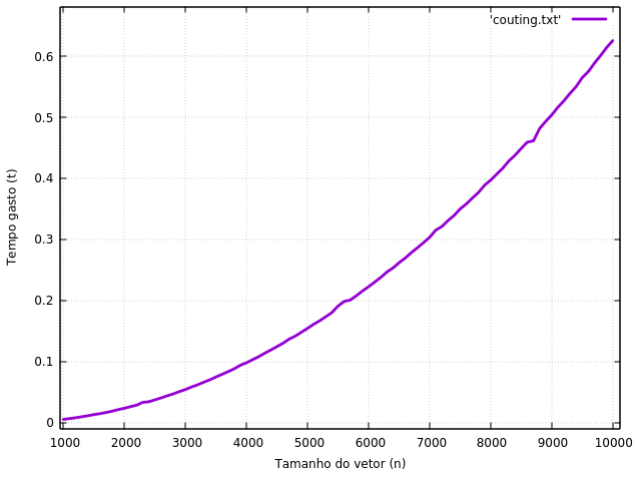


Figure 17: Gráfico de tempo de execução do couting sort

na figura 17 é possível visualizar o tempo de execução do algorítmo em função de vetores com tamanho entre 1000 e 10000.

2.6 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.

```
int *distribution (int *v, int n){
   int l = min(v,n);
   int b = max(v,n);
   int k = b-l+1;
   int i, j;
   int *w = new_0(k);
   int *y = new_0(n);

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

    for (j = 1; j <= b-l;j++){
       w[j] = w[j] + w[j-1];
    }

    for (i = 0;i <= n-1; i++){
        j = w[v[i]- l];
        y[j-1] = v[i];
        w[v[i]-l]--;
    }

   return y;
   free(y);
   free(w);
}
```

Figure 18: Código em C do algoritmo distribution sort

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

$$T(n) = a + (n+1)C_7 + nC_8 + (k+1)C_9 + kC_{10} + (n+1)C_{11} + n(C_{12} + C_{13} + C_{14}) =$$

$$a + nC_7 + C_7 + nC_8 + kC_9 + C_9 + kC_{10} + nC_{11} + C_{11} + n\left(C_{12} + C_{13} + C_{14}\right) =$$
$$k\left(C_9 + C_{10}\right) + n\left(C_7 + C_8 + C_{11} + C_{12} + C_{13} + C_{14}\right) + C_7 + C_9 + C_{11} + a$$

e podemos observar que é provada sua 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.

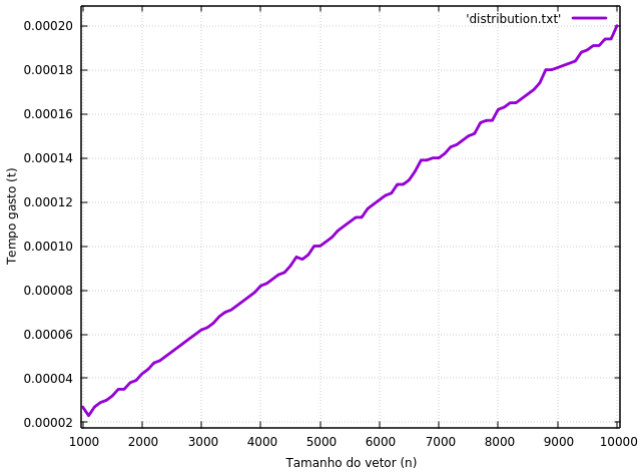
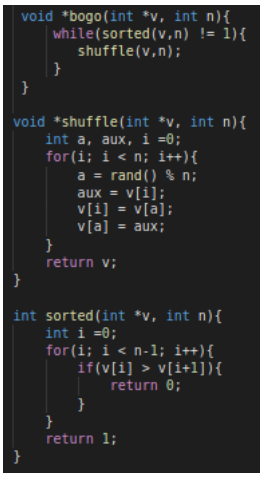


Figure 19: Gráfico de tempo de execução do distribution sort

2.7 (Extra) Bogo sort

O bogo sort é um algoritmo não estável de ordenação ineficiente que se baseia em reordenação aleatória de elementos. A complexidade esperada deste algoritmo é $O\left(nxn!\right)$ e seu tempo de execução esperado depende da quantidade de operações realizadas até que o vetor esteja ordenado.



```c
void *bogo(int *v, int n){
    while(sorted(v,n) != 1){
        shuffle(v,n);
    }
}

void *shuffle(int *v, int n){
    int a, aux, i =0;
    for(i; i < n; i++){
        a = rand() % n;
        aux = v[i];
        v[i] = v[a];
        v[a] = aux;
    }
    return v;
}

int sorted(int *v, int n){
    int i =0;
    for(i; i < n-1; i++){
        if(v[i] > v[i+1]){
            return 0;
        }
    }
    return 1;
}
```

Figure 20: Código em C do algoritmo bogo sort

Este algoritmo depende da sorte e acaso para ordenar o vetor, entretanto é possível ter noção da probabilidade de do vetor gerado já está ordenado. Para isso, considera-se que a probabilidade de uma dada ordenação qualquer gerada aleatoriamente ser correta é de $\frac{1}{n!}$ (desprezando a multiplicação por n), então, sendo n o tamanho do vetor, se obtém as seguintes probabilidades

n	$n!$	Probabilidade do vetor está ordenado
1	1	100%
2	2	50%
3	6	$16,66\%$
4	24	$4,16\%$
5	120	$0,83\%$
6	720	$0,138\%$
7	5040	$0,01984126\%$
8	40320	$0,00248015\%$
9	362880	$0,00027557\%$
10	3628800	$0,00002755\%$
15	1307674368000	$0,000000000007647\%$
20	2432902008176640000	$0,00000000000000000411\%$
25	15511210043330985984000000	$0,0000000000000000000000006447\%$
30	$2,6\left(10^{32}\right)$	$0,000000000000000000000000000003699\%$

Observa-se que a probabilidade de a sorte e o acaso escolher a ordenação correta piora muito rapidamente cada vez que um elemento é adicionado. A figura 21 mostra o tempo de execução do algoritmo em função de um vetor que varia entre 3 e 13 posições.

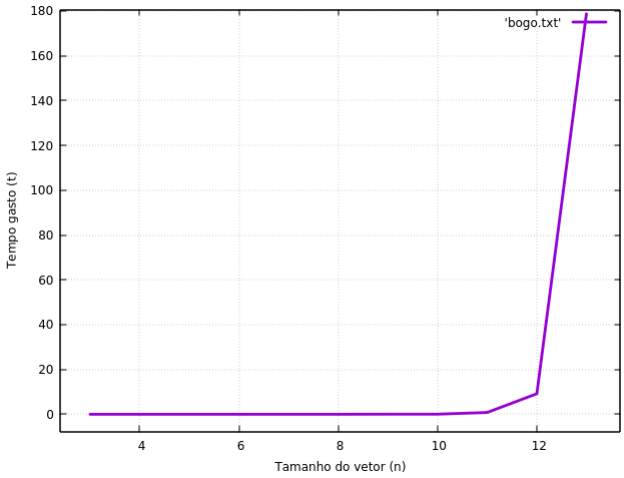


Figure 21: Gráfico de tempo de execução do algoritmo bogo sort

Para se ter uma base, uma máquina capaz de embaralhar e testar a ordenação de um vetor um bilhão de vezes por segundo demoraria cerca de $245,7$ milhões de anos para ordenar o vetor (considerando que uma mesma ordenação não é sorteada duas vezes), enquanto um vetor de 30 posições demoraria cerca de $4,2$ quadrilhões de anos para ser ordenado.

2.8 Comparação de desempenho

O quick sort é considerado por muitos o melhor algoritmo de ordenação, entretando, existem casos e casos, em determinadas circustâncias possa ser que outro algoritmo traga resultados mais satisfatórios. Essa seção tratará de comparar as complexidades dos algoritmos e mostrar qual é o melhor dada a circunstância.

Para a comparação ser justa, todos algorítmos quadráticos serão comparados na ordem 1; na ordem 2 será feita uma comparação entre os casos médios dos algoritmos, aqueles que não tem caso médio definido também entrarão nessa lista; na ordem 3 serão comparados os algoritmos logarítmicos e lineares, e na ordem 4 serão mostrados todos algoritmos com todos seus casos.

2.8.1 Ordem 1

Iniciando com um gráfico com todos tempos de execução

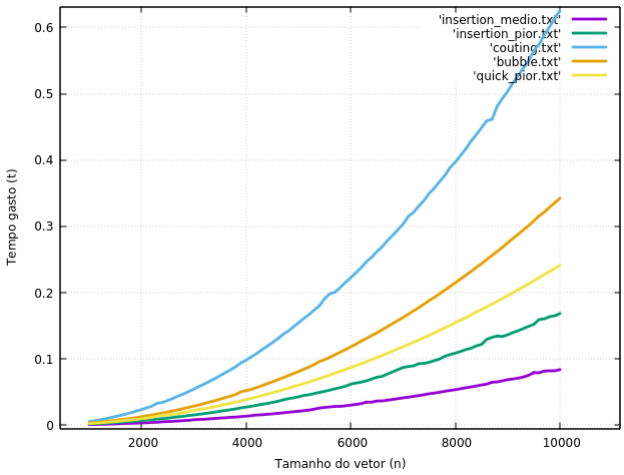


Figure 22: Gráfico de tempo de execução dos algoritmos de ordem quadrática

Vemos que, de todos os algoritimos 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. O couting sort é o algoritmo com maior tempo de execução, consequentemente pode-se considerá-lo o pior dessa ordem, e, tendo como base que a ordem quadrática é a com maior tempo de execução dentre as demais, pode-se dizer que é o pior algorítmo mostrado nesse trabalho, inclusive pior que o bubble sort, que vem logo atrás. O quick sort em seu pior caso é o meio termo deste gráfico.

2.8.2 Ordem 2

Iniciando com um gráfico com todos os tempos de execução

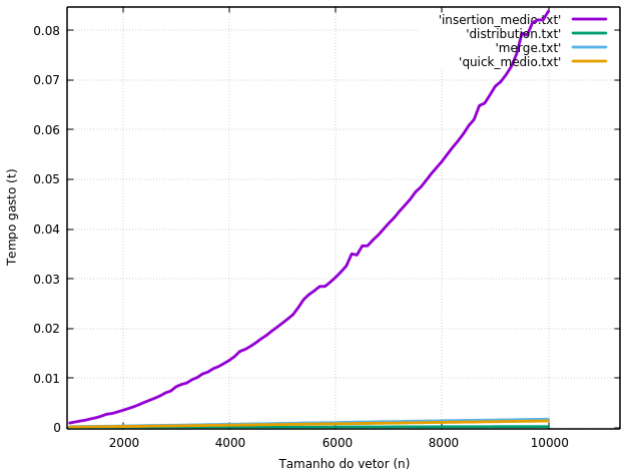


Figure 23: Gráfico de tempo de execução dos algoritmos em seu caso médio/que não tem caso definido

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 $nlogn$, desfigurando o gráfico.

2.8.3 Ordem 3

Iniciando com um gráfico com os tempos de execução se tem

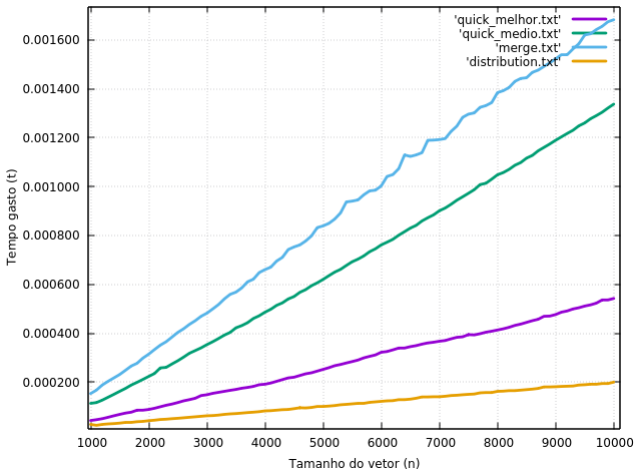


Figure 24: Gráfico de tempo de execução dos algoritmos lineares

E, toma-se como conclusão, como dito anteriormente neste artigo que o dis-

tribution sort é o algoritmo mais rápido dentre os apresentados, inclusive que o quick sort em seu melhor caso, entretando ressalva-se que 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. Em casos especiais em que o gasto de memória não será um problema o algoritmo distribution sort é o mais recomendado, nos demais casos, o quick sort tem melhor eficiência.

2.8.4 Ordem 4

A seguir temos o gráfico com todos os tempos de execução dos algoritmos mostrados neste trabalho

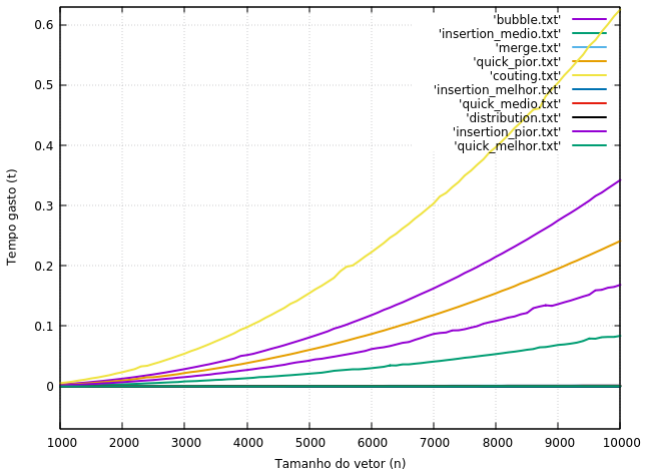


Figure 25: Gráfico de tempo de execução de todos algoritmos apresentados neste trabalho

Como mencionado anteriormente, o couting sort é o algoritmo com maior tempo de execução apresentado neste trabalho. Na figura 25, é possível visualizar a diferença de tempo de execução dos algoritmos de ordem quadrática para aqueles que tem ordem linear que parecem ser constantes. Embora não de para distinguir bem no gráfico da figura 25, já foi provado anteriormente também que, dentre os algoritmos apresentados o distribution sort é aquele que tem menor tempo de execução seguido do quick sort no caso médio.

3 Algoritmos de busca

Algoritmos de busca 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. Nesta seção será apresentados e comparados dois algoritmos, sendo um de busca linear e outro de busca binária.

3.1 Busca linear

Chama-se busca linear aquela que 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.

```c
int search(int *v, int n, int x){

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

 return -1;

}
```

Figure 26: Código em C do algoritmo busca linear

Seu melhor caso ocorre quando o algoritmo encontra o elemento especificado na primeira verificação, tornando a função constante independente do tamanho do vetor, logo a equação que o define é dada por

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

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 na figura 27.

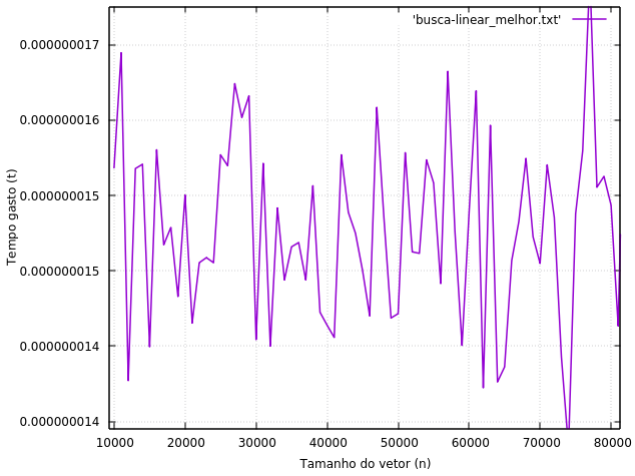


Figure 27: Gráfico de tempo de execução da busca linear em seu melhor caso

O pior caso da busca linear ocorre quando o elemento à ser procurado não está presente no vetor, assim, sua complexidade é igual a n e pode ser equa-

cionada como

$$T(n) = C_1 + (n+1)\,C_2 + nC_3 + C_5 =$$
$$n\,(C_2 + C_3) + C_1 + C_2 + C_5$$

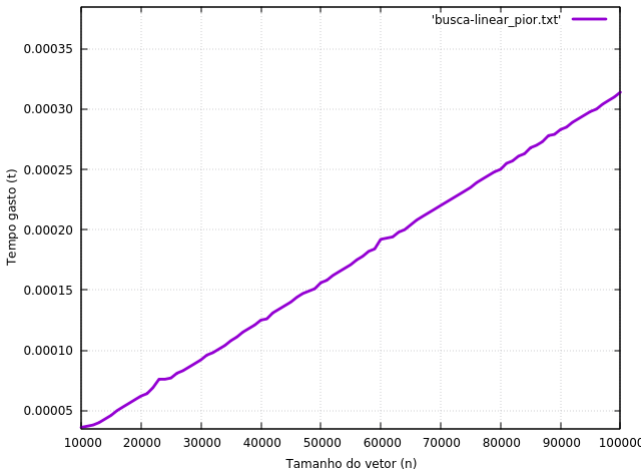


Figure 28: Gráfico de tempo de execução da busca linear em seu pior caso

O gráfico da figura 28 mostra seu tempo de execução em função de um vetor que varia entre dez mil e cem mil posições.

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.

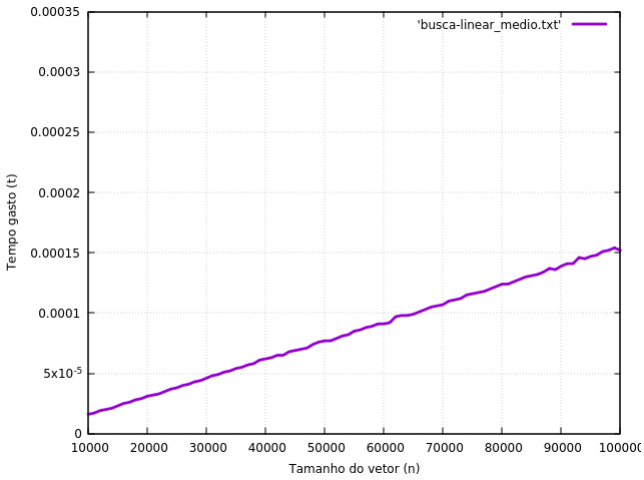


Figure 29: Gráfico de tempo de execução da busca linear em seu caso médio esperado

A figura 30 mostra a comparação entre os casos da busca linear.

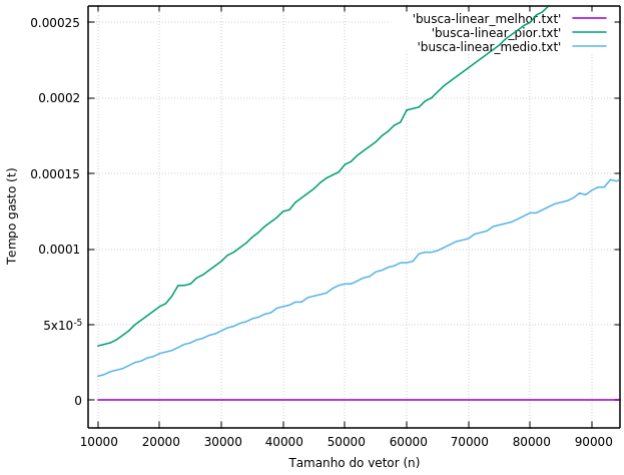


Figure 30: Comparação entre os casos do algoritmo busca linear

3.2 Busca binária

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

```c
int search(int *v, int s, int e, int k){
 int m = (s+e)/2;
 if(s <= e){
   if(v[m] == k){
     return m;
   }else if (v[m] > k){
     search(v, s, m-1, k);
   }else{
     search(v, m+1, e, k);
   }
 }else{
   return -1;
 }
}
```

Figure 31: Código em C do algoritmo busca binária

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

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

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 na figura 32.

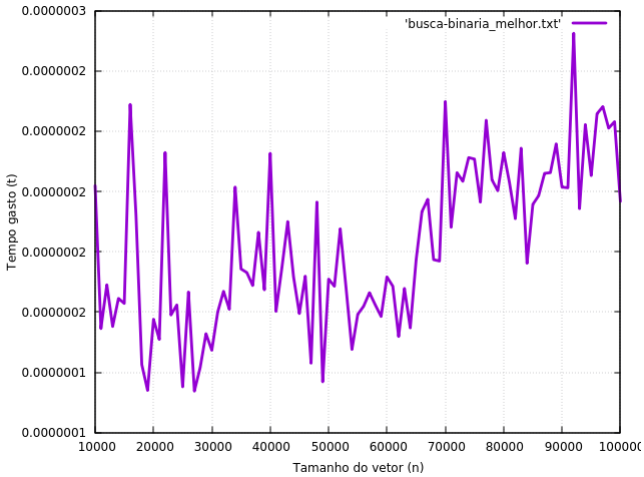


Figure 32: Gráfico de tempo de execução da busca binária em seu melhor caso

O pior caso da busca binária, assim como a linear, ocorre quando o elemento à ser procurado não está presente no vetor, assim, sua ordem é logarítmica e pode ser equacionada como

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

percebe-se que é encontrada uma relação de recorrência, que tem como caso base um vetor com 0 posições, neste se tem

$$T_w(0) = C_1 + C_2 + C_9 + C_{10}$$

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

$$T_w\left(\frac{n-1}{2}\right) = a + T_w\left(\frac{n-3}{4}\right)$$

substituindo na equação original

$$T_w(n) = a + \left[a + T_w\left(\frac{n-3}{4}\right)\right] =$$

$$2a + T_w\left(\frac{n-3}{4}\right)$$

calculando a próxima relação se obtém

$$T_w\left(\frac{n-3}{4}\right) = a + T_w\left(\frac{\frac{n-3}{4}}{2} - 1\right) =$$

$$a + T_w\left(\frac{n-7}{8}\right)$$

substituindo na equação original

$$T_w(n) = a + \left[a + \left[a + T_w\left(\frac{n-7}{8}\right)\right]\right] =$$

$$3a + T_w\left(\frac{n-7}{8}\right)$$

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

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

sendo assim provada que sua ordem é logarítimca, o gráfico que mostra o tempo de execução do algoritmo pode ser visto na figura 33.

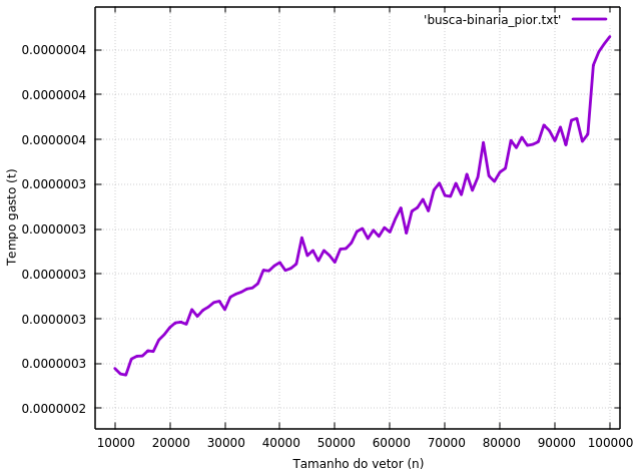


Figure 33: Gráfico de tempo de execução da busca binária em seu pior caso

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 figura 34.

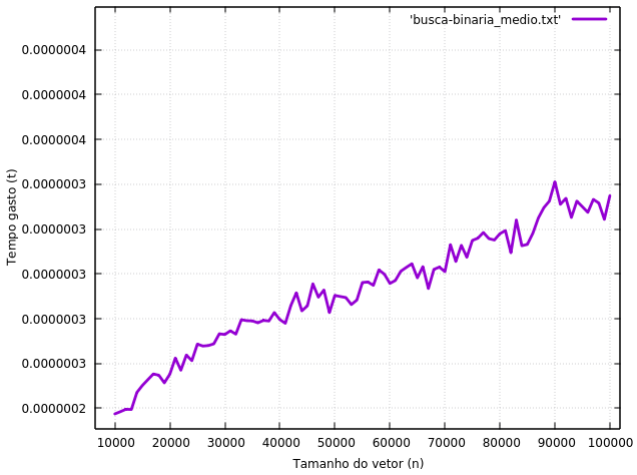


Figure 34: Gráfico de tempo de execução da busca binária em seu caso médio

O gráfico da figura 35 mostra a comparação entre os casos do algoritmo de busca binária.

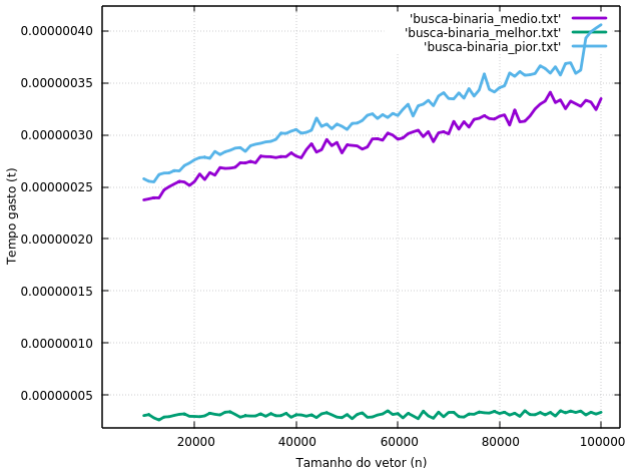


Figure 35: Comparação entre os casos do algoritmo busca linear

3.3 Comparação de desempenho

Como visto nas seções anteriores sabe-se que a complexidade da busca linear é n enquanto a da busca binária é $logn$. Aqui serão mostrados gráficos comparativos entre os casos dos algoritmos. Na ordem 1 será comparado o melhor caso de ambos; na ordem 2 será comparado o caso médio de ambos; na ordem 3 será

comparado o pior caso de ambos, e por fim, na ordem 4 será apresentado um gráfico com ambos os algoritmos e todos seus casos.

3.3.1 Ordem 1

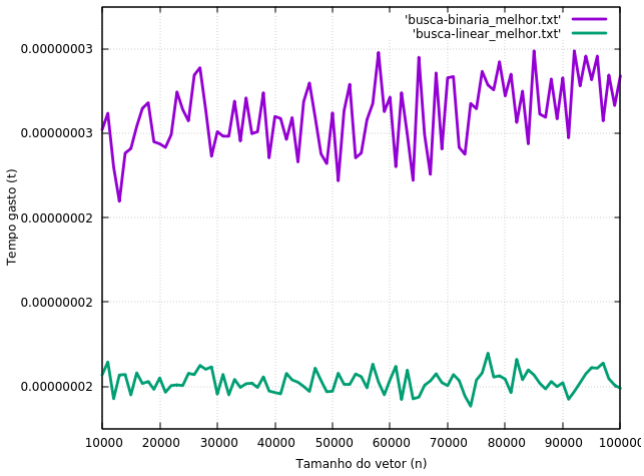


Figure 36: Gráfico de tempo de execução dos algoritmos em seu melhor caso

É possível visualizar na figura 36 que a busca linear em seu melhor caso leva vantagem sobre a busca binária em seu melhor caso, isso ocorre por conta que a quantidade de verificações é menor na busca linear.

3.3.2 Ordem 2

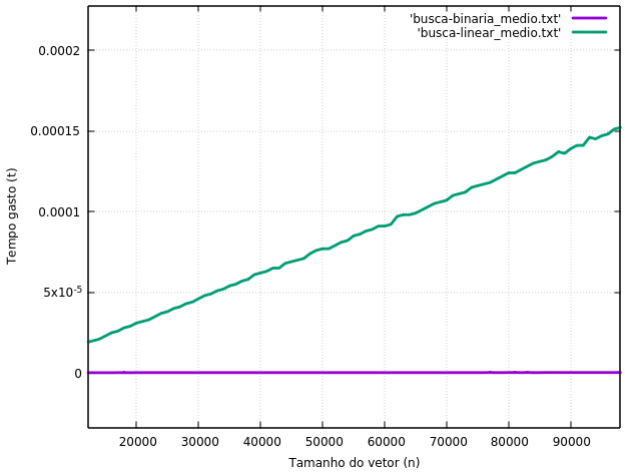


Figure 37: Gráfico de tempo de execução dos algoritmos em seu caso médio

Ao contrário do melhor caso que é basicamente constante, o caso médio dos algoritmos já adquire sua ordem, ou seja, n para linear e $logn$ para logarítmica. O gráfico da figura 37 mostra a grande vantagem que a busca logarítmica leva sobre a linear em relação à tempo de execução.

3.3.3 Ordem 3

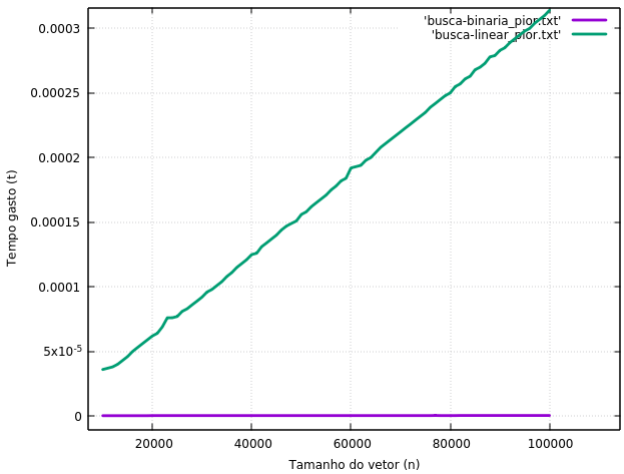


Figure 38: Gráfico de tempo de execução dos algoritmos em seu pior caso

Assim como no caso médio, o pior caso da busca binária tem tempo de execução muito inferior à busca linear.

3.3.4 Ordem 4

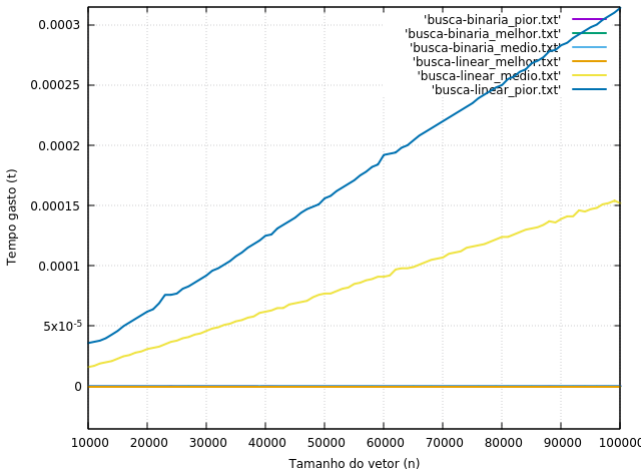


Figure 39: Gráfico de tempo de execução de todos os casos de ambos algoritmos de busca

Na figura 39 é confirmado que o pior caso da busca linear é com o maior tempo de execução seguido do caso médio da mesma. Bem distante desses tempos de exeução estão os casos do algoritmo busca binária e o melhor caso da busca linear. Já foi visto anteriormente que o melhor caso da busca linear é o com menor tempo de execução dentre os demais.

4 Conclusão

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

Dentre os algoritmos de ordenação apresentados, o couting sort é o com maior tempo de execução e também necessita de um extra de memória para criar um vetor auxiliar então pode ser considerado o pior dentre os demais. O bubble sort assim como o couting é quadrático, entretando, não cria vetores auxiliares e tem menor tempo de execução por possuir um laço de repetição a menos, é melhor que o couting sort mas pior que os demais algoritmos. O insertion sort é o melhor dos algoritmos de ordem quadrática dentre os apresentados neste trabalho, seu tempo de execução é inferior ao couting e ao bubble e não utiliza vetores auxiliares em seu método, entretanto, em comparação com outros algoritmos de ordem linear tem tempo de execução muito alto. O merge sort, de ordem $nlogn$ tem tempo de execução inferior a todos os algorimos 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 $nlogn$, 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 algorimo a se utilizar. O distribution sort é o algoritmo mais rápido dentre os outros, entretando, seu uso de memória é em função da diferença do maior e menor elemento do vetor, tornando seu uso perigoso. Em casos que o consumo de memória não é o problema, o distribution sort é o melhor algoritmo a se utilizar.

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.

References

[1] Levitin, A.. Introduction to the Design and Analysis of Algorithms. 2. Addison Wesley. 2006

[2] Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C.. Introduction to Algorithms. 3. The MIT Press. 2009