Universidade Federal do Rio Grande do Norte
Centro de Ensino Superior do Seridó
Departmento de Computação e Tecnologia
Curso de Bacharelado em Sistemas de Informação

Trabalho de Estrutura de Dados

Felippe da Silva Guedes

Caicó-RN

Maio de 2023

Felippe da Silva Guedes

Avaliação 1

Trabalho apresentado a disciplina Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da avaliação I.

Orientador:
Prof. Dr. João Paulo de Souza Medeiros

BSI – Bacharelado em Sistemas de Informação
DCT – Departamento de Computação e Tecnologia
CERES – Centro de Ensino Superior do Seridó
UFRN – Universidade Federal do Rio Grande do Norte

Caicó-RN

Maio de 2023

Avaliação 1

Autor: Felippe da Silva Guedes
Orientador(a):Prof. Dr. João Paulo de Souza Medeiros

Resumo

O Trabalho apresentado tem como base fazer algoritmos que foram pedidos pelo professor João Paulo, mostrando os códigos feitos para os algoritmos, comentando e explicando sobre cada um deles, apresentando os gráficos que foram gerados por cada um dos algoritmos e também fazendo comparções entre os gráficos para analise dos tempos de execução dos mesmos.

Palavras-chave: Palavra-chave 1, Palavra-chave 2, Palavra-chave 3.

Sumário

1 Trabalho Avaliativo de Estrutura de Dados

1.1 Introdução

Trabalho abaixo tem como êfanse apresentar as soluções para os algoritmos passados pelo professor para fins avaliativos.

Segundo Mendes, Silveira e Galvão (2008) estudar é foda.

Função é uma regra (HOFFMAN; BRADLEY, 1992).

Segundo Hoffman e Bradley (1992) função é uma regra.

1.2 Organização do trabalho

O trabalho abaixo tem as explicações dos algoritmos seguidas da equeções assintónicas, gráficos gerados por cada código feito e comparações entre os algoritmos.

2 Soluções Das Questões

Este é o capítulo da parte central do trabalho, isto é, o desenvolvimento, a parte mais extensa de todo o trabalho. Aqui estará todas as soluções feitas para apresentação do trabalho.

2.1 Distribution Sort

O ordenamento de distribuição, também conhecido como ordenamento por base, é um algoritmo de ordenação que organiza os elementos com base em seus dígitos ou caracteres individuais. Ele é particularmente eficaz para ordenar números inteiros ou cadeias de caracteres com um número fixo de dígitos ou caracteres.

O ordenamento de distribuição possui uma complexidade de tempo de O(n·k), em que n é o número de elementos na lista e k é o número de dígitos ou caracteres. Portanto, o desempenho do algoritmo depende do número de dígitos ou caracteres. É importante notar que o ordenamento de distribuição é estável, o que significa que a ordem relativa dos elementos com o mesmo dígito ou caractere é preservada durante a ordenação.

Uma das limitações do ordenamento de distribuição é que ele requer espaço adicional para armazenar os recipientes, o que pode ser um problema em situações de restrição de memória. Além disso, o algoritmo é mais eficiente quando o número de dígitos ou caracteres é pequeno.

O ordenamento de distribuição é útil quando há uma grande quantidade de dados e os elementos a serem ordenados possuem uma estrutura fixa, como números inteiros com tamanho fixo ou cadeias de caracteres com comprimento fixo.

Codigo Distribution:

```
1.  int max = array[0];
2.  for (int i = 1; i < tamanho; i++) {
```

```
3.      if (array[i] > max) {
4.          max = array[i];
5.      }
6.  }
```

Calculo Distribution:

O algoritmo Distribution Sort possui uma implementação que consiste em três principais laços. O primeiro laço percorre o array A de tamanho n, o segundo laço percorre o array C de tamanho k, e o terceiro laço percorre novamente o array A. Portanto, podemos expressar o tempo de execução da seguinte forma:

$T(n) = 2n + k$

Ao aplicarmos as diretrizes para análise assintótica, podemos simplificar para:

$T(n) = n + k$

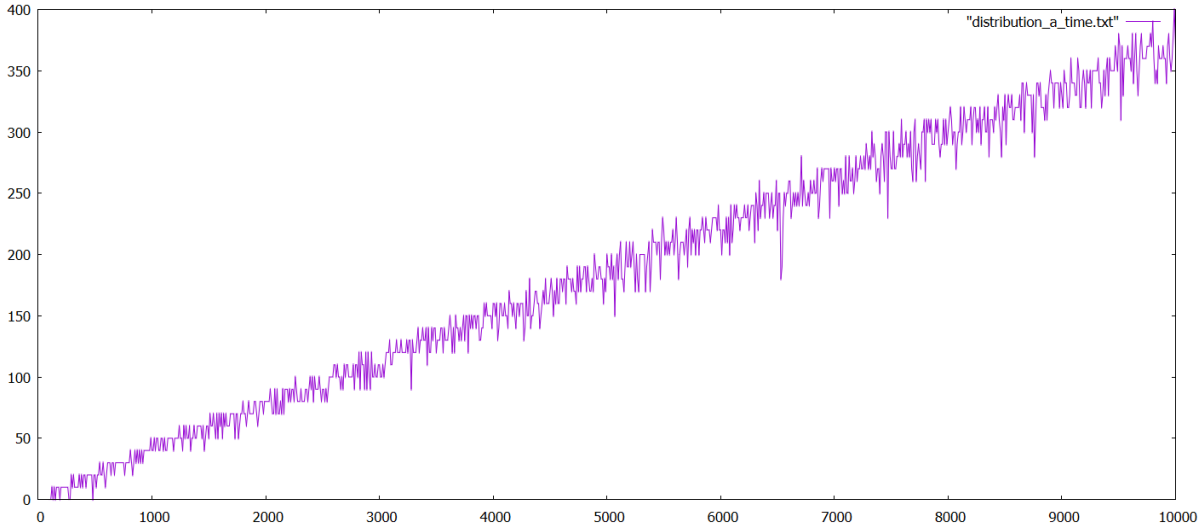
É importante ressaltar que o tempo de execução desse algoritmo é linear em relação aos tamanhos de n e k, e não apenas em relação ao tamanho de n. Essa característica torna o algoritmo significativamente mais eficiente em comparação com outros algoritmos que estudamos. No entanto, é preciso considerar o custo associado ao uso de memória, uma vez que é necessário criar um array de contagem C com tamanho k e o array B resultante com tamanho igual ao array original. Portanto, o consumo de memória também é dado por:

$T(n) = n + k.$

resposta assintótica:

T(n+k)

Gráfico distribution:



2.2 Insertion Sort Melhor Caso

O Insertion Sort é um algoritmo de ordenação simples que percorre uma lista de elementos e os insere no lugar correto dentro de uma sublista já ordenada. No melhor caso para o Insertion Sort, a lista de entrada já está completamente ordenada, o que significa que os elementos estão dispostos em ordem crescente.

Quando a lista já está ordenada, o Insertion Sort se torna altamente eficiente, pois ele precisa apenas comparar cada elemento com o seu predecessor imediato e verificar se precisa ou não ser movido para a posição correta. Como a lista já está ordenada, em cada iteração, o algoritmo compara um elemento com um número cada vez menor de elementos à sua esquerda, até encontrar a posição correta.

No melhor caso, o tempo de execução do Insertion Sort é O(n), onde "n"é o número de elementos na lista. Isso ocorre porque, quando a lista já está ordenada, o algoritmo não precisa realizar muitas trocas ou deslocamentos, pois os elementos já estão em suas posições corretas. Ele apenas faz comparações entre elementos adjacentes para verificar se estão em ordem correta.

Ao percorrer a lista no seu melhor caso, o Insertion Sort identifica que cada elemento já está na posição correta e, portanto, não precisa fazer nenhum movimento. Ele simplesmente avança para o próximo elemento sem alterar a ordem da lista.

Embora o melhor caso do Insertion Sort seja muito eficiente, é importante notar que seu desempenho piora à medida que a lista se torna mais desordenada. Em casos médios e piores, o tempo de execução do algoritmo é $O(n^2)$, tornando-o menos eficiente em comparação com outros algoritmos de ordenação, como o Merge Sort ou o Quick Sort.

Em resumo, no melhor caso do Insertion Sort, em que a lista já está ordenada, o algoritmo executa em tempo linear O(n), sem a necessidade de fazer qualquer movimento de elementos. No entanto, é importante considerar o desempenho do algoritmo em outros cenários para escolher a melhor abordagem de ordenação, dependendo da natureza dos dados a serem ordenados.

Codigo Insertion Sort Melhor Caso:

```
1. int i, key, j;
2.  for (i = 1; i < n; i++) {

3.       key = arr[i];
4.       j = i - 1;
5.       while (j >= 0 && arr[j] > key) {
6.           arr[j + 1] = arr[j];
7.           j = j - 1;
8.       }
9.       arr[j + 1] = key;
10.   }
11. }
```

Calculo Insertion Melhor Caso:

Para ordenar uma matriz de forma aleatória, no algoritmo - Insertion Sort-, se utiliza de $\frac{1}{4}N^2$ para comparar e $\frac{1}{2}N^2$ para trocar.

$$\sum_{1}^{\text{i}} k = \frac{1}{i} \cdot \frac{i(i+1)}{2} = \frac{i+1}{2}$$

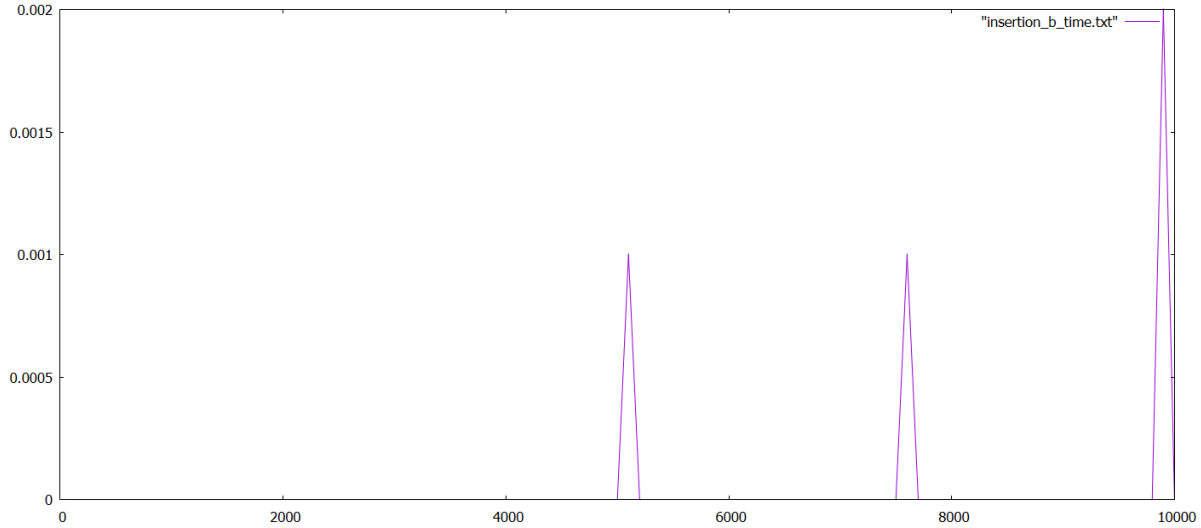
Para o melhor caso (itens já ordenados) temos;

$$\sum_{i=1}^{\text{n-1}} 1 = n - 1 = O(n)$$

Resposta Assintótica:

O(n)

Grafico Insertion Sort Melhor Caso:



2.3 Insertion Sort Caso Médio

No caso médio do Insertion Sort, a lista de entrada não está completamente ordenada, mas também não está completamente desordenada. Isso significa que os elementos estão dispostos em uma ordem aleatória ou com algum grau de desordem.

O Insertion Sort funciona percorrendo a lista a partir do segundo elemento e, em cada iteração, comparando-o com os elementos à sua esquerda. Ele continua avançando e inserindo cada elemento em sua posição correta dentro da sublista ordenada. No caso médio, espera-se que haja uma quantidade moderada de elementos fora de ordem, resultando em um número moderado de comparações e deslocamentos.

O tempo de execução do Insertion Sort no caso médio é $O(n^2)$, onde "n"é o número de elementos na lista. Isso ocorre porque, em média, o algoritmo precisará fazer comparações e, potencialmente, deslocar elementos em cada iteração. À medida que a lista se torna mais desordenada, o número de comparações e deslocamentos aumenta, tornando o algoritmo menos eficiente.

O caso médio do Insertion Sort pode ser melhorado se o algoritmo for combinado com técnicas de otimização, como a pesquisa binária para encontrar a posição correta de inserção. Isso reduz o número de comparações necessárias, melhorando o desempenho geral do algoritmo.

Apesar de ter um desempenho pior do que alguns outros algoritmos de ordenação, como o Merge Sort ou o Quick Sort, o Insertion Sort possui algumas vantagens no caso médio. Ele é estável, ou seja, preserva a ordem relativa de elementos iguais, e também é um algoritmo in-place, o que significa que requer uma quantidade mínima de espaço

adicional para executar.

Em resumo, no caso médio do Insertion Sort, a lista de entrada não está completamente ordenada, e o algoritmo executa em tempo quadrático O(n*2), com um número moderado de comparações e deslocamentos. Apesar de suas limitações, o Insertion Sort pode ser útil para listas de tamanho pequeno ou parcialmente ordenadas.

Codigo Insertion Sort Caso Médio:

```
1. int i, key, j;
2. for (i = 1; i < size; i++) {
3.     key = arr[i];
4.     j = i - 1;

5.     while (j >= 0 && arr[j] > key) {
6.         arr[j + 1] = arr[j];
7.         j = j - 1;
8.     }
9.     arr[j + 1] = key;
10.  }
```

Calculo Insertion Caso Médio:

Para ordenar uma matriz de forma aleatória, no algoritmo - Insertion Sort-, se utiliza de $\frac{1}{4}N^2$ para comparar e $\frac{1}{2}N^2$ para trocar.

$$\sum_{1}^{\mathrm{i}} k = \frac{1}{i} \cdot \frac{i(i+1)}{2} = \frac{i+1}{2}$$

Para o caso médio, supõem que todas as permutações de entrada são igualmente prováveis. O caso médio do Insertion sort é equivalente a:

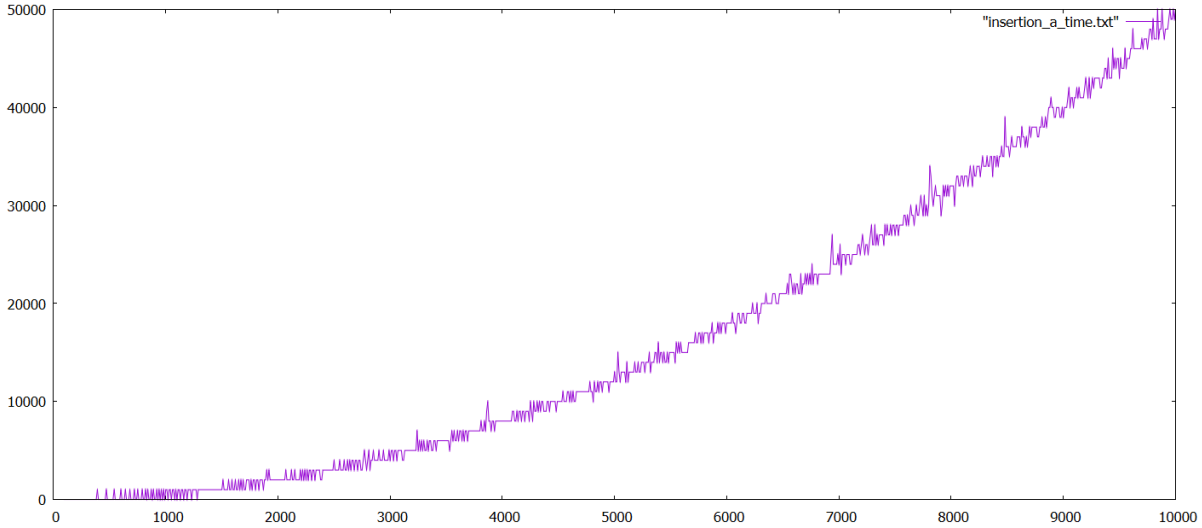
$$B(z) = zB(z) + \frac{z}{2} \sum_{n=>0} zk^k = zB(z) + \frac{1}{2} \cdot \frac{z^2}{(1-z)^2}$$

onde $B(z)$ é a função correspondente ao número de inversões.

Resposta Assintótica:

$O(n^2)$

Grafico Insertion Sort Caso Médio:



2.4 Insertion sort Pior Caso

No pior caso do Insertion Sort, a lista de entrada está completamente reversa ou quase completamente reversa, ou seja, os elementos estão dispostos em ordem decrescente. Isso significa que cada elemento precisa ser movido para o início da lista, resultando em um número máximo de comparações e deslocamentos.

O Insertion Sort percorre a lista a partir do segundo elemento e, em cada iteração, compara o elemento atual com os elementos à sua esquerda até encontrar a posição correta de inserção. No pior caso, cada elemento precisa ser comparado com todos os elementos à sua esquerda antes de ser inserido na posição correta.

O tempo de execução do Insertion Sort no pior caso é $O(n^2)$, onde "n"é o número de elementos na lista. Isso ocorre porque, em cada iteração, o algoritmo precisa fazer uma comparação com cada elemento anterior para encontrar a posição correta. Portanto, o número total de comparações e deslocamentos é proporcional ao quadrado do número de elementos.

O pior caso do Insertion Sort pode ser extremamente ineficiente para listas grandes ou altamente desordenadas. Em comparação com outros algoritmos de ordenação, como o Merge Sort ou o Quick Sort, o Insertion Sort geralmente é considerado mais lento.

No entanto, é importante observar que o Insertion Sort possui algumas vantagens em relação a outros algoritmos. Ele é estável, preservando a ordem relativa de elementos iguais, e é um algoritmo in-place, o que significa que requer uma quantidade mínima de espaço adicional para executar.

Em resumo, no pior caso do Insertion Sort, em que a lista de entrada está completamente reversa, o algoritmo executa em tempo quadrático $O(n^2)$, realizando um número máximo de comparações e deslocamentos. Portanto, em situações em que a lista está fortemente desordenada ou o tamanho da lista é grande, é recomendado considerar outros algoritmos de ordenação mais eficientes.

Codigo Insertion Sort Pior Caso:

```
1. int i, key, j;
2. for (i = 1; i < size; i++) {
3.    key = arr[i];
4.     j = i - 1;

5.     while (j >= 0 && arr[j] > key) {
6.         arr[j + 1] = arr[j];
7.         j = j - 1;
8.     }
9.     arr[j + 1] = key;
10. }
```

Calculo Insertion Sort Pior Caso: Para ordenar uma matriz de forma aleatória, no algoritmo - Insertion Sort-, se utiliza de $\frac{1}{4}N^2$ para comparar e $\frac{1}{2}N^2$ para trocar.

$$\sum_{1}^{\text{i}} k = \frac{1}{i} \cdot \frac{i(i+1)}{2} = \frac{i+1}{2}$$

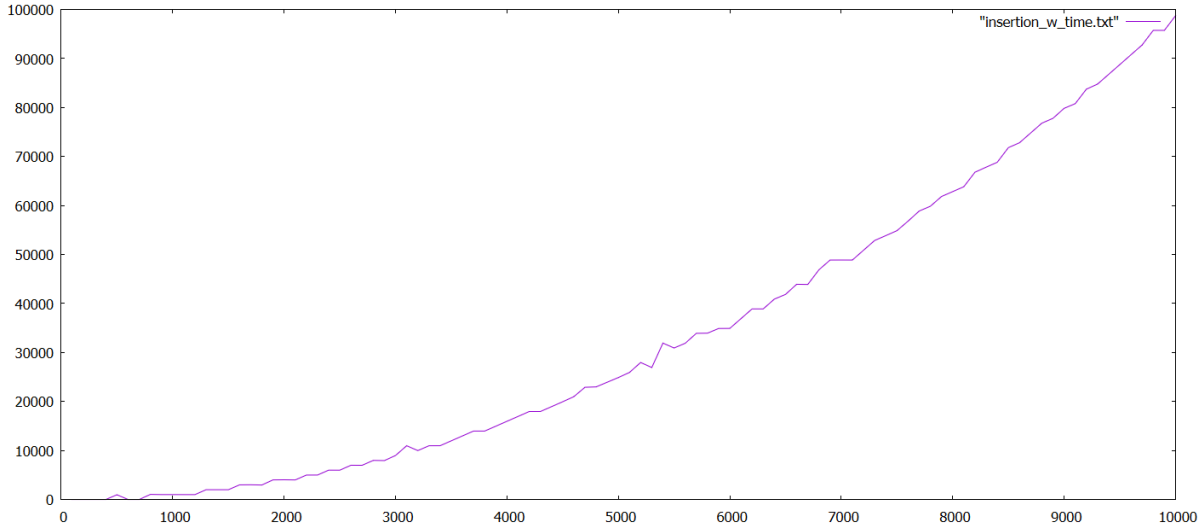
Para o Pior caso temos:

$$\sum_{i=1}^{\text{n-1}} i = \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Resposta Assintótica:

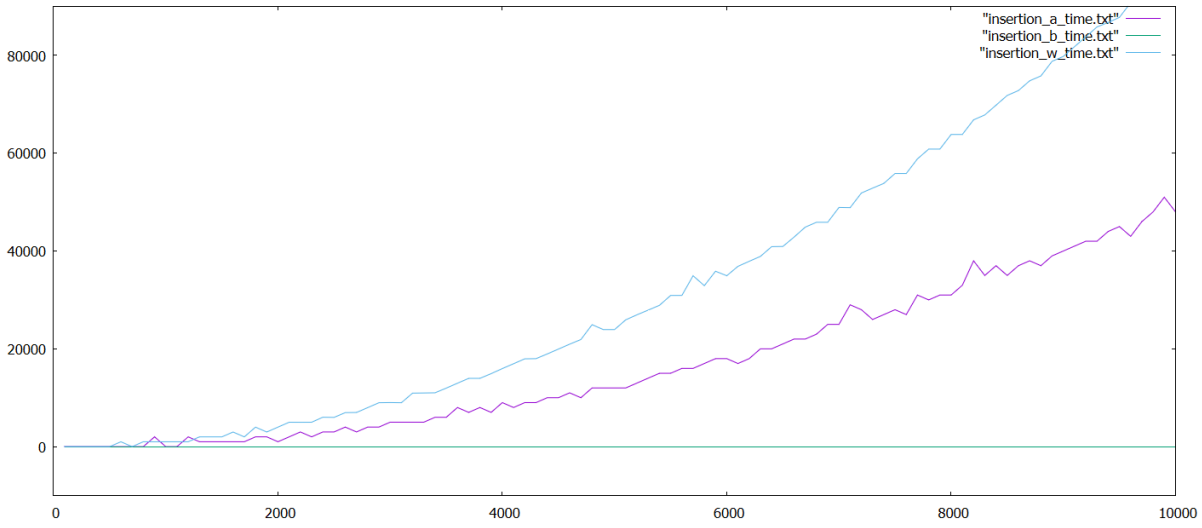
$O(n^2)$

Grafico Insertion Sort Pior Caso:



2.5 Comparação Entre os Graficos

Grafico de Comparação:



Explicação Para Cada Caso:

Melhor caso: O melhor caso ocorre quando o array já está ordenado, ou seja, quando os elementos estão dispostos em ordem crescente. Nesse caso, o Insertion Sort requer apenas uma comparação entre elementos adjacentes em cada iteração e nenhum movimento de elementos é necessário. Graficamente, o algoritmo simplesmente percorre o array uma vez, sem alterar a posição dos elementos. O tempo de execução no melhor caso é O(n), linear, onde "n"é o número de elementos a serem ordenados.

Pior caso: O pior caso ocorre quando o array está ordenado em ordem decrescente, exigindo que todos os elementos sejam deslocados em cada iteração para encontrar sua

posição correta. Graficamente, o algoritmo realizará um grande número de comparações e movimentos de elementos, deslocando-os gradualmente para a posição correta. Isso resulta em um tempo de execução quadrático, $O(n^2)$, onde "n"é o número de elementos a serem ordenados. No pior caso, o Insertion Sort se torna menos eficiente em comparação a outros algoritmos de ordenação mais avançados, como o Merge Sort e o Quicksort.

Caso médio: O caso médio é uma estimativa do desempenho do Insertion Sort com base em uma distribuição aleatória de elementos. Graficamente, o algoritmo realizará um número variável de comparações e movimentos de elementos, dependendo da posição dos elementos no array. No caso médio, o tempo de execução do Insertion Sort também é quadrático, $O(n^2)$. No entanto, em muitos cenários práticos, onde o array não está totalmente desordenado, o Insertion Sort pode apresentar um desempenho aceitável e até mesmo superar outros algoritmos em pequenos conjuntos de dados.

Em resumo, o Insertion Sort tem um desempenho ótimo no melhor caso, onde o array já está ordenado, enquanto apresenta um desempenho menos eficiente no pior caso, onde o array está ordenado em ordem decrescente. O caso médio também tem um tempo de execução quadrático, mas o algoritmo pode ser adequado para conjuntos de dados menores ou quase ordenados. É importante considerar esses diferentes casos ao escolher o algoritmo de ordenação mais apropriado para uma determinada situação.

2.6 Merge sort

O Merge Sort é um algoritmo de ordenação eficiente baseado no paradigma "divide and conquer"(dividir e conquistar). Ele divide a lista de entrada em pequenas partes até que cada parte contenha apenas um elemento. Em seguida, combina essas partes de forma ordenada, até que a lista completa esteja ordenada.

O funcionamento do Merge Sort começa dividindo a lista pela metade recursivamente, até que sejam formadas sublistas com apenas um elemento. Em seguida, essas sublistas são mescladas, combinando-as em pares e ordenando-as ao mesmo tempo. Esse processo de mesclagem é repetido até que a lista completa esteja ordenada.

O tempo de execução do Merge Sort é O(n log n), onde "n"é o número de elementos na lista. Essa complexidade é alcançada porque o algoritmo divide repetidamente a lista pela metade, formando uma árvore de recursão com altura logarítmica. Em cada nível da árvore, todas as sublistas são mescladas, e o número total de elementos processados em cada nível é linear em relação ao tamanho da lista. Portanto, o tempo de execução total é proporcional a n log n.

Uma das principais vantagens do Merge Sort é que ele é um algoritmo estável, ou seja, preserva a ordem relativa de elementos iguais durante a ordenação. Além disso, ele também é um algoritmo in-place, o que significa que requer uma quantidade mínima de espaço adicional para executar. No entanto, é comum usar uma abordagem out-of-place, em que o algoritmo cria novas listas durante o processo de mesclagem.

Devido à sua eficiência e estabilidade, o Merge Sort é amplamente utilizado em situações em que a ordenação precisa ser garantida e o desempenho é importante. No entanto, é importante mencionar que o Merge Sort pode exigir uma quantidade significativa de espaço adicional para a criação das sublistas durante a etapa de mesclagem.

Em resumo, o Merge Sort é um algoritmo de ordenação eficiente que divide a lista em partes menores, ordena essas partes individualmente e depois as mescla de forma ordenada para obter a lista final ordenada. Com uma complexidade de tempo O(n log n), é amplamente utilizado em aplicações que requerem ordenação confiável e desempenho satisfatório.

Codigo Merge Sort:

```
1. int com1 = comeco, com2 = meio + 1, comAux = 0, tam = fim - comeco + 1;
2.     int *vetAux;
```

```
3.      vetAux = (int*)malloc(tam * sizeof(int));

4.      while(com1 <= meio && com2 <= fim) {
5.          if(vetor[com1] < vetor[com2]) {
6.              vetAux[comAux] = vetor[com1];
7.              com1++;
8.          } else {
9.              vetAux[comAux] = vetor[com2];
10             com2++;
11.         }
12.         comAux++;
13.     }

14.     while(com1 <= meio) {
15.         vetAux[comAux] = vetor[com1];
16.         comAux++;
17.         com1++;
18.     }

19.     while(com2 <= fim) {
20.         vetAux[comAux] = vetor[com2];
21.         comAux++;
22.         com2++;
23.     }

24.     for(comAux = comeco; comAux <= fim; comAux++) {
25.         vetor[comAux] = vetAux[comAux - comeco];
26.     }
```

Calculo Merge Sort: $T(n) = \begin{cases} \Theta(1), & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{se } n > 1 \end{cases}$

$T(n) = 2^1 \cdot T\left(\frac{n}{2^1}\right) + 1 \cdot n$

$T(n) = 2^1 \cdot \left(2 \cdot T\left(\frac{n}{2^2} + \frac{n}{2}\right)\right) + n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2 \cdot n$

$T(n) = 2^2 \cdot \left(2 \cdot T\left(\frac{n}{2^3} + \frac{n}{4}\right)\right) + 2 \cdot n = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3 \cdot n$

$\vdots$

$T(n) = 2^{h-1} \cdot \left(2 \cdot T\left(\frac{n}{2^h}\right) + \left(\frac{n}{2^{h-1}}\right)\right) + h \cdot n = 2^h \cdot T\left(\frac{n}{2^h}\right) + h \cdot n$

Para qual

$T(1) = T(\frac{n}{n}) = \Theta(1)$

tendo que fazer com que

$2^h = n \Rightarrow lg2^h = lgn \Rightarrow h = lgn$

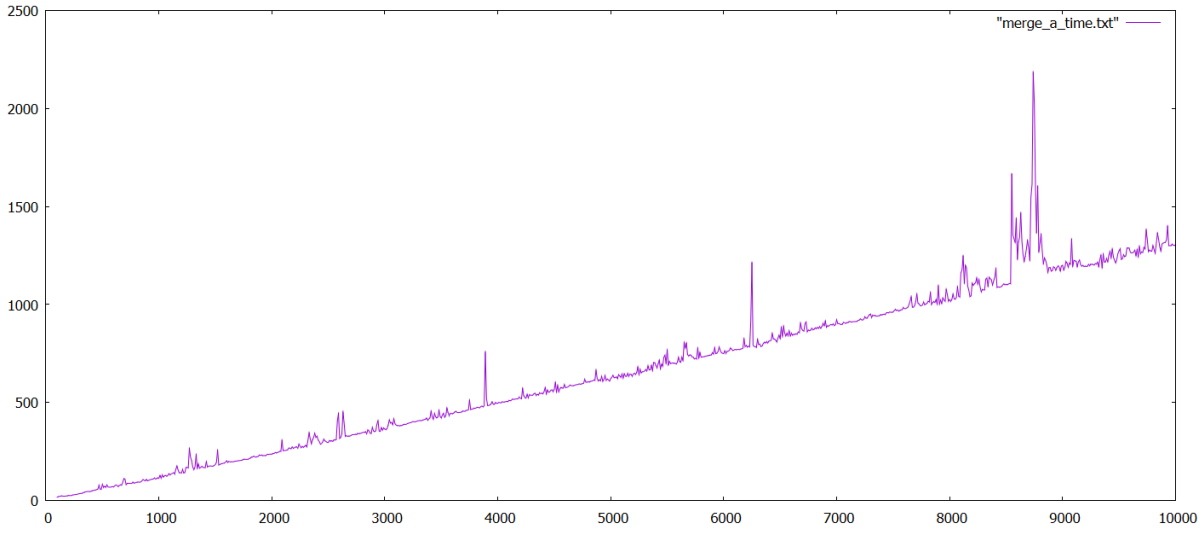
assim:

$T(n) = 2^h T\left(\frac{n}{2^h}\right) + nh = nT\left(\frac{n}{n}\right) + nlgn = nT(1) + nlgn = n\Theta(1) + nlgn = \Theta(nlgn)$

Resposta Assintótica:

O(n log n)

Grafico Merge Sort:



2.7 Quick Sort Melhor Caso

O Quick Sort é um algoritmo de ordenação eficiente que segue o paradigma "divide and conquer"(dividir e conquistar). Ele seleciona um elemento pivô da lista e rearranja os elementos ao seu redor, de forma que os elementos menores que o pivô fiquem à sua esquerda e os elementos maiores fiquem à sua direita. Esse processo é realizado recursivamente até que a lista esteja completamente ordenada.

No melhor caso para o Quick Sort, o pivô escolhido divide a lista em duas partes de tamanhos aproximadamente iguais. Isso resulta em uma divisão balanceada, em que o algoritmo divide a lista de forma uniforme, reduzindo significativamente o tempo de execução.

Quando a divisão é balanceada, o Quick Sort tem um desempenho excepcional, com um tempo de execução médio O(n log n), onde "n"é o número de elementos na lista. Essa complexidade é alcançada porque o algoritmo realiza uma série de divisões balanceadas, reduzindo o tamanho da lista pela metade a cada iteração. Isso garante que o número total de comparações e trocas seja minimizado, resultando em um desempenho rápido.

Além do bom desempenho, o Quick Sort é um algoritmo in-place, o que significa que ele não requer espaço adicional significativo além da própria lista. Isso torna o Quick Sort eficiente em termos de utilização de memória.

No entanto, é importante observar que o desempenho do Quick Sort pode variar dependendo da escolha do pivô. Em cenários desfavoráveis, onde o pivô não divide a lista de forma balanceada, o desempenho pode se degradar para uma complexidade $O(n^2)$. Para mitigar esse problema, técnicas como a escolha do pivô mediano ou o uso de algoritmos de otimização, como o Quick Sort de três vias, podem ser aplicadas.

Em resumo, no melhor caso do Quick Sort, o algoritmo realiza divisões balanceadas, o que resulta em um desempenho excepcional com complexidade O(n log n). Ele é um algoritmo in-place e eficiente em termos de memória. No entanto, é necessário considerar estratégias adequadas para a escolha do pivô, a fim de garantir o desempenho ideal em todos os cenários.

Codigo Quick Sort Melhor Caso:

```
1. void swap(int* a, int* b) {
2.     int t = *a;
3.     *a = *b;
```

```
4.    *b = t;
5. }

6. int partition(int arr[], int low, int high) {
7.     int pivot = arr[high];
8.     int i = (low - 1);

9.     for (int j = low; j <= high - 1; j++) {
10.         if (arr[j] <= pivot) {
11.            i++;
12.             swap(&arr[i], &arr[j]);
13.         }
14.     }
15.     swap(&arr[i + 1], &arr[high]);
16.     return (i + 1);
17. }
```

Calculo Quick Sort Melhor Caso:

$$T(n) = \begin{cases} O(n \log n), & \text{se } v_1 = v_2 \\ O(n^2), & \text{se } v_1 \neq v_2 \end{cases}$$

v_1, v_2 são as partes divididas do vetor.

$$T(n) = T(|left|) + T(|right|) + \Theta(f(n))$$

$$T(n) = T(|left|) + T(|right|) + \Theta(n)$$

O caso ideal de particionamento seria aquele que se assemelhasse ao Merge Sort, onde o array é sempre dividido ao meio. Em outras palavras, se o pivô sempre estiver localizado no meio do array, teremos uma recursão em forma de árvore binária, na qual o ramo esquerdo tem metade do tamanho do array e o ramo direito também tem metade do tamanho. Essa árvore terá a altura mínima, ou seja:

$$O(lgn)$$

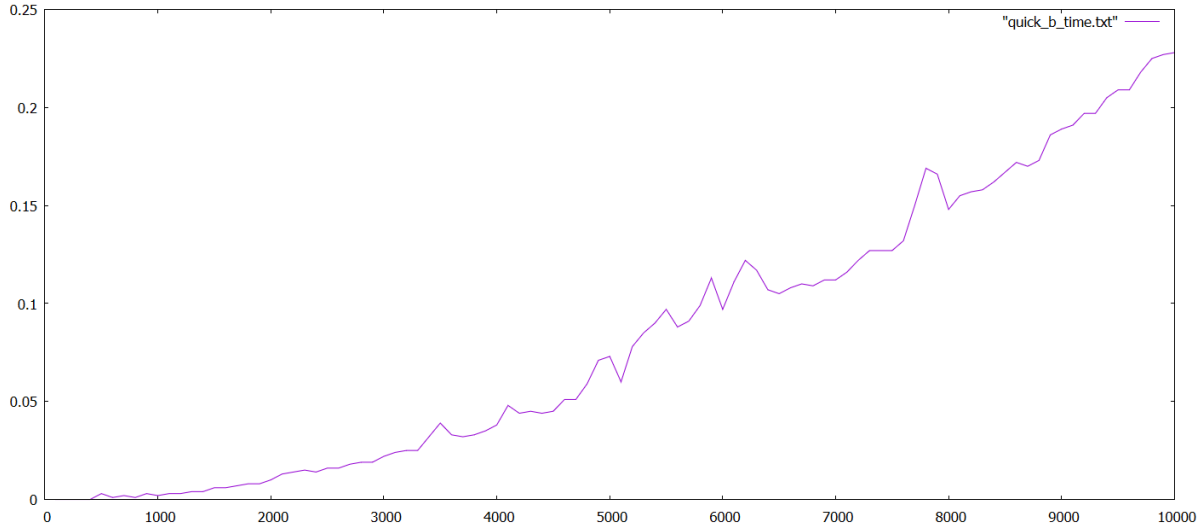
Como resultado, o melhor caso do Quick Sort é o custo do lgn execuções do particiona, apos os custo temos como resultado:

$$O(nlgn)$$

Resposta Assintótica:

O(n log n)

Grafico Quick Sort Melhor Caso:



2.8 Quick Sort Caso Médio

No caso médio do Quick Sort, a escolha do pivô e a divisão da lista levam a uma divisão razoavelmente equilibrada. Isso resulta em um desempenho eficiente do algoritmo.

O Quick Sort segue o paradigma "divide and conquer"(dividir e conquistar). Ele seleciona um pivô da lista e rearranja os elementos ao redor dele, dividindo a lista em duas partes. Essa operação é repetida de forma recursiva nas sub-listas até que a lista esteja ordenada.

No caso médio, espera-se que a divisão da lista seja equilibrada, com tamanhos aproximadamente iguais nas sub-listas. Isso permite que o Quick Sort tenha um desempenho médio eficiente, com uma complexidade de tempo média O(n log n), onde "n"é o número de elementos na lista.

Apesar do desempenho médio eficiente, é importante mencionar que o Quick Sort pode ter um desempenho pior no pior caso, quando a escolha do pivô resulta em divisões desequilibradas. Nesses casos, a complexidade de tempo pode chegar a $O(n^2)$. No entanto, existem técnicas para melhorar o desempenho, como a escolha do pivô mediano ou o uso do Quick Sort de três vias.

Resumindo, no caso médio do Quick Sort, a escolha do pivô e a divisão da lista são equilibradas, resultando em um desempenho eficiente com uma complexidade de tempo média O(n log n). O algoritmo é amplamente utilizado e eficiente para lidar com grandes conjuntos de dados. No entanto, é importante considerar os possíveis cenários de desempenho pior, especialmente quando há escolhas inadequadas de pivô.

Codigo Quick Sort Caso Médio:

```
1. int temp = *a;
2.  *a = *b;
3.  *b = temp;
4. int pivot = arr[high];
5.    int i = (low - 1);

6.    for (int j = low; j <= high - 1; j++) {
7.        if (arr[j] <= pivot) {
8.            i++;
9.            swap(&arr[i], &arr[j]);
```

```
10.         }
11.     }
12.     swap(&arr[i + 1], &arr[high]);
13.     return (i + 1);
```

Calculo Quick Sort Caso Médio:

$$T(n) = \begin{cases} O(n \log n), & \text{se } v_1 = v_2 \\ O(n^2), & \text{se } v_1 \neq v_2 \end{cases}$$

v_1, v_2 sendo as partes divididas do vetor

$$T(n) = T(|left|) + T(|right|) + \Theta(f(n))$$

$$T(n) = T(|left|) + T(|right|) + \Theta(n)$$

Para o caso médio de particionamento seria aquele também que se assemelhasse ao Merge Sort, onde o array também é sempre dividido ao meio. Em outras palavras o caso médio é igual ao melhor caso, tendo também quando o pivô estiver localizado no meio do array, teremos uma recursão em forma de árvore binária, na qual o ramo esquerdo tem metade do tamanho do array e o ramo direito também tem metade do tamanho. Essa árvore terá a altura mínima, ou seja:

$$O(lgn)$$

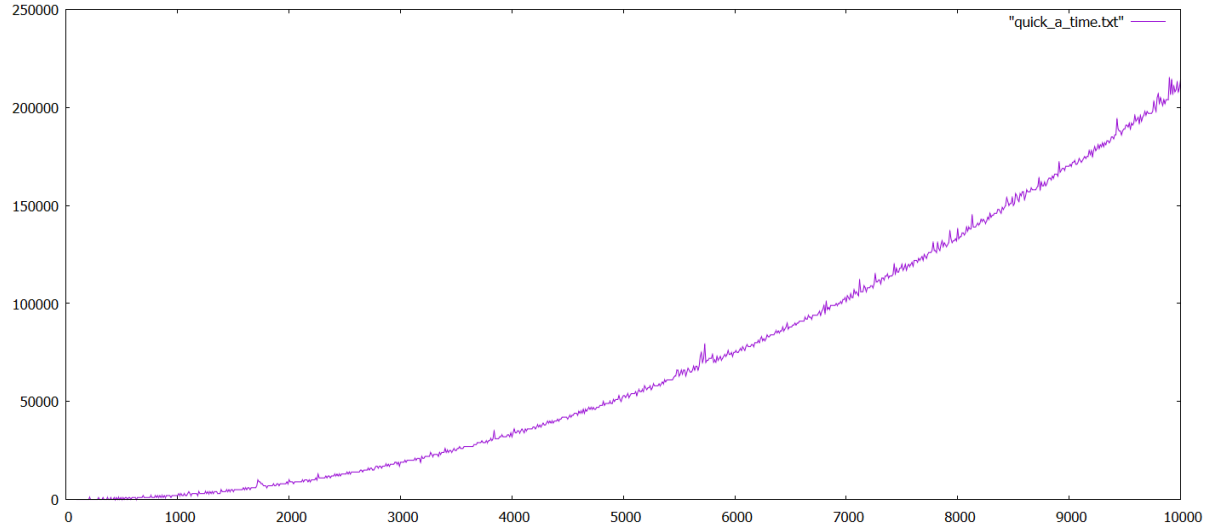
Como resultado, o caso médio do Quick Sort também é o custo do lgn execuções do particiona, apos os custo temos o mesmo resultado do melhor caso, que seria:

$$O(nlgn)$$

Resposta Assintótica:

O(n log n)

Grafico Quick Sort Caso Médio:



2.9 Quick Sort Pior Caso:

No pior caso do Quick Sort, ocorre um desequilíbrio nas divisões devido à escolha do pivô. Isso ocorre quando o pivô é o elemento mínimo ou máximo da lista, ou quando a lista está quase totalmente ordenada.

Quando o Quick Sort se depara com divisões desequilibradas, sua eficiência é significativamente afetada. No pior caso, a complexidade de tempo pode se tornar quadrática, com uma notação $O(n^2)$, onde "n"é o número de elementos na lista. Isso ocorre porque o algoritmo precisa realizar um grande número de comparações e trocas em cada nível da recursão.

No pior caso, o Quick Sort pode se tornar mais lento do que outros algoritmos de ordenação, como o Merge Sort, que garantem uma complexidade de tempo linear.

Para mitigar o pior caso, são aplicadas técnicas de otimização no Quick Sort, como a escolha inteligente do pivô ou o uso do Quick Sort de três vias. Essas abordagens visam garantir uma divisão mais equilibrada da lista, mesmo em casos em que a lista esteja parcialmente ordenada ou contenha elementos repetidos.

Resumindo, o pior caso do Quick Sort ocorre quando a escolha do pivô resulta em divisões desequilibradas, levando a uma complexidade de tempo quadrática $O(n^2)$. No entanto, técnicas de otimização podem ser aplicadas para melhorar o desempenho, mesmo em casos de dados parcialmente ordenados. O Quick Sort continua sendo amplamente utilizado em situações em que a distribuição dos dados é aleatória ou desconhecida, devido à sua eficiência média.

Codigo Quick Sort Pior Caso:

```
1.    int t = *a;
2.    *a = *b;
3.    *b = t;
4.    int pivot = arr[high];
5.    int i = (low - 1);

6.    for (int j = low; j <= high - 1; j++) {
7.        if (arr[j] < pivot) {
8.            i++;
9.            swap(&arr[i], &arr[j]);
10.        }
11.    }
12.    swap(&arr[i + 1], &arr[high]);
13.    return (i + 1);
```

Calculo Quick Sort Pior Caso:

$$T(n) = \begin{cases} O(n \log n), & \text{se } v_1 = v_2 \\ O(n^2), & \text{se } v_1 \neq v_2 \end{cases}$$

v_1, v_2 sendo as partes divididas do vetor

$$T(n) = T(|left|) + T(|right|) + \Theta(f(n))$$

$$T(n) = T(|left|) + T(|right|) + \Theta(n)$$

No pior caso o Quick Sort é $O(n2)$. Esse caso se manifesta quando o pivot sempre divide o array em duas porções de tamanho 0 e $n - 1$, respectivamente

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

ou seja

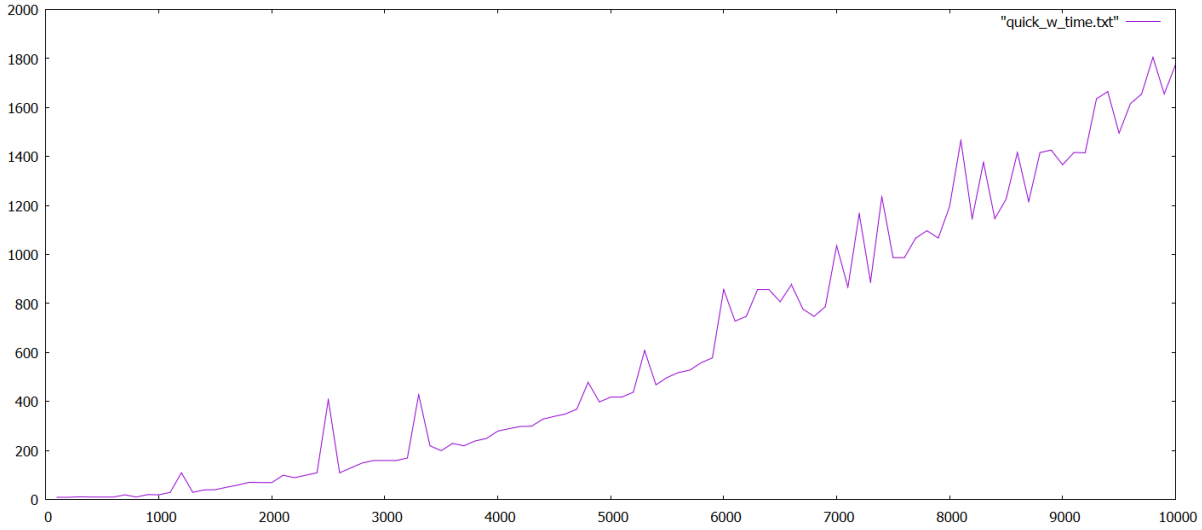
$$T(n) = T(n-1) + \Theta(n)$$

$$O(n^2)$$

resposta assintótica:

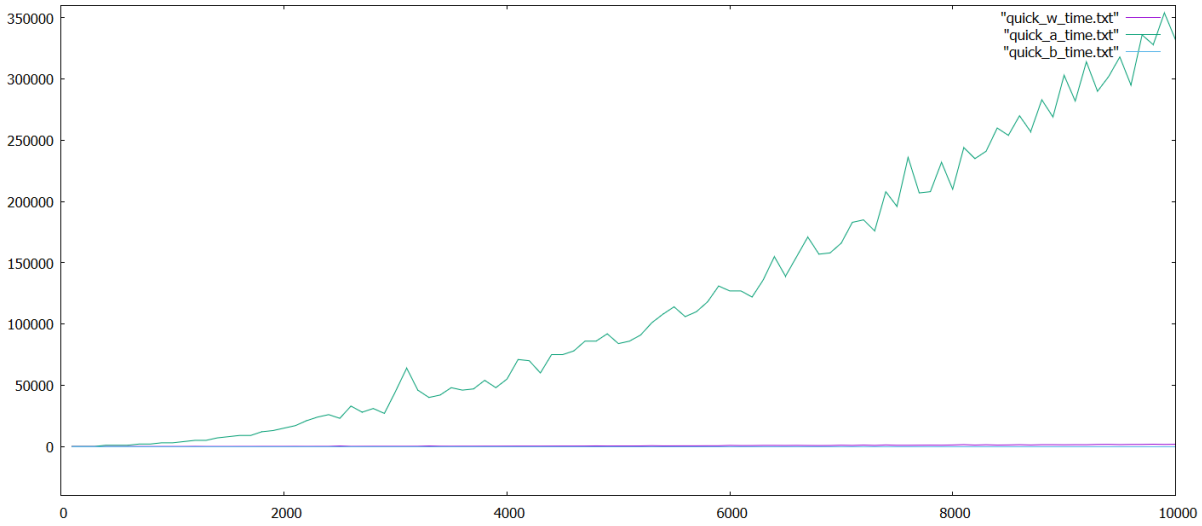
$O(n^2)$

Grafico Quick Sort Pior Caso:



2.10 Comparação Entre os Graficos Quick Sort

Grafico de Comparação.



Explicação dos Graficos Para Cada Caso.

Melhor caso: O melhor caso ocorre quando o Quicksort seleciona um pivô que divide o array em duas partes exatamente iguais em cada iteração. Isso resulta em partições equilibradas e leva a um desempenho ideal. Graficamente, o algoritmo irá dividir o array em duas metades aproximadamente iguais em cada passo, formando uma árvore de recursão balanceada. Como resultado, o tempo de execução no melhor caso é geralmente proporcional a O(n log n), onde "n"é o número de elementos a serem ordenados.

Pior caso: O pior caso ocorre quando o Quicksort seleciona um pivô que divide o array de maneira desequilibrada em cada iteração. Isso pode acontecer, por exemplo, quando o pivô escolhido é sempre o menor ou o maior elemento do array. Graficamente, o algoritmo irá dividir o array de forma desigual, com uma das partições tendo a maioria dos elementos e a outra com poucos elementos. Isso resulta em uma árvore de recursão desbalanceada, que pode se transformar em uma estrutura semelhante a uma lista encadeada. No pior caso, o tempo de execução é quadrático, $O(n^2)$, o que significa que o algoritmo pode se tornar muito lento quando o número de elementos a serem ordenados aumenta.

Caso médio: O caso médio é uma estimativa do desempenho do Quicksort com base em uma distribuição aleatória de elementos. Graficamente, o algoritmo dividirá o array em partições aproximadamente iguais na maioria dos casos, mas pode haver algumas variações ocasionais. O caso médio é geralmente muito mais próximo do melhor caso do que do pior caso em termos de tempo de execução. No caso médio, o tempo de execução

do Quicksort é O(n log n), que é uma das razões pelas quais o algoritmo é amplamente utilizado.

Em resumo, o Quicksort tem um melhor desempenho no caso médio e um desempenho pior no pior caso. O melhor caso é alcançado quando as partições são equilibradas em cada iteração. É importante levar em consideração essas diferentes situações ao analisar a eficiência do Quicksort e escolher o algoritmo de ordenação mais adequado para um determinado conjunto de dados.

2.11 Selection Sort

O selection sort é um método de ordenação que consiste em encontrar repetidamente o menor (ou maior) elemento de uma lista desordenada e movê-lo para a posição correta. Esse processo é repetido até que todos os elementos estejam ordenados.

Durante cada iteração do selection sort, o algoritmo identifica o menor elemento restante na lista e o coloca na sua posição correta. Esse processo é repetido até que todos os elementos estejam nas suas posições finais. O tempo de execução do algoritmo é de $O(n^2)$.

Embora seja fácil de entender e implementar, o selection sort não é eficiente para conjuntos de dados grandes, pois requer um grande número de comparações e trocas. Algoritmos mais avançados, como o merge sort e o quicksort, são geralmente preferidos para ordenar grandes volumes de dados. No entanto, o selection sort pode ser útil para listas pequenas ou em situações em que a simplicidade do código e o uso de memória são preocupações importantes.

Codigo Selection Sort:

```
1.    int i, j, minIndex, temp;
2.    for (i = 0; i < n - 1; i++) {
3.        minIndex = i;
4.        for (j = i + 1; j < n; j++) {
5.            if (arr[j] < arr[minIndex])
6.                minIndex = j;
7.        }
8.        if (minIndex != i) {
9.            temp = arr[i];
10.            arr[i] = arr[minIndex];
11.            arr[minIndex] = temp;
12.        }
13.    }
```

Calculo Selection Sort:

Começa buscando pelo menor elemento que custa $T(1) = n-1$, na primeira iteração,$T(2) = n - 2$ na segunda, $T(3) = n - 3$ na terceira.

Assim como no caso do Insertion Sort, o custo é dado por:

$$\sum_{1}^{\text{n-1}} T(n) = 1 + 2 + 3 + \ldots (n-1)$$

A soma dos termos de uma PA é:

$$T(n) = (a1 + an) \cdot \frac{n}{2}$$

Então, temos que o tempo de execução do algoritmo é dado por:

$$T(n) = (1 + (n-1)) \cdot \frac{n}{2} = \frac{n^2}{2}$$

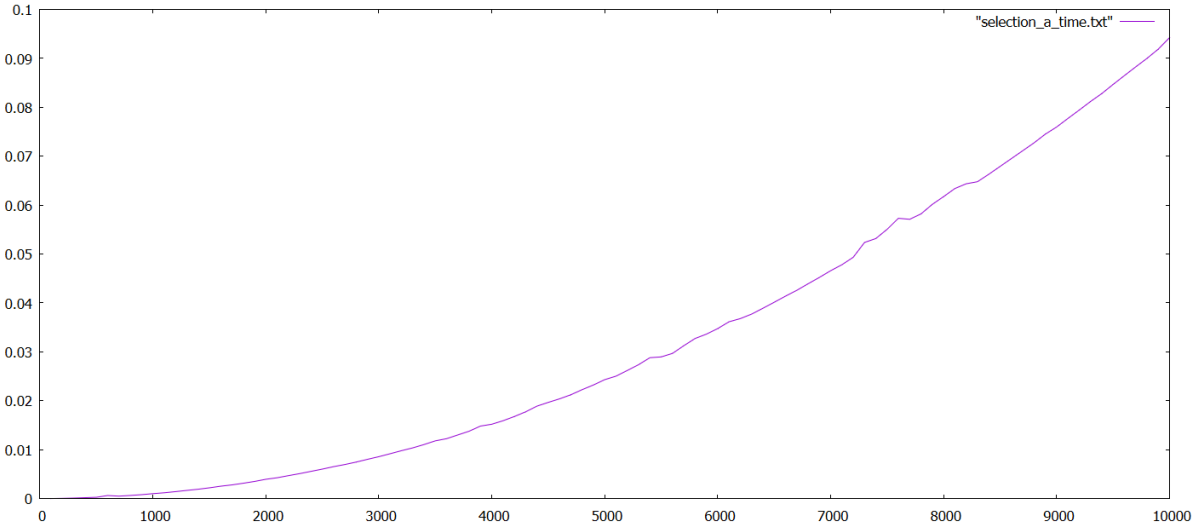
Aplicando as diretrizes de simplificação, o Selection Sort é:

$$T(n) = \Theta(n^2)$$

Resposta Assintótica:

$O(n^2)$

Gráfico selection sort:



3 Conclusão

Em conclusão, os algoritmos de ordenação são ferramentas essenciais no campo da ciência da computação, permitindo que os desenvolvedores organizem e classifiquem eficientemente conjuntos de dados. Neste texto, exploramos cinco algoritmos de ordenação populares: Insertion Sort, Quick Sort, Distribution Sort, Selection Sort e Merge Sort. Cada um desses algoritmos possui suas próprias características e desempenho em diferentes casos de uso.

O Insertion Sort é um algoritmo simples e intuitivo que insere cada elemento em sua posição correta em uma lista. Sua eficiência varia dependendo do caso: melhor, pior e médio. No melhor caso, é eficiente O(n), no pior caso é lento $O(n^2)$, e no caso médio também é $O(n^2)$. É útil para listas pequenas ou quase ordenadas, mas menos eficiente em grandes conjuntos de dados.

O Quick Sort é um algoritmo de ordenação eficiente que utiliza a estratégia "dividir para conquistar". Ele seleciona um elemento pivot, rearranja os elementos de forma que os menores fiquem antes do pivot e os maiores fiquem depois, e recursivamente aplica essa operação nos subarrays resultantes. O Quick Sort tem um desempenho médio muito bom que se assemelha bastante ao seu melhor desempenho, com uma complexidade média/melhor de O(n log n). No entanto, seu desempenho pior caso pode ser $O(n^2)$ se a escolha do pivot não for adequada.

O Distribution Sort, também conhecido como Bucket Sort, é um algoritmo de ordenação eficiente para conjuntos de dados com uma distribuição uniforme. Ele divide o intervalo dos valores de entrada em intervalos menores, chamados de "baldes", e distribui os elementos nos baldes apropriados. Em seguida, cada balde é ordenado individualmente e, por fim, os elementos são concatenados para obter a lista ordenada. O Distribution Sort tem uma complexidade média de O(n + k), onde n é o número de elementos e k é o número de baldes. No entanto, sua eficiência depende da distribuição dos dados e pode ser afetada se os elementos não forem uniformemente distribuídos.

O Selection Sort é um algoritmo simples que divide a lista de elementos em duas partes: a parte ordenada e a parte não ordenada. Ele encontra repetidamente o elemento mínimo da parte não ordenada e o coloca no final da parte ordenada. Embora seja fácil de implementar, o Selection Sort tem um desempenho relativamente baixo, com uma complexidade de $O(n^2)$ em todos os casos, independentemente da ordem dos elementos.

Por fim, o Merge Sort é um algoritmo de ordenação baseado na estratégia "dividir para conquistar". Ele divide a lista em subarrays menores, recursivamente ordena esses subarrays e, em seguida, mescla-os para obter a lista final ordenada. O Merge Sort tem um desempenho consistente e eficiente, com uma complexidade média e no pior caso de O(n log n). No entanto, requer um espaço adicional para a criação dos subarrays temporários, o que pode ser uma desvantagem em cenários com restrições de memória.

Em resumo, cada algoritmo de ordenação possui suas vantagens e desvantagens, e a escolha do algoritmo adequado depende do tamanho do conjunto de dados, da distribuição dos elementos e dos recursos disponíveis. Os desenvolvedores devem considerar esses fatores ao selecionar o algoritmo de ordenação mais adequado para cada caso específico, buscando um equilíbrio entre eficiência e simplicidade de implementação.

Referências

HOFFMAN, L. D.; BRADLEY, G. L. *Calculus for Business, Economics, and the Social and Life Sciences*. [S.l.]: McGraw-Hill, 1992.

MENDES, K. D. S.; SILVEIRA, R. C. d. C. P.; GALVÃO, C. M. Revisão integrativa: método de pesquisa para a incorporação de evidências na saúde e na enfermagem. *Texto & contexto enfermagem*, Universidade Federal de Santa Catarina, v. 17, n. 4, p. 758–764, 2008.