PONTIFÍCIA UNIVERSIDADE CATÓLICA

ENGENHARIA DE COMPUTAÇÃO

MAIZA LETICIA OLIVEIRA RA: 22900229

MARIANA ASSIS FERREIRA RA: 21943527

VITOR YUZO TAKEI RA: 22023740

ANÁLISE EMPÍRICA DE ALGORÍTIMOS DE ORDENAÇÃO

Projeto 2

CAMPINAS

2023

MAIZA LETICIA OLIVEIRA RA: 22900229

MARIANA ASSIS FERREIRA RA: 21943527

VITOR YUZO TAKEI RA: 22023740

ANÁLISE EMPÍRICA DE ALGORÍTIMOS DE ORDENAÇÃO

Projeto 2

Trabalho apresentado no curso de graduação Engenharia da Computação da Pontifícia Universidade Católica. Sob orientação da professora Lucia Filomena De Almeida Guimaraes.

CAMPINAS

2023

Sumário

Introdução

A ordenação de dados é uma operação essencial na computação, impactando diretamente o desempenho de algoritmos e sistemas computacionais. Este estudo visa comparar e analisar o desempenho de alguns métodos de ordenação clássicos.

A ideia do projeto é analisar e ordenar, de maneira decrescente, vetores de diferentes tamanhos:

- 10 000
- 50 000
- 100 000
- 500 000
- 1 000 000

Sendo que cada elemento será constituído de dois campos: a chave (tipo inteiro) e o real (tipo float). A referência de ordenação é a chave.

A avaliação será feita através da análise do tempo (em segundos) de execução de cada método.

Cada método deverá analisar:

- 10 casos de números gerados aleatoriamente (forma 1)
- 10 casos de números crescentes gerados aleatoriamente (forma 2)

Ao final, busca-se extrair conclusões sobre a eficiência de cada algoritmo, e entender qual foi o pior, médio e melhor caso analisado, para cada método. Este trabalho contribuirá para a compreensão aprimorada das complexidades na seleção de métodos de ordenação.

Instruções para o uso do programa

O algoritmo está organizado para demonstrar que todos os métodos estão funcionando e ordenando corretamente e como determinado. No entanto, recomenda-se que, para uma maior eficiência e desempenho, o usuário selecione no “switch case” apenas o método desejado e “comente” o restante com o uso do comando: /* */. Segue um exemplo:

```
//do
//{
    //printf ("\n\nOpcao: ");
    //opcao = 0;
    //scanf  ("%d",&opcao);
//}while(opcao<1 || opcao>1);

//switch(opcao)
//{
    //case 1:
        /*
        start_time = 0;
        end_time = 0;
        elapsed_time = 0;


        printf ("\n--------------- STATUS ---------------");
        start_time = clock();
        quicksort(vetor,0,tamanho-1);
        end_time = clock();
        elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

        printf("\nTEMPO DE EXECUCAO EM MILISEGUNDOS:%0.2f", elapsed_time*1000);
        printf("\nTEMPO DE EXECUCAO EM SEGUNDOS:%f \n-------------------------------------\n\n", elapsed_time);

        */
    //break;
```

```
    //case 2:

        start_time = 0;
        end_time = 0;
        elapsed_time = 0;


        printf ("\n--------------- STATUS ---------------");
        start_time = clock();
        insertionSort(vetor,tamanho);
        end_time = clock();
        elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

        printf("\nTEMPO DE EXECUCAO EM MILISEGUNDOS:%0.2f", elapsed_time*1000);
        printf("\nTEMPO DE EXECUCAO EM SEGUNDOS:%f \n-------------------------------------\n\n", elapsed_time);


    //break;
```

Vale ressaltar que os tempos vão variar de acordo com o computador. Todos os testes arquivados nesse relatório foram realizados em um Notebook com as seguintes especificações:

- Acer
- Ryzen 7 – 4000 series
- 16 gb de memória RAM
- GTX 1650 4gb

Além disso, as sementes (seed) utilizadas para a geração dos números aleatórios foram: 10, 20, 30, 40, 50, 60, 70, 80, 90 e 100. Porém, isso já está registrado no programa e não precisa de intervenção do usuário.

Apresentação dos Métodos de Ordenação e Análise dos Resultados

1. InsertionSort

1.1. Algoritmo com as devidas alterações

Função InsertionSortDecrescente(vetor):

Para i de 1 até tamanho - 1:

valorAtual <- vetor[i]

j <- i - 1

Enquanto j >= 0 E vetor[j] < valorAtual:

vetor[j + 1] <- vetor[j]

j <- j - 1

vetor[j + 1] <- valorAtual

O algoritmo percorre o vetor, compara cada elemento com os elementos à sua esquerda e os move para a direita até encontrar a posição correta para o elemento atual. Isso resulta no vetor final ordenado em ordem decrescente. O processo é realizado através de dois loops, um externo que percorre todo o vetor, e um interno que compara e move os elementos à esquerda. O vetor ordenado é então retornado pela função.

1.2. Apresentação dos dados e situações

InsertionSort	
Melhor Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,064
$\mathbf{5 * 10^4}$	1,71
$\mathbf{10^5}$	6,799
$\mathbf{5 * 10^5}$	168,309
$\mathbf{10^6}$	676,577

InsertionSort	
Caso Médio **Geração Aleatória – Forma 1**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,064
$\mathbf{5 * 10^4}$	1,724
$\mathbf{10^5}$	6,796
$\mathbf{5 * 10^5}$	168,794
$\mathbf{10^6}$	680,259

InsertionSort	
Pior Caso **Geração Aleatória – Forma 1**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,066
$\mathbf{5 * 10^4}$	1,705
$\mathbf{10^5}$	6,809
$\mathbf{5 * 10^5}$	168,555
$\mathbf{10^6}$	718,088

InsertionSort	
Melhor Caso **Ordem Inversa (Crescente -> Decrescente) – Forma 2**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,14
$\mathbf{5 * 10^4}$	3,44
$\mathbf{10^5}$	13,808
$\mathbf{5 * 10^5}$	341,322
$\mathbf{10^6}$	1364,821

InsertionSort	
Caso Médio **Ordem Inversa (Crescente -> Decrescente) – Forma 2**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,134
$\mathbf{5 * 10^4}$	3,423
$\mathbf{10^5}$	13,865
$\mathbf{5 * 10^5}$	340,573
$\mathbf{10^6}$	1369,683

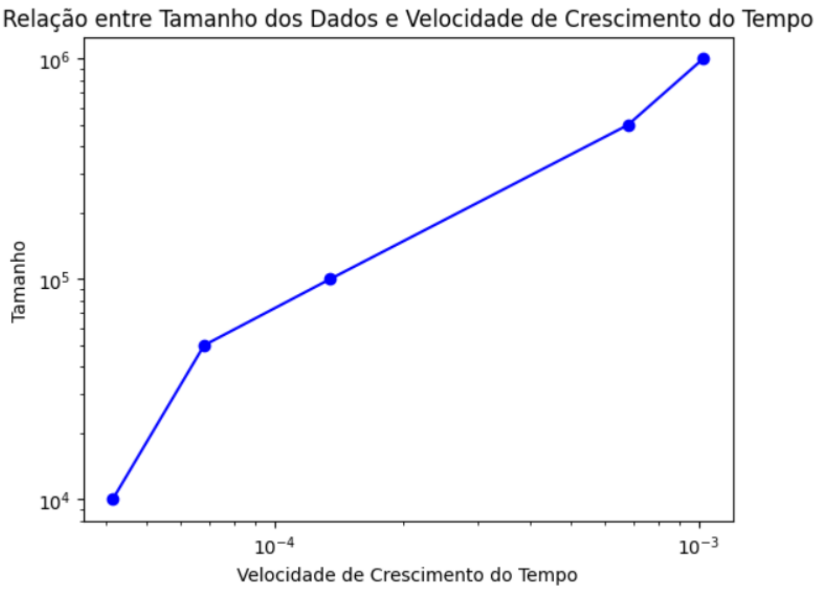
InsertionSort	
Pior Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,14
$\mathbf{5 * 10^4}$	3,435
$\mathbf{10^5}$	13,886
$\mathbf{5 * 10^5}$	343,395
$\mathbf{10^6}$	2269,823

1.3. Considerações

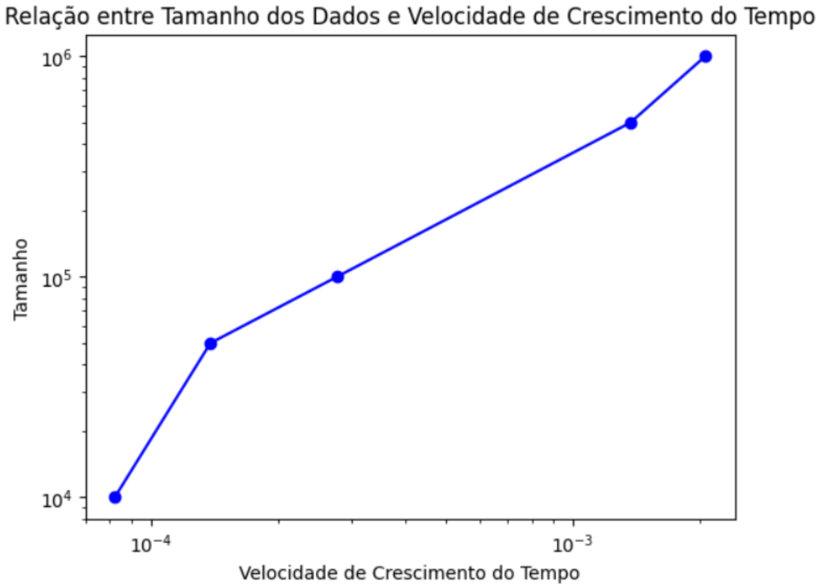
Segundo a análise das tabelas no tópico anterior, observa-se que o método InsertionSort possui um melhor desempenho na forma 1 e um pior desempenho na forma 2.

O tempo médio registrado na forma 1 foi de 171,53 segundos. E o tempo médio registrado na forma 2 foi de 345,54 segundos. Pode-se considerar uma diferença relativamente alta. Seguem os gráficos que relacionam Velocidade de Crescimento do Tempo x Tamanho:

Caso médio da forma 1:



Caso médio da forma 2:



2. BubleSort

2.1. Algoritmo com as devidas alterações

Função BubbleSortDecrescente(vetor):

Para i de 0 até tamanho - 1:

Para j de 0 até tamanho - i - 1:

Se vetor[j] < vetor[j + 1]:

Troque vetor[j] e vetor[j + 1]

O algoritmo utiliza dois loops, um externo para percorrer o vetor e garantir que o maior elemento seja movido para a última posição após cada passagem, e um interno para comparar e trocar elementos adjacentes que estão fora de ordem. A troca é realizada se o elemento atual for menor que o próximo elemento. O processo é repetido em passagens sucessivas até que todo o vetor esteja ordenado em ordem decrescente, e o vetor ordenado é então retornado pela função.

2.2. Apresentação dos dados e situações

BubleSort Melhor Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0,27
$5 * 10^4$	5,973
10^5	24,195
$5 * 10^5$	561,597
10^6	2245,019

BubleSort Caso Médio Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0,285
$5 * 10^4$	5,76
10^5	24,04
$5 * 10^5$	562,603
10^6	2252,372

BubleSort Pior Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0,293
$5 * 10^4$	5,885
10^5	24,409
$5 * 10^5$	599,691
10^6	4496,973

BubleSort	
Melhor Caso **Ordem Inversa (Crescente -> Decrescente) – Forma 2**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,305
$\mathbf{5 * 10^4}$	5,444
$\mathbf{10^5}$	19,563
$\mathbf{5 * 10^5}$	439,208
$\mathbf{10^6}$	1749,25

BubleSort	
Caso Médio **Ordem Inversa (Crescente -> Decrescente) – Forma 2**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,304
$\mathbf{5 * 10^4}$	4,76
$\mathbf{10^5}$	18,979
$\mathbf{5 * 10^5}$	440,728
$\mathbf{10^6}$	1752,794

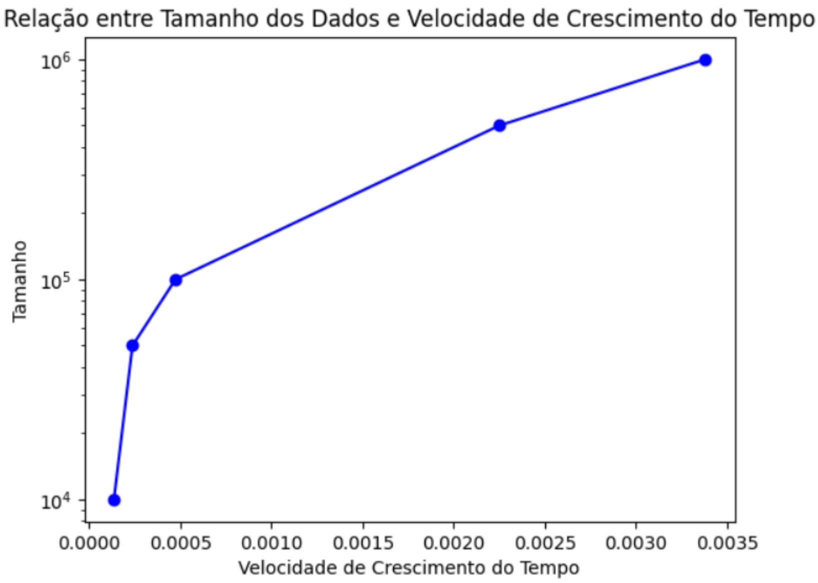
BubleSort	
Pior Caso **Ordem Inversa (Crescente -> Decrescente) – Forma 2**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,31
$\mathbf{5 * 10^4}$	4,744
$\mathbf{10^5}$	18,691
$\mathbf{5 * 10^5}$	1993,415
$\mathbf{10^6}$	1751,506

2.3. Considerações

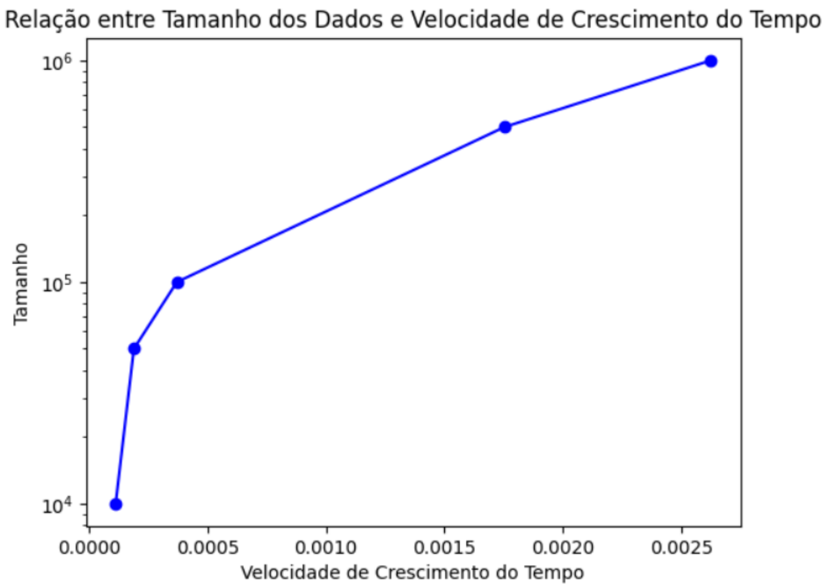
Segundo a análise das tabelas no tópico anterior, observa-se que o método BubleSort possui um melhor desempenho na forma 2 e um pior desempenho na forma 1.

O tempo médio registrado na forma 1 foi de 569,01 segundos. E o tempo médio registrado na forma 2 foi de 443,513 segundos. Pode-se considerar uma diferença relativamente baixa. Seguem os gráficos:

Caso médio da forma 1:



Caso médio da forma 2:



3. ShellSort

3.1. Algoritmo com as devidas alterações

Enquanto (j >= h) e (vetor[j - h] < vetor[j]) faça

trocar vetor[j] e vetor[j - h]

j <- j - h

fim enquanto

O algoritmo ShellSort utiliza dois loops, um loop externo para percorrer o vetor e mover o maior elemento para a primeira posição em cada passagem, e um loop interno para comparar e trocar elementos adjacentes que estão fora de ordem.

Para ordenar o vetor de forma decrescente basta criar uma condição (dentro do loop interno) para que sejam realizadas apenas se o elemento atual for menor que o próximo, e esse processo é repetido até que todo o vetor esteja ordenado em ordem decrescente.

3.2. Apresentação dos dados e situações

ShellSort	
Melhor Caso **Geração Aleatória – Forma 1**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,094
$\mathbf{5 * 10^4}$	2,17
$\mathbf{10^5}$	5,957
$\mathbf{5 * 10^5}$	142,493
$\mathbf{10^6}$	574,156

ShellSort Caso Médio Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,110
$\mathbf{5 * 10^4}$	1,480
$\mathbf{10^5}$	5,689
$\mathbf{5 * 10^5}$	143,136
$\mathbf{10^6}$	576,666

ShellSort Pior Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,094
$\mathbf{5 * 10^4}$	2,582
$\mathbf{10^5}$	9,849
$\mathbf{5 * 10^5}$	143,405
$\mathbf{10^6}$	573,648

ShellSort Melhor Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,188
$\mathbf{5 * 10^4}$	3,031
$\mathbf{10^5}$	11,613
$\mathbf{5 * 10^5}$	289,605
$\mathbf{10^6}$	1156,283

ShellSort Caso Médio Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,203
$\mathbf{5 * 10^4}$	2,922
$\mathbf{10^5}$	11,84
$\mathbf{5 * 10^5}$	291,078
$\mathbf{10^6}$	1158,678

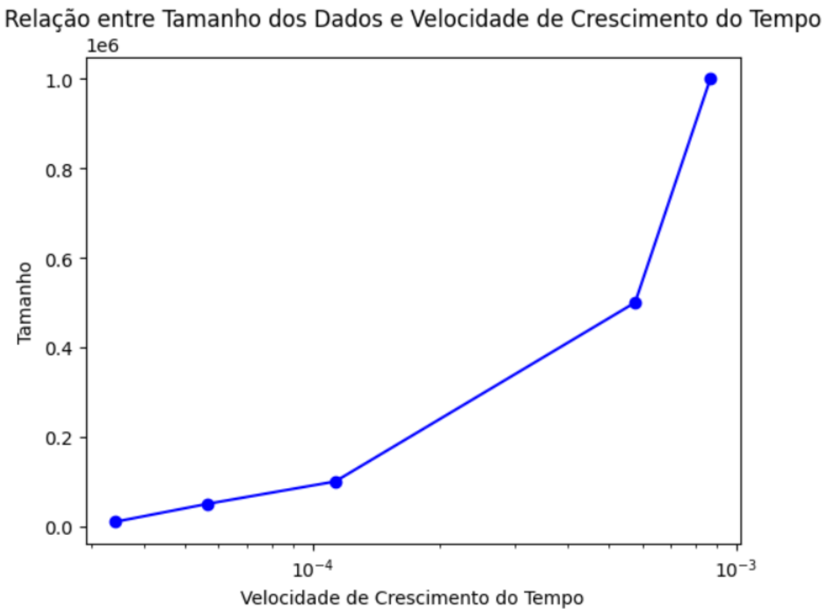
ShellSort	
Pior Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,218
$\mathbf{5 * 10^4}$	3,516
$\mathbf{10^5}$	11,734
$\mathbf{5 * 10^5}$	287,839
$\mathbf{10^6}$	1163,871

3.3. Considerações

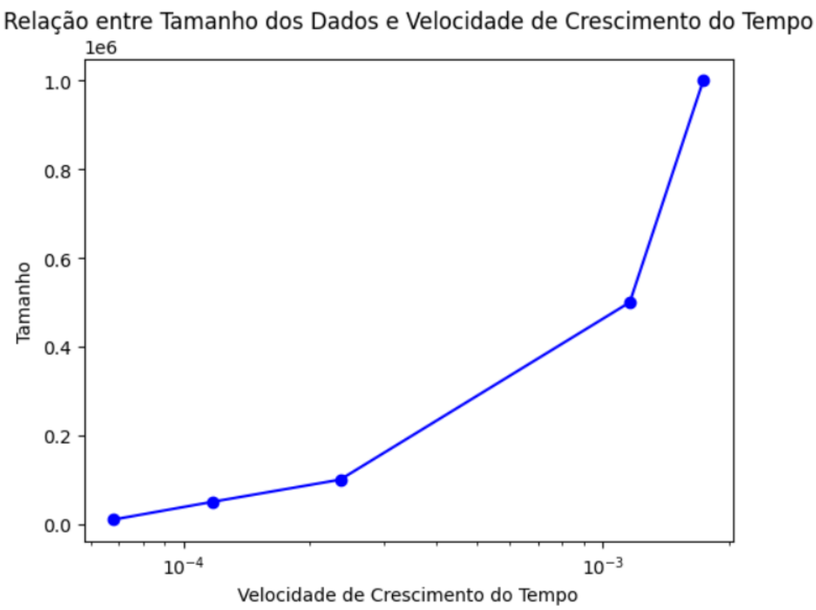
Segundo a análise das tabelas no tópico anterior, observa-se que o método ShellSort possui um melhor desempenho na forma 1 e um pior desempenho na forma 2.

O tempo médio registrado na forma 1 foi de 145,12 segundos. E o tempo médio registrado na forma 2 foi de 292,94 segundos. Pode-se considerar uma diferença relativamente baixa comparado com os outros métodos. Seguem os gráficos:

Caso médio da forma 1:



Caso médio da forma 2:



4. MergeSort

4.1. Algoritmo com as devidas alterações

Procedimento merge_sort(vetor,inicio,fim,tamanho)

se inicio < fim então

meio = (ini + fim) / 2

merge_sort(vetor, inicio, meio, tamanho)

merge_sort(vetor, meio + 1, fim, tamanho)

merge(vetor, inicio, meio, fim, tamanho)

fim se

Fim Procedimento

Função merge(vetor,inicio,meio,fim,tamanho)

i = inicio

j = meio + 1

k = inicio

aux = novo vetor de estrutura_vetor com tamanho = 'tamanho'

```
    enquanto i <= meio e j <= fim faça
        se elemento em i > elemento em j então
            elemento auxiliar em k = elemento em i
            incrementa i
        senão
            elemento auxiliar em k = elemento em j
            incrementa j
        fim se
        incrementa k
    fim enquanto

    enquanto i <= meio faça
        elemento auxiliar em k = elemento em i
        incrementa i
        incrementa k
    fim enquanto

    enquanto j <= fim faça
        elemento auxiliar em k = elemento em j
        incrementa j
        incrementa k
    fim enquanto

    para i de inicio até fim faça
        elemento em i = elemento auxiliar em i
    fim para

    liberar memória do vetor aux
Fim Função
```

O algoritmo MergeSort divide os elementos dos vetores até ter um vetor unitário de forma recursiva. Em seguida, é feita a junção e comparação dos vetores. A mudança fundamental na ordenação de forma descrente ocorre na comparação. Se o elemento no índice ‘i’ for maior que o elemento no índice ‘j’, ele é colocado primeiro na parte ordenada, resultando na ordenação decrescente do array.

4.2. Apresentação dos dados e situações

MergeSort	
Melhor Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0,015
$\mathbf{5 * 10^4}$	0,000
$\mathbf{10^5}$	0,032
$\mathbf{5 * 10^5}$	4,658
$\mathbf{10^6}$	9,635

MergeSort	
Caso Médio Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0
$\mathbf{10^5}$	0,015
$\mathbf{5 * 10^5}$	4,639
$\mathbf{10^6}$	9,714

MergeSort	
Pior Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0
$\mathbf{10^5}$	0,031
$\mathbf{5 * 10^5}$	4,686
$\mathbf{10^6}$	9,701

MergeSort	
Melhor Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0
$\mathbf{10^5}$	0,016
$\mathbf{5 * 10^5}$	4,659
$\mathbf{10^6}$	9,544

MergeSort	
Caso Médio Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0,015
$\mathbf{10^5}$	0,016
$\mathbf{5 * 10^5}$	4,63
$\mathbf{10^6}$	9,644

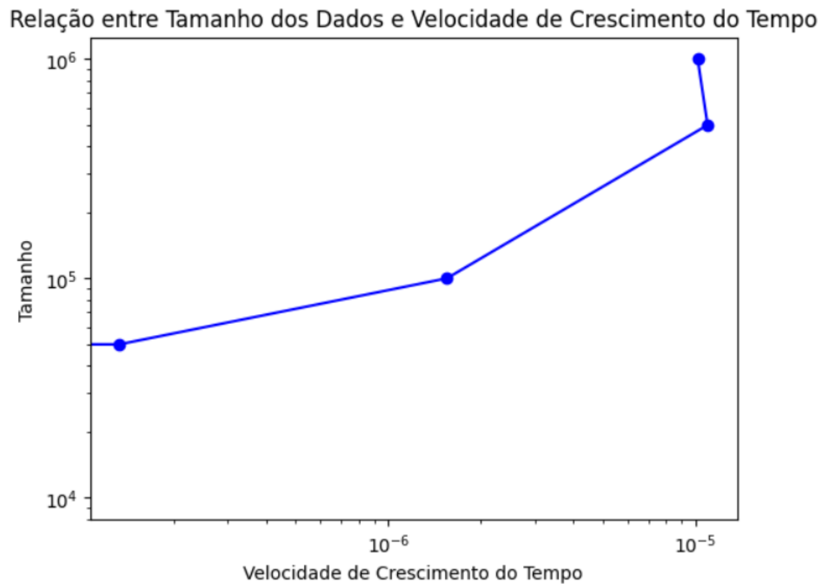
MergeSort	
Pior Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0,016
$\mathbf{10^5}$	0,016
$\mathbf{5 * 10^5}$	4,661
$\mathbf{10^6}$	9,675

4.3. Considerações

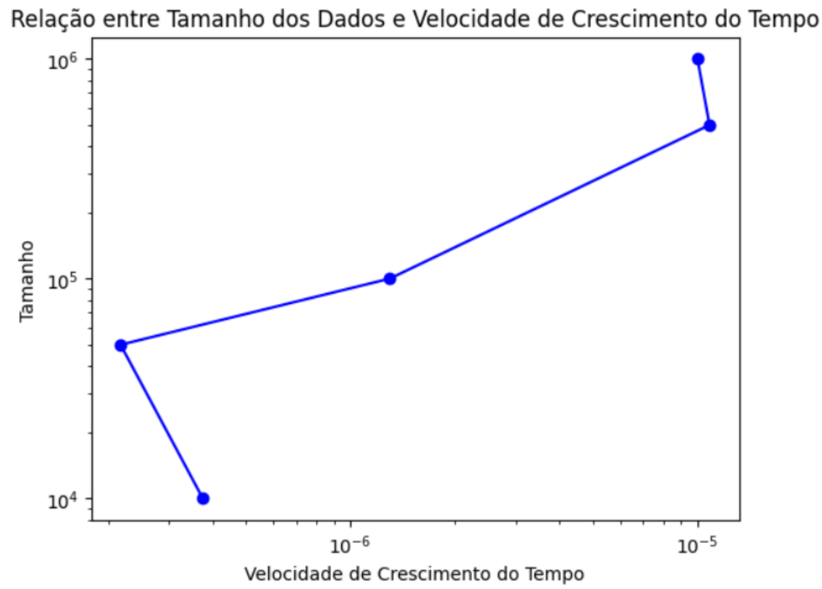
Segundo a análise das tabelas no tópico anterior, observa-se que o método MergeSort possui um desempenho aproximadamente igual em ambas as formas. Além disso, mostrou ser um método de ordenação muito mais rápido quando comparada com os anteriores.

O tempo médio registrado na forma 1 foi de 2,87 segundos. E o tempo médio registrado na forma 2 foi de 2,86 segundos. Pode-se considerar uma diferença extremamente baixa em relação aos outros métodos. Seguem os gráficos:

Caso médio da forma 1:



Caso médio da forma 2:



5. QuickSort

5.1. Com pivô no início: Algoritmo com as devidas alterações

Seguem as devidas alterações em Negrito:

QuickSort_recursiva(vetor, limite inferior, limite superior)

Se o limite inferior < limite superior

//Chama a função de partição
novo limite(p) = função que divide o vetor em partes (partição)

//Realiza a chamada recursiva com novos limites
QuickSort_recursiva(vetor, limite inferior, p - 1)
QuickSort_recursiva(vetor, p + 1, limite superior)

Fim da Condição

Fim

Partição(vetor, limite superior, limite inferior)

Pivô = elemento do limite inferior

Esquerda = posição do limite inferior

Direita = posição do limite superior

Enquanto esquerda < direita

Enquanto elemento da esquerda >= pivô

Incremento a esquerda

Enquanto elemento da direita < pivô

Decrementa a direita

Se esquerda < direita

Troca elemento da esquerda com a direita

Fim Enquanto

Troca elemento do limite inferior com o elemento da direita

Retorna o endereço do elemento da direita

Fim

A função particão é responsável por realizar a partição do vetor com base no pivô, que é o primeiro elemento do vetor. Inicialmente, são definidos dois índices, “e” e “d”, que apontam para o início e o final do subvetor a ser ordenado. O pivô é escolhido como o elemento na posição LI (limite inferior). Em seguida, a função entra em um loop while, onde e é incrementado enquanto o elemento na posição e é maior ou igual ao pivô, e d é decrementado enquanto o elemento na posição d é menor que o pivô.

Dentro desse loop, se “e” é menor que “d”, ocorre a troca dos elementos nas posições “e” e “d”. Esse processo é repetido até que “e” seja maior ou igual a “d”, indicando que a partição está completa. Após sair desse loop, realiza-se a troca do pivô (v[LI]) com o elemento na posição “d”, colocando o pivô em sua posição correta. A função então retorna o índice “d”, que será utilizado como ponto de divisão para as chamadas recursivas.

A função QuickSort_recursiva é a função principal do QuickSort. Ela inicia verificando se o índice do limite inferior (LI) é menor que o índice do limite superior (LS), garantindo que existam mais de dois elementos no subvetor a ser ordenado. Se essa condição for satisfeita, a função chama particão para obter o índice do pivô após a partição. Em seguida, faz chamadas recursivas para QuickSort_recursiva nos subvetores à esquerda (limite inferior até p-1) e à direita (p+1 até o limite superior), excluindo o próprio pivô que já está na posição correta.

5.2. Com pivô no Fim: Algoritmo com as devidas alterações

Seguem as devidas alterações em Negrito:

QuickSort_recursiva(vetor, limite inferior, limite superior)

Se o limite inferior < limite superior

//Chama a função de partição
novo limite(p) = função que divide o vetor em partes (partição)

//Realiza a chamada recursiva com novos limites
QuickSort_recursiva(vetor, limite inferior, p - 1)
QuickSort_recursiva(vetor, p + 1, limite superior)

Fim da Condição

Fim

Partição(vetor, limite superior, limite inferior)

Pivô = elemento do limite inferior

Esquerda = posição do limite inferior

Direita = posição do limite superior

Enquanto esquerda < direita

Enquanto elemento da esquerda > pivô

Incremento a esquerda

Enquanto elemento da direita <= pivô

Decrementa a direita

Se esquerda < direita

Troca elemento da esquerda com a direita

Fim Enquanto

Troca elemento do limite superior com o elemento da esquerda

Retorna o endereço do elemento da esquerda

Fim

A função partição desempenha um papel crucial no processo. Inicialmente, dois índices, ‘e’ e ‘d’, são definidos para apontar para o início (LI) e o final (LS) do subvetor a ser ordenado. O pivô é escolhido como o último elemento do vetor (v[LS]). Em seguida, um loop while é empregado para percorrer o subvetor, com e sendo incrementado enquanto o elemento na posição ‘e’ é maior que o pivô e ‘e’ é menor que o limite superior (LS). Simultaneamente, d é decrementado enquanto o elemento na posição d é menor ou igual ao pivô e d é maior que o limite inferior (LI). Se ‘e’ é menor que ‘d’, ocorre a troca dos elementos nas posições ‘e’ e ‘d’. Este processo continua até que ‘e’ seja maior ou igual a d. Após sair do loop, realiza-se a troca do pivô (v[LS]) com o elemento na posição e, assegurando que o pivô está na sua posição correta.

A função *QuickSort_recursiva* é responsável pela recursividade do algoritmo. Inicia-se verificando se o índice do limite inferior (LI) é menor que o índice do limite superior (LS). Se essa condição for atendida, a função chama partição para obter o índice do pivô após a partição. Em seguida, são realizadas chamadas recursivas para *QuickSort_recursiva* nos subvetores à esquerda (LI até p-1) e à direita (p+1 até LS). Importante notar que o próprio pivô já está na posição correta, então não é incluído nas chamadas recursivas.

5.3. Com pivô no Meio: Algoritmo com as devidas alterações

Seguem as devidas alterações em Negrito:

QuickSort_recursiva(vetor, limite inferior, limite superior)

Pivo = endereço do (lim. Inferior + lim. Superior) / 2

Esquerda = posição do limite inferior

Direita = posição do limite superior

Faça enquanto esquerda <= direita

Enquanto elemento da esquerda > pivo

Incrementa esquerda

Enquanto elemento da direita < pivo

Decrementa Direita

Se esquerda <= direita

Troca elemento da direita com elemento da esquerda

Incrementa a esquerda

Decrementa a direita

Fim Enquanto

Se limite inferior < direita

QuickSort_recursiva(vetor, limite inferior, direita)

Se esquerda < limite superior

QuickSort_recursiva(vetor, esquerda, limite superior)

Fim

A função *QuickSort_recursiva* inicia determinando dois índices, ‘e’ e ‘d’, que apontam para o início (LI) e o final (LS) do subvetor a ser ordenado, respectivamente. O pivô é selecionado como o elemento central do vetor, calculado através da expressão (LI + LS) / 2. Em seguida, é iniciado um loop do-while que continuará enquanto e for menor ou igual a d.

Dentro desse loop, e é incrementado enquanto o elemento na posição ‘e’ é maior que o pivô, e ‘d’ é decrementado enquanto o elemento na posição ‘d’ é menor que o pivô. Se ‘e’ é menor ou igual a ‘d’, ocorre a troca dos elementos nas posições ‘e’ e ‘d’, e ambos são movidos para suas respectivas direções. Esse processo se repete até que ‘e’ seja maior que ‘d’.

Após o término do loop do-while, são realizadas chamadas recursivas da própria função *QuickSort_recursiva* para os subvetores à esquerda (LI até d) e à direita (e até LS). Isso assegura que o vetor seja ordenado de forma completa. Importante notar que as verificações if(LI < d) e if(e < LS) garantem que há elementos à esquerda e à direita do pivô, respectivamente.

5.4. Apresentação dos dados e situações

Dentre as 3 modalidades de posições do pivô apresentadas anteriormente, tanto a modalidade do pivô no inicio quanto a modalidade do pivô no fim, apresentaram o estouro de memória Stack (StackOverflow) para vetores a partir do tamanho 50 000, na forma 2 (Ordem Inversa Crescente -> Decrescente). Sendo assim, considera-se o pivô no meio como a melhor modalidade do método QuickSort.

Portanto, seguem as tabelas:

QuickSort com Pivô no Meio	
Melhor Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0
$5 * 10^4$	0,016
10^5	0
$5 * 10^5$	0,047
10^6	0,093

QuickSort com Pivô no Meio	
Caso Médio Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0
$5 * 10^4$	0,017
10^5	0
$5 * 10^5$	0,047
10^6	0,118

QuickSort com Pivô no Meio	
Pior Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0
$5 * 10^4$	0,016
10^5	0
$5 * 10^5$	0,047
10^6	0,125

QuickSort com Pivô no Meio	
Melhor Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
10^4	0
$5 * 10^4$	0
10^5	0
$5 * 10^5$	0,016
10^6	0,044

QuickSort com Pivô no Meio	
Caso Médio Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0,001
$\mathbf{10^5}$	0
$\mathbf{5 * 10^5}$	0,031
$\mathbf{10^6}$	0,046

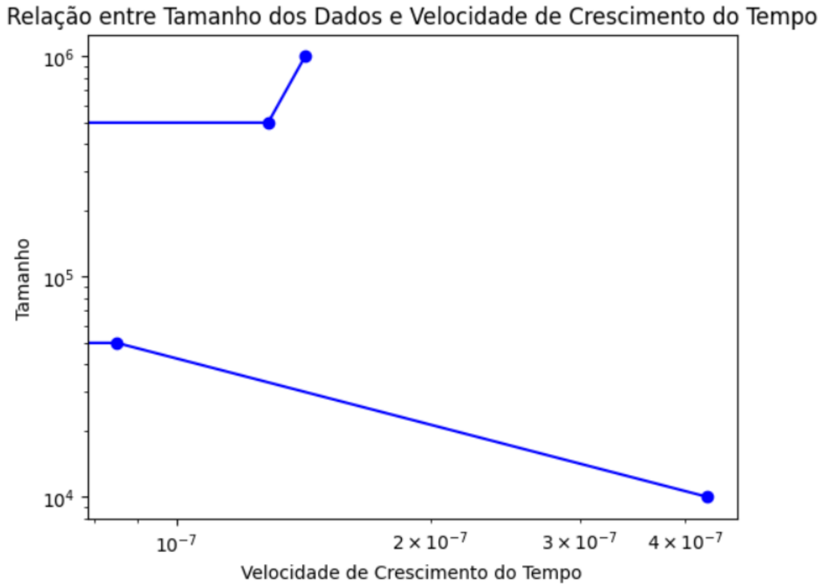
QuickSort com Pivô no Meio	
Pior Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0,001
$\mathbf{10^5}$	0
$\mathbf{5 * 10^5}$	0,016
$\mathbf{10^6}$	0,047

5.5. Considerações

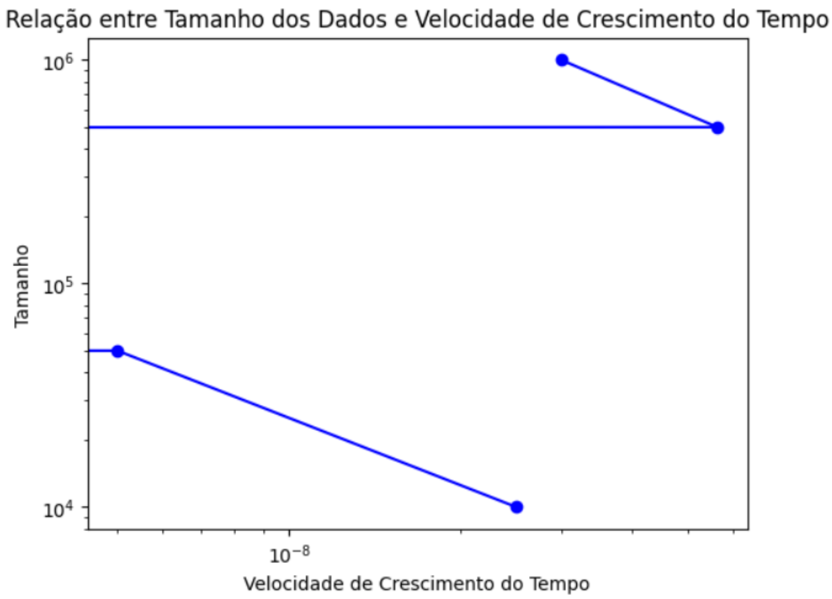
Segundo a análise das tabelas no tópico anterior, observa-se que o método QuickSort possui um desempenho aproximadamente igual em ambas as formas. Além disso, mostrou ser o método de ordenação mais rápido quando comparado com os anteriores.

O tempo médio registrado na forma 1 foi de 0,04 segundos. E o tempo médio registrado na forma 2 foi de 0,02 segundos. Pode-se considerar uma diferença extremamente baixa em relação aos outros métodos. Seguem os gráficos:

Caso médio da forma 1:



Caso médio da forma 2:



6. CombSort

6.1. Algoritmo com as devidas alterações

```
CombSort(vetor,tamanho)
    Trocado = 1 // verificador
    Intervalo = tamanho
        Enquanto intervalo > 1 ou trocado == 1
            Intervalo = intervalo * 10 / 13
            Se o intervalo == 9 ou intervalo == 10
                Intervalo = 11
            Se intervalo > 1
                Intervalo = 1
            Trocado = 0
            Esquerda  = 0
            Direita = intervalo
            Enquanto direita < intervalo
                Se elemento da esquerda < elemento da direita
                    Troca elemento da direita com esquerda
                    Trocado = 1 // Verificador
            Fim enquanto
        Fim enquanto
Fim
```

No início, a função estabelece um intervalo inicial igual ao tamanho do vetor. Em seguida, entra em um loop principal que se repete enquanto o intervalo for maior que 1 ou se houveram trocas no último percurso pelo vetor.

Dentro desse loop, o intervalo é ajustado de acordo com uma fórmula específica. Se o intervalo calcular um valor entre 9 e 10 (inclusive), ele é corrigido para 11. Caso o intervalo resulte em um valor menor que 1, ele é fixado como 1.

Seguindo, um percurso pelo vetor é realizado, comparando elementos a uma distância específica igual ao intervalo. Se um elemento na posição e for menor que o elemento na posição d (onde d = e + intervalo), ocorre a troca desses elementos, e a variável trocado é marcada como 1 para indicar que houve uma troca.

Esse processo é repetido até que o intervalo seja menor ou igual a 1 e não haja mais trocas no percurso pelo vetor. Assim, a função Comb Sort realiza iterativamente percorridos pelo vetor, comparando e trocando elementos a uma distância específica, reduzindo gradativamente essa distância.

6.2. Apresentação dos dados e situações

CombSort	
Melhor Caso **Geração Aleatória – Forma 1**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0
$\mathbf{10^5}$	0
$\mathbf{5 * 10^5}$	0,047
$\mathbf{10^6}$	0,115

CombSort	
Caso Médio **Geração Aleatória – Forma 1**	
Tamanhos do Vetor (n)	**Tempo em segundos**
$\mathbf{10^4}$	0
$\mathbf{5 * 10^4}$	0
$\mathbf{10^5}$	0,015
$\mathbf{5 * 10^5}$	0,047
$\mathbf{10^6}$	0,125

CombSort	
Pior Caso Geração Aleatória – Forma 1	
Tamanhos do Vetor (n)	Tempo em segundos
10^4	0
$5 * 10^4$	0,016
10^5	0,016
$5 * 10^5$	0,063
10^6	0,125

CombSort	
Melhor Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	Tempo em segundos
10^4	0
$5 * 10^4$	0
10^5	0,015
$5 * 10^5$	0,031
10^6	0,079

CombSort	
Caso Médio Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	Tempo em segundos
10^4	0
$5 * 10^4$	0
10^5	0
$5 * 10^5$	0,047
10^6	0,093

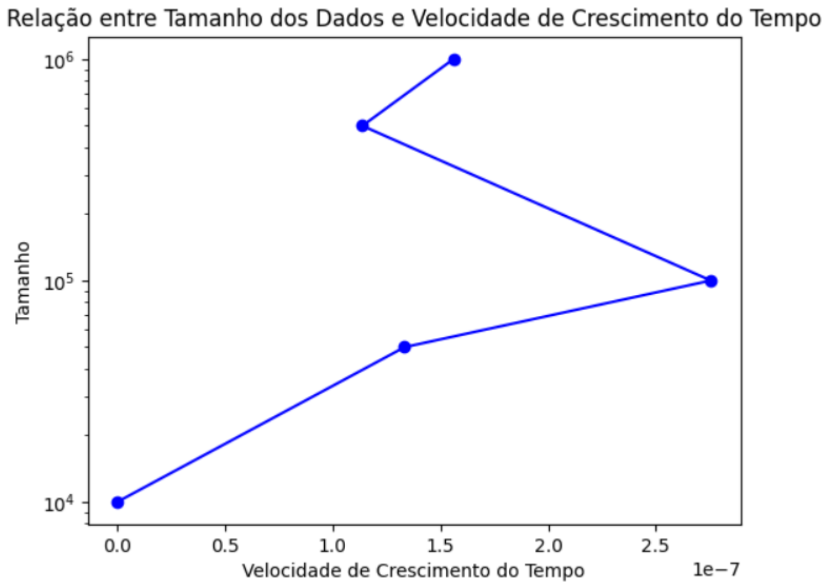
CombSort	
Pior Caso Ordem Inversa (Crescente -> Decrescente) – Forma 2	
Tamanhos do Vetor (n)	Tempo em segundos
10^4	0
$5 * 10^4$	0,016
10^5	0,016
$5 * 10^5$	0,047
10^6	0,093

6.3. Considerações

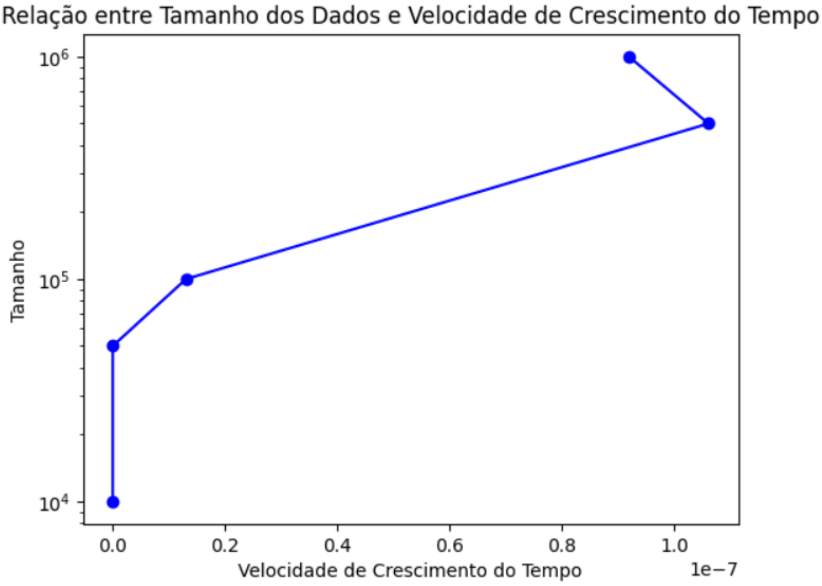
Segundo a análise das tabelas no tópico anterior, observa-se que o método CombSort possui um desempenho aproximadamente igual em ambas as formas. Além disso, mostrou ser um dos métodos de ordenação mais rápidos quando comparado com os anteriores.

O tempo médio registrado na forma 1 foi de 0,037 segundos. E o tempo médio registrado na forma 2 foi de 0,028 segundos. Pode-se considerar uma diferença extremamente baixa em relação aos outros métodos. Seguem os gráficos:

Caso médio da forma 1:



Caso médio da forma 2:



Conclusão

A análise dos métodos de ordenação revela informações valiosas sobre o desempenho dessas técnicas em diferentes cenários. Em particular, ao considerar conjuntos de dados de tamanhos variados e diferentes estados iniciais, podemos chegar a conclusões significativas.

O método de ordenação QuickSort com Pivô no Meio se destaca como a escolha mais eficiente e versátil. Independentemente do tamanho do vetor ou da natureza dos dados (aleatórios ou crescentes), o QuickSort demonstrou consistentemente alto desempenho. Sua capacidade de adaptação o coloca como uma escolha robusta para uma ampla gama de situações. Essa conclusão fundamenta-se no seu tempo médio de execução da forma 1: 0,0364 segundos, e da forma 2: 0,0156 segundos

O CombSort também se destacou, exibindo um desempenho notável em comparação com os outros métodos. Embora não tenha alcançado a eficiência do QuickSort, o CombSort mostrou ser uma opção sólida e eficaz. Essa conclusão fundamenta-se no seu tempo médio de execução da forma 1: 0,0374 segundos, e da forma 2: 0,0282 segundos

Por outro lado, o BubbleSort revelou-se o método menos eficiente, especialmente em conjuntos de dados mais extensos e, curiosamente, quando os dados não estão ordenados. Sua lentidão relativa torna-o menos recomendado para tarefas que envolvem grandes volumes de dados. Essa conclusão fundamenta-se no seu tempo médio de execução da forma 1: 569,012 segundos, e da forma 2: 443,513 segundos

Portanto, com base nessas análises, a recomendação principal seria a adoção do QuickSort com Pivô no Meio para obter a melhor performance em termos de velocidade de ordenação e adaptabilidade a diferentes condições iniciais dos dados. O CombSort também se apresenta como uma opção viável, enquanto o BubbleSort deve ser evitado em cenários que demandam eficiência na ordenação.

Referências Bibliográficas

LUCIA FILOMENA DE ALMEIDA GUIMARAES. Estrutura da Dados. Aula Presencial. Pontífice Universidade Católica de Campinas, 2023. Anotações na Lousa e Apostilas Online.

PROGRAME SEU MUNDO. YouTube, 16 de novembro de 2023. Disponível em: https://youtu.be/sWR3fWHs_Bg?si=PzFa8FxzaJk934b9. Acesso em: 16 nov. 2023.

Maloca do Código. Quicksort Descomplicado. Maloca do Código, 4 de maio de 2015. Disponível em: https://malocadocodigo.wordpress.com/2015/05/04/quicksort-descomplicado/. Acesso em: 16 nov. 2023.