CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Algoritmos e Estruturas de Dados I (EDI)
Professor: Eduardo de Lucena Falcão
Discente: Guilherme Pablo de Santana Maciel
Avaliação Unidade II

Considere os seguinte vetores:

- a = [3, 6, 2, 5, 4, 3, 7, 1]
- b = [7, 6, 5, 4, 3, 3, 2, 1]

Ilustre, em detalhes, o funcionamento dos seguintes algoritmos com os seguintes vetores: **(4.0)**

a. **BubbleSort (melhor versão) com o vetor a**

```c
void bubbleSort(struct array* vetor) {
    for (int varredura = 0; varredura < vetor->tamanho - 1; varredura++)
    {
        bool validation = false;
        for (int i = 0; i < vetor->tamanho - 1 - varredura; i++)
        {
            if (vetor->elementos[i] > vetor->elementos[i+1]) {
                int temp = vetor->elementos[i];
                vetor->elementos[i] = vetor->elementos[i + 1];
                vetor->elementos[i + 1] = temp;
                validation = true;
            }
        }
        if (!validation) {
            printf("Array ja esta ordenado \n");
            return;
        }
    }
}
```

função BubbleSort (melhor versão), esta função verifica se o vetor já está ordenado caso esteja ordenado o processo será interrompida.

Quando bloco amarelo for maior que o bloco verde os elementos troca de posição no array

tam = 8,

amarelo > verde se for troca de posição

varredura	index	tam - 1 - varredura	Array							
0	0	0 < 7	3	6	2	5	4	3	7	1
0	1	1 < 7	3	6	2	5	4	3	7	1
0	2	2 < 7	3	2	6	5	4	3	7	1
0	3	3 < 7	3	2	5	6	4	3	7	1
0	4	4 < 7	3	2	5	4	6	3	7	1
0	5	5 < 7	3	2	5	4	3	6	7	1
0	6	6 < 7	3	2	5	4	3	6	7	1

final da 1ª varredura

3	2	5	4	3	6	1	7

varredura	index	tam - 1 - varredura	Array							
1	0	0 < 6	3	2	5	4	3	6	1	7
1	1	1 < 6	2	3	5	4	3	6	1	7
1	2	2 < 6	2	3	5	4	3	6	1	7
1	3	3 < 6	2	3	4	5	3	6	1	7
1	4	4 < 6	2	3	4	3	5	6	1	7
1	5	5 < 6	2	3	4	3	5	6	1	7

final da 2ª varredura

2	3	4	3	5	1	6	7

tam = 8,

amarelo > verde se for troca de posição

varredura	index	tam - 1 - varredura	Array							
2	0	0 < 5	2	3	4	3	5	1	6	7
2	1	1 < 5	2	3	4	3	5	1	6	7
2	2	2 < 5	2	3	4	3	5	1	6	7
2	3	3 < 5	2	3	3	4	5	1	6	7
2	4	4 < 5	2	3	3	4	5	1	6	7

final da 3ª varredura

2	3	3	4	1	5	6	7

varredura	index	tam - 1 - varredura	Array							
3	0	0 < 4	2	3	3	4	1	5	6	7
3	1	1 < 4	2	3	3	4	1	5	6	7
3	2	2 < 4	2	3	3	4	1	5	6	7
3	3	3 < 4	2	3	3	4	1	5	6	7

final da 4ª varredura

2	3	3	1	4	5	6	7

varredura	index	tam - 1 - varredura	Array							
4	0	0 < 3	2	3	3	1	4	5	6	7
4	1	1 < 3	2	3	3	1	4	5	6	7
4	2	2 < 3	2	3	3	1	4	5	6	7

final da 5ª varredura

2	3	1	3	4	5	6	7

varredura	index	tam - 1 - varredura	Array							
5	0	0 < 2	2	3	1	3	4	5	6	7
5	1	1 < 2	2	3	1	3	4	5	6	7

final da 6ª varredura	2	1	3	3	4	5	6	7

varredura	index	tam - 1 - varredura	Array							
6	0	0 < 1	2	1	3	3	4	5	6	7

final da 7ª varredura	1	2	3	3	4	5	6	7

b. InsertionSort (in-place, melhor versão) com o vetor b

```c
void insertionSort(struct array* vetor) {
    for (int i = 1; i < vetor->qtdade; i++) {
        int selecionado = vetor->elementos[i];
        int j;
        for ( j = i; j > 0 && vetor->elementos[j-1] > selecionado; j--){
            vetor->elementos[j] = vetor->elementos[j - 1];
        }
        vetor->elementos[j] = selecionado;
    }
}
```

Função insertinSort versão com menos operações de troca

Nesta função o trocar acontece no final do código, assim gastando menos processamento do servidor. A troca acontecerá se o índice J for maior que zero e se o elemento anterior for maior que o selecionado.

legenda : o bloco verde é o elemento [J - 1]

tam = 8

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
1	1	6	1 > 0 && 7 > 6	7	6	5	4	3	3	2	1	
1	0	6	0 > 0 && 7 > 6	7	7	5	4	3	3	2	1	

Final da 1ª varredura array[j] = selec… array [0] = 6

6	7	5	4	3	3	2	1

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
2	2	5	2 > 0 && 7 > 5	6	7	5	4	3	3	2	1	
2	1	5	1 > 0 && 6 > 5	6	7	7	4	3	3	2	1	
2	0	5	1>0	6	6	7	4	3	3	2	1	

Final da 2ª varredura array[j] = selec… array [0] = 5

5	6	7	4	3	3	2	1

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
3	3	4	3 > 0 && 7 > 4	5	6	7	4	3	3	2	1	
3	2	4	2 > 0 && 6 > 4	5	6	7	7	3	3	2	1	
3	1	4	1 > 0 && 5 > 4	5	6	6	7	3	3	2	1	
3	0	4	0 > 0	5	5	6	7	3	3	2	1	

Final da 3ª varredura array[j] = selec… array [0] = 4

4	5	6	7	3	3	2	1

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
4	4	3	4 > 0 && 7 > 3	4	5	6	7	3	3	2	1	
4	3	3	3 > 0 && 6 > 3	4	5	6	7	7	3	2	1	
4	2	3	2 > 0 && 5 > 3	4	5	6	6	7	3	2	1	
4	1	3	1 > 0 && 4 > 3	4	5	5	6	7	3	2	1	
4	0	3	0 > 0	4	4	5	6	7	3	2	1	

Final da 4ª varredura array[j] = selec… array [0] = 3

3	4	5	6	7	3	2	1

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
5	5	3	5 > 0 && 7 > 3	3	4	5	6	7	3	2	1	
5	4	3	4 > 0 && 6 > 3	3	4	5	6	7	7	2	1	
5	3	3	3 > 0 && 5 > 3	3	4	5	6	6	7	2	1	
5	2	3	2 > 0 && 4 > 3	3	4	5	5	6	7	2	1	
5	1	3	1 > 0 && 3 > 3	3	4	4	5	6	7	2	1	

Final da 5ª varredura array[j] = selec… array [1] = 3

3	3	4	5	6	7	2	1

index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
6	6	2	6 > 0 && 7 > 2	3	3	4	5	6	7	2	1	
6	5	2	5 > 0 && 6 > 2	3	3	4	5	6	7	7	1	
6	4	2	4 > 0 && 5 > 2	3	3	4	5	6	6	7	1	
6	3	2	3 > 0 && 4 > 2	3	3	4	5	5	6	7	1	
6	2	2	2 > 0 && 3 > 2	3	3	4	4	5	6	7	1	
6	1	2	1 > 0 && 3 > 2	3	3	3	4	5	6	7	1	
6	0	2	0 > 0	3	3	3	4	5	6	7	1	

Final da 6ª varredura array[j] = selec... array [0] = 2

2	3	3	4	5	6	7	1

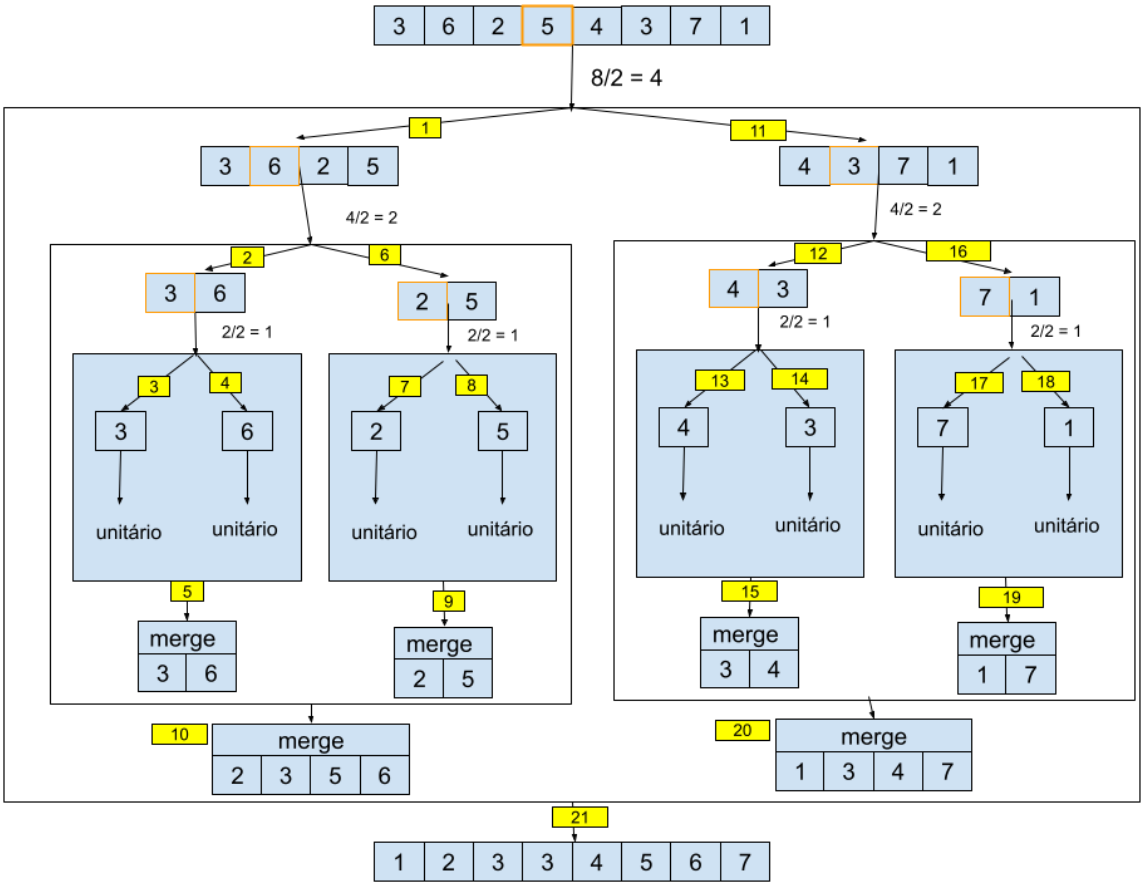
index i	index j	selec...	j > 0 && ■ > selec...	0	1	2	3	4	5	6	7	index
				array								
7	7	1	7 > 0 && 7 > 1	2	3	3	4	5	6	7	1	
7	6	1	6 > 0 && 6 > 1	2	3	3	4	5	6	7	7	
7	5	1	5 > 0 && 5 > 1	2	3	3	4	5	6	6	7	
7	4	1	4 > 0 && 4 > 1	2	3	3	4	5	5	6	7	
7	3	1	3 > 0 && 3 > 1	2	3	3	4	4	5	6	7	
7	2	1	2 > 0 && 3 > 1	2	3	3	3	4	5	6	7	
7	1	1	1 > 0 && 2 > 1	2	3	3	3	4	5	6	7	
7	0	1	0 > 0	2	2	3	3	4	5	6	7	

Final da 7ª varredura array[j] = selec... array [0] = 1

1	2	3	3	4	5	6	7

c. MergeSort com o vetor a

Os blocos amarelos enumerados mostram a sequência da função e as bordas laranja mostram onde o vetor vai ser repartido.



no passo 21 se faz um merge entre o passado 10 é o passo 20 para obter o vetor ordenado.

d. QuickSort (s/ randomização de pivô) com o vetor b

Os blocos laranja representam o pivô selecionado, o pIndex é a posição onde o pivô irá ficar após a finalização.

O pIndex só será incrementado se o elemento na posição v[i] <= pivô, neste caso incrementando o pIndex, o índice i será incrementado a cada interação até chegar no final. Quando as interações chegam no fim, logo após isso o pivô vai para posição do pIndex.

7	6	5	4	3	3	2	1

				Index							
				0	1	2	3	4	5	6	7
pIndex	i	pivô	V[i] <= pivo	V							
0	0	1	7 <= 1	7	6	5	4	3	3	2	1
0	1	1	6 <= 1	7	6	5	4	3	3	2	1
0	2	1	5 <= 1	7	6	5	4	3	3	2	1
0	3	1	4 <= 1	7	6	5	4	3	3	2	1
0	4	1	3 <= 1	7	6	5	4	3	3	2	1
0	5	1	3 <= 1	7	6	5	4	3	3	2	1
0	6	1	2 <= 1	7	6	5	4	3	3	2	1
0	7	1	1 <= 1	7	6	5	4	3	3	2	1

v[pIndex] = pivô logo v[0] = 1
v[fim] = 7

1	6	5	4	3	3	2	7

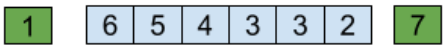
quickSort(v,0,0 -1)

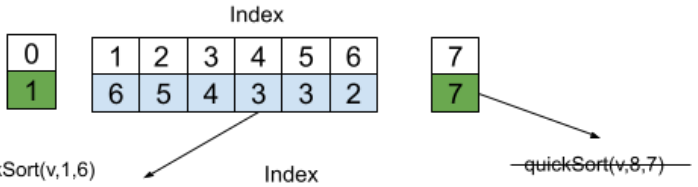
	Index						
0	1	2	3	4	5	6	7
1	6	5	4	3	3	2	7

quickSort(v,1,7)

				Index							
				1	2	3	4	5	6	7	
pIndex	i	pivô	V[i] <= pivo	V							
1	1	7	6 <= 7	6	5	4	3	3	2	7	
2	2	7	5 <= 7	6	5	4	3	3	2	7	
3	3	7	4 <= 7	6	5	4	3	3	2	7	
4	4	7	3 <= 7	6	5	4	3	3	2	7	
5	5	7	3 <= 7	6	5	4	3	3	2	7	
6	6	7	2 <= 7	6	5	4	3	3	2	7	
7	7	7	7 <= 7	6	5	4	3	3	2	7	

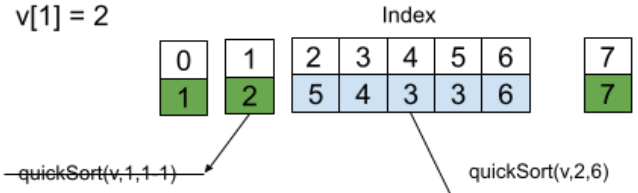
v[pIndex] = pivô logo v[7] = 7
v[fim] = 7



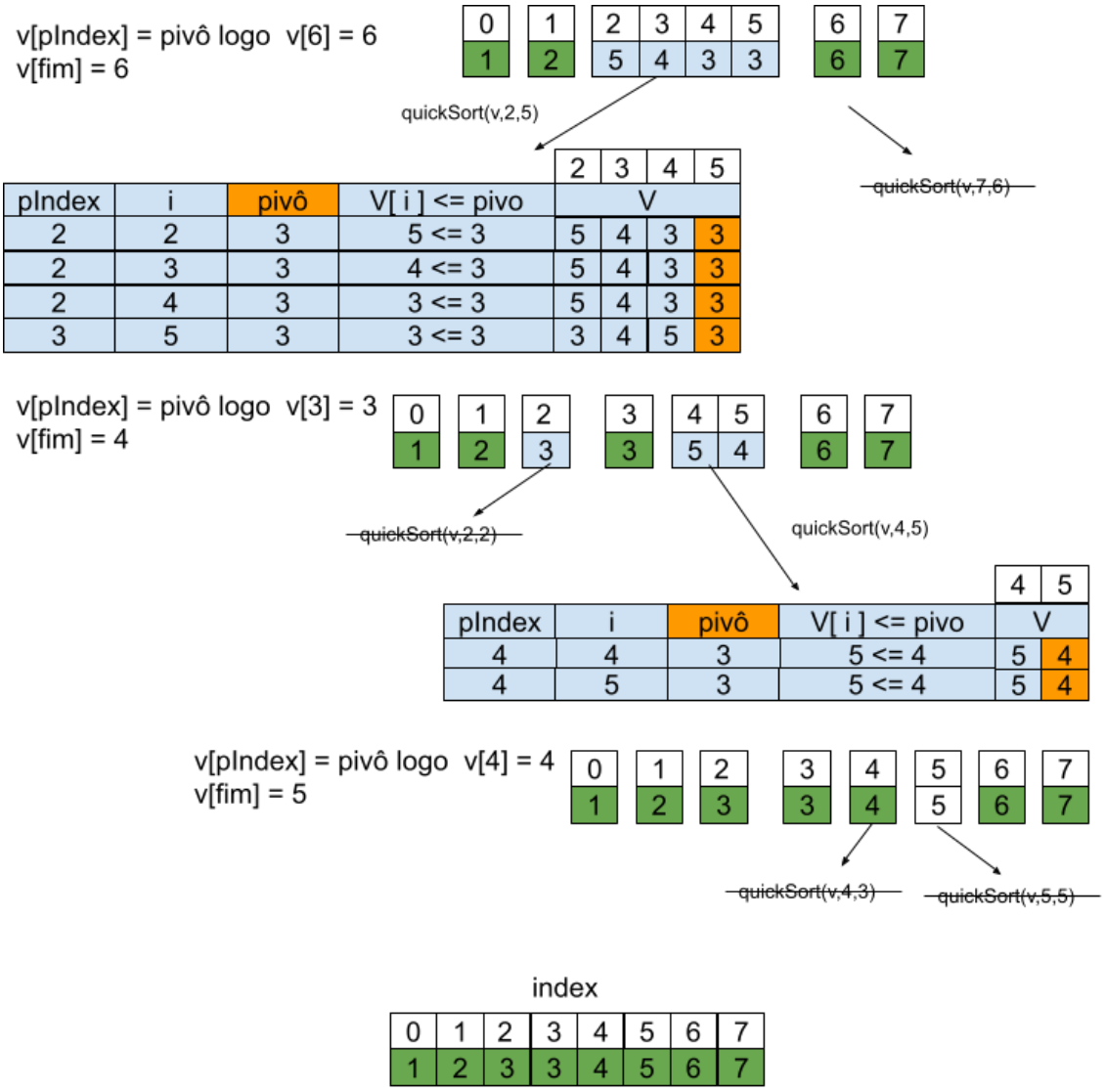


pIndex	i	pivô	V[i] <= pivo	Index 1	2	3	4	5	6
				V					
1	1	2	6 <= 2	6	5	4	3	3	2
1	2	2	5 <= 2	6	5	4	3	3	2
1	3	2	4 <= 2	6	5	4	3	3	2
1	4	2	3 <= 2	6	5	4	3	3	2
1	5	2	3 <= 2	6	5	4	3	3	2

v[pIndex] = pivô logo v[1] = 2
v[fim] = 6



pIndex	i	pivô	V[i] <= pivo	2	3	4	5	6
			V					
2	2	6	5 <= 6	5	4	3	3	6
3	3	6	4 <= 6	5	4	3	3	6
4	4	6	3 <= 6	5	4	3	3	6
5	5	6	3 <= 6	5	4	3	3	6



2. Implemente o QuickSort com seleção randomizada do pivô. (1.0)

```
void swap(int* v, int i, int j) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

```c
int particionaPivoRandomizada(int* v, int ini, int fim) {

    // gerar numero Randomizada entre ini e fim
    int pivot_index = ini + rand() % (fim - ini + 1);

    swap(v, pivot_index, fim);

    int pIndex = ini;
    int pivo = v[fim];
    for (int i = ini; i < fim; i++) {
        if (v[i] <= pivo) {
            swap(v, i, pIndex);
            pIndex++;
        }
    }
    swap(v, pIndex, fim);
    return pIndex;
}
```

```c
void quickSortPivoRandomizada(int* v, int ini, int fim) {
    if (fim > ini) {
        int indexPivo = particionaPivoRandomizada(v, ini, fim);
        quickSortPivoRandomizada(v, ini, indexPivo - 1);
        quickSortPivoRandomizada(v, indexPivo + 1, fim);   //indexPivo já está no seu local
    }
}
```

3. Vamos fazer alguns **experimentos** com os seguintes algoritmos: **SelectionSort (in-place), BubbleSort (melhor versão), InsertionSort (in-place, melhor versão), MergeSort, QuickSort, QuickSort (com seleção randomizada de pivô) e CountingSort.** Crie vetores com os seguintes **tamanhos** 10^1, 10^3, 10^5(se julgar interessante, pode escolher outros tamanhos). Para cada tamanho, você criará um **vetor ordenado**, um **vetor com valores aleatórios**, e um **vetor ordenado de forma decrescente** (use sementes para obter valores iguais). Para cada combinação de fatores, execute 30 repetições. Compute a média e mediana dessas 30 execuções para cada combinação de fatores. Faça uma análise dissertativa sobre a performance dos algoritmos para diferentes vetores e tamanhos, explicando quais algoritmos têm boa performance em quais situações. **(5.0).**

Neste experimento será possível avaliar a performance em relação ao tempo de execução de cada função com tamanho de vetores diferentes. Na análise foi utilizado uma função de time para calcular o tempo que cada função leva para ordenar um vetor.

```
{
    LARGE_INTEGER tempoInicial, tempoFinal, freq;
    float tempoTotal;

    QueryPerformanceCounter(&tempoInicial);

        // código para análise

    QueryPerformanceCounter(&tempoFinal);
    QueryPerformanceFrequency(&freq);
   tempoTotal = (float)(tempoFinal.QuadPart - tempoInicial.QuadPart)/freq.QuadPart;

    printf("tempo = %Lf \n", tempoTotal );
}
```

função para medir o tempo das funções em Windows

No primeiro experimento foram analisadas todas as funções de ordenação com tamanho do vetor 10^1

Tabela referente aos vetores de tamanho 10^1

Função	Tipo	Tempo médio em ms	Tempo mediana em ms
SelectionSort (in-place)	**ordenado**	0.00	0.00
SelectionSort (in-place)	**aleatório**	0.00	0.00
SelectionSort (in-place)	**decrescente**	0.00	0.00
BubbleSort	**ordenado**	0.00	0.00
BubbleSort	**aleatório**	0.00	0.00
BubbleSort	**decrescente**	0.00	0.00

InsertionSort (in-place)	**ordenado**	0.00	0.00
InsertionSort (in-place)	**aleatório**	0.00	0.00
InsertionSort (in-place)	**decrescente**	0.00	0.00
MergeSort	**ordenado**	0.00	0.00
MergeSort	**aleatório**	0.00	0.00
MergeSort	**decrescente**	0.00	0.00
QuickSort	**ordenado**	0.00	0.00
QuickSort	**aleatório**	0.00	0.00
QuickSort	**decrescente**	0.00	0.00
QuickSort (com seleção randomizada de pivô)	**ordenado**	0.00	0.00
QuickSort (com seleção randomizada de pivô)	**aleatório**	0.00	0.00
QuickSort (com seleção randomizada de pivô)	**decrescente**	0.00	0.00
CountingSort	**ordenado**	0.00	0.00
CountingSort	**aleatório**	0.00	0.00
CountingSort	**decrescente**	0.00	0.00

Analisando os resultados da média e mediana das funções, nota-se que as funções de ordenação tiveram um resultado excelente com vetor de 10 posições, qualquer uma das função irá ter um bom desempenho já que todas tiveram o mesmo resultado aparente.

No segundo experimento foram analisadas todas as funções de ordenação com tamanho do 10^3

Tabela referente aos vetores de tamanho 10^3

Função	Tipo	Tempo médio em ms	Tempo mediana em ms
SelectionSort (in-place)	**ordenado**	1.14	1.14
SelectionSort (in-place)	**aleatório**	1.49	1.03
SelectionSort (in-place)	**decrescente**	1.04	1.03
BubbleSort	**ordenado**	0.00	0.00
BubbleSort	**aleatório**	1.84	1.83
BubbleSort	**decrescente**	2.05	2.03
InsertionSort (in-place)	**ordenado**	0.00	0.00
InsertionSort (in-place)	**aleatório**	0.91	0.80
InsertionSort (in-place)	**decrescente**	1.03	1.01
MergeSort	**ordenado**	0.58	0.57
MergeSort	**aleatório**	0.61	0.61
MergeSort	**decrescente**	0.57	0.58

QuickSort	**ordenado**	2.48	2.46
QuickSort	**aleatório**	0.09	0.09
QuickSort	**decrescente**	1.78	1.78
QuickSort (com seleção randomizada de pivô)	**ordenado**	0.11	0.11
QuickSort (com seleção randomizada de pivô)	**aleatório**	0.13	0.13
QuickSort (com seleção randomizada de pivô)	**decrescente**	0.11	0.11
CountingSort	**ordenado**	0.02	0.01
CountingSort	**aleatório**	0.02	0.02
CountingSort	**decrescente**	0.02	0.01

Analisando os resultados do tempo das funções acima, nota-se que **SelectionSort (in-place), BubbleSort é QuickSort** tiveram um tempo maior que 1.2 ms, a média entre os vetores ordenado, aleatório e decrescente. As outras funções tiveram um desempenho melhor, pois a média entre os vetores foi menor que 0 ms.

No terceiro experimento foram analisadas todas as funções de ordenação com tamanho do 10^5

Tabela referente aos vetores de tamanho 10^5

Função	Tipo	Tempo médio em ms	Tempo mediana em ms
SelectionSort (in-place)	**ordenado**	9245.73	9240.95
SelectionSort (in-place)	**aleatório**	9250.49	9234.70
SelectionSort (in-place)	**decrescente**	9463.80	9458.55
BubbleSort	**ordenado**	0.19	0.19
BubbleSort	**aleatório**	28734.58	28977.15
BubbleSort	**decrescente**	20794.37	20755.07
InsertionSort (in-place)	**ordenado**	0.33	0.32
InsertionSort (in-place)	**aleatório**	5153.54	5166.17
InsertionSort (in-place)	**decrescente**	10232.69	10203.16
MergeSort	**ordenado**	58.86	58.78
MergeSort	**aleatório**	68.46	66.22
MergeSort	**decrescente**	59.28	59.15
QuickSort	**ordenado**	error	error
QuickSort	**aleatório**	14.52	14.34
QuickSort	**decrescente**	error	error
QuickSort (com seleção randomizada de pivô)	**ordenado**	14.45	14.06

QuickSort (com seleção randomizada de pivô)	**aleatório**	20.31	20.21
QuickSort (com seleção randomizada de pivô)	**decrescente**	15.17	15.15
CountingSort	**ordenado**	1.55	1.52
CountingSort	**aleatório**	2.89	2.85
CountingSort	**decrescente**	1.52	01.51

Analisando os resultados do tempo das funções acima.

O SelectionSort (in-place) teve uma média maior que 9 segundos isso se dar pela forma como algoritmo é construído dizemos que é $O(n^2)$, ou seja quanto maior o vetor pior desempenho.

O **BubbleSort** nas análises teve um resultado pior que o **SelectionSort** nos vetores aleatório e decrescente, porém dizemos que esta função é $O(n^2)$ no pior caso, O(n) no melhor caso.

O **MergeSort** nas análises teve um desempenho excelente a média entre os 3 vetores ordenado, aleatório e decrescente foi de 62.2 ms é a maior mediana entre eles foi de 66.22 ms, o bom tempo se dar pela arquitetura do algoritmo que dizemos que é O(nlogn).

O **QuickSort** nas análises teve dois desempenho um excelente é outro indeterminado isso se dar pela arquitetura do algoritmo, esta função tem a complexidade de tempo O(nlogn) no melhor caso que foi no vetor aleatório, no pior caso a complexidade de tempo O(n^2) isso ocorre pela escolha do pivô que nesta situação é o último elemento do array (pode ser evitado randomizando a escolha do pivô).

O **QuickSort (com seleção randomizada de pivô)** nas análises teve um desempenho excelente com essa seleção do pivô pode-se evitar o pior caso que seria complexidade de tempo O(n^2), com isso a média entre os 3 vetores ordenado, aleatório e decrescente foi de 16.64 ms é a maior mediana entre eles foi de 20.21 ms.

O **CountingSort** nas análises teve um desempenho excelente com a complexidade de tempo O(n), com isso a média entre os 3 vetores ordenado, aleatório e decrescente foi de 1,98 ms é a maior mediana entre eles foi de 2.8 ms. O countingSort é um algoritmo muito bom, desde

que os intervalos entre os números não sejam muito grande caso contrário irá usar mais processamento, já que esta função seleciona o menor elemento e o maior elemento do vetor é criar um vetor com essa diferença + 1. Porém mesmo assim o countingSort teve o melhor desempenho entre todas as análises.

Conclusão

Vetores com tamanho de até 10^3 Pode-se usar qualquer uma das funções de ordenação, pois o desempenho entre elas não varia muito entre si. Com os vetores maior que 10^3 **recomenda utilizar uma dessas funções na ordem de desempenho: CountingSort , QuickSort (com seleção randomizada de pivô) é MergeSort** estas funções tiveram um desempenho excelente nas análise.