CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Algoritmos e Estruturas de Dados I (EDI)
Professor: Eduardo de Lucena Falcão
Avaliação Unidade II
Aluno: Lucas de Oliveira Umbelino

Considere os seguinte vetores:

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

1. Ilustre, em detalhes, o funcionamento dos seguintes algoritmos com os seguintes vetores: **(4.0)**
    a. SelectionSort (in-place) com o vetor a
    b. BubbleSort (melhor versão) com o vetor a
    c. InsertionSort (in-place, melhor versão) com o vetor b
    d. MergeSort com o vetor a
    e. QuickSort (s/ randomização de pivô) com o vetor b
2. Implemente o QuickSort com seleção randomizada do pivô. **(1.0)**
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 (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.algoritmos para diferentes vetores e tamanhos, explicando quais algoritmos têm boa performance em quais situações. **(5.0)**

RESPOSTAS

Questão 1

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

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

Ambos os vetores têm tamanho n=8.

1a. SelectionSort (in-place) com o vetor a

Nesta ordenação, a cada interação de **i,** iremos selecionar o menor valor do vetor e colocá-lo na posição correta do vetor (no fim de cada iteração de i, iremos trocar os valores).

O número marcado em laranja é o menor valor de determinada iteração. No caso, estamos armazenando o índice do menor valor (int iMenor = i).

O número marcado em verde é o número que será comparado com o menor valor naquele determinado momento (v[j]).

O número marcado em azul já está ordenado.

Início	3	6	2	5	4	3	7	1
n-1	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
	3	6	2	5	4	3	7	1
n-2	1	6	2	5	4	3	7	3
	1	6	2	5	4	3	7	3
	1	6	2	5	4	3	7	3
	1	6	2	5	4	3	7	3
	1	6	2	5	4	3	7	3
	1	6	2	5	4	3	7	3
n-3	1	2	6	5	4	3	7	3
	...							
	1	2	6	5	4	3	7	3
n-4	1	2	3	5	4	6	7	3
n-5	1	2	3	3	4	6	7	5

n-6	1	2	3	3	4	6	7	5
n-7	1	2	3	3	4	5	7	6
Fim	1	2	3	3	4	5	6	7

1b. BubbleSort (melhor versão) com o vetor a

O bubbleSort funciona da seguinte maneira, o maior número sempre vai estar indo a direita do vetor (como se fosse realmente uma bolha). Caso em algum momento a variável de flag chamado “trocou” após o fim do segundo for permanecer com um valor igual a falso, já podemos dar um “return” pois o vetor já está ordenado.

A coloração em laranja é a comparação dos termos do vetor (nesse caso, sempre compara v[i] com v[i+1]).

Início	Trocou	3	6	2	5	4	3	7	1
n-1	false	3	6	2	5	4	3	7	1
	false	3	6	2	5	4	3	7	1
	true	3	2	6	5	4	3	7	1
	true	3	2	5	6	4	3	7	1
	true	3	2	5	4	6	3	7	1
	true	3	2	5	4	3	6	7	1
	true	3	2	5	4	3	6	7	1
n-2	false	3	2	5	4	3	6	1	7
	true	2	3	5	4	3	6	1	7
	true	2	3	5	4	3	6	1	7
	true	2	3	4	5	3	6	1	7
	true	2	3	4	3	5	6	1	7
	true	2	3	4	3	5	6	1	7
n-3	false	2	3	4	3	5	1	6	7
n-4	false	2	3	3	4	1	5	6	7
n-5	false	2	3	3	1	4	5	6	7
n-6	false	2	3	1	3	4	5	6	7
n-7	false	2	1	3	3	4	5	6	7
Fim	– – –	1	2	3	3	4	5	6	7

1c. InsertionSort (in-place, melhor versão) com o vetor b

Em InsertionSort, pode ser confundido bastante com o selectionSort, mas no caso do Insertion, sempre é selecionado um valor (sem levar em conta o primeiro índice, já que ele seria o primeiro valor da ordenação,logo já estaria ordenado) e o respectivo valor é comparado com os valores já ordenados, enquanto o valor salvo for menor do que os valores já ordenado, move-se os valores para a direita do vetor até encontrar o determinado índice que o valor deve ser armazenado no vetor.

Os números acompanhados de asterisco, querem dizer que estão ordenados.

Sempre que a condição (j > 0 && v[j - 1] > valor) for verdadeira, haverá a cor verde sinalizada na comparação.Caso seja falso, será atribuída a cor laranja.

A cor amarela sinaliza que o índice ao qual o Valor vai ser armazenado.

Início	j	Valor	7*	6	5	4	3	3	2	1
n-7	=1	=6	7*	6	5	4	3	3	2	1
	=0	=6	7*	7*	5	4	3	3	2	1
	=0	=6	7*	7*	5	4	3	3	2	1
	=0	=6	6*	7*	5	4	3	3	2	1
n-6	=2	=5	6*	7*	5	4	3	3	2	1
	=1	=5	6*	7*	7*	4	3	3	2	1
	=0	=5	6*	6*	7*	4	3	3	2	1
	=0	=5	6*	6*	7*	4	3	3	2	1
	=0	=5	5*	6*	7*	4	3	3	2	1
n-5	=3	=4	5*	6*	7*	4	3	3	2	1
	=2	=4	5*	6*	7*	7*	3	3	2	1
	=1	=4	5*	6*	6*	7*	3	3	2	1
	=0	=4	5*	5*	6*	7*	3	3	2	1
	=0	=4	5*	5*	6*	7*	3	3	2	1
	=0	=4	4*	5*	6*	7*	3	3	2	1
n-4	=4	=4	4*	5*	6*	7*	3	3	2	1
	...									
	=0	=3	3*	4*	5*	6*	7*	3	2	1
n-4	=5	=3	3*	4*	5*	6*	7*	3	2	1
	=4	=3	3*	4*	5*	6*	7*	7*	2	1
	=3	=3	3*	4*	5*	6*	6*	7*	2	1
	=2	=3	3*	4*	5*	5*	6*	7*	2	1
	=1	=3	3*	4*	4*	5*	6*	7*	2	1
	=1	=3	3*	4*	4*	5*	6*	7*	2	1

	=1	=3	3*	3*	4*	5*	6*	7*	2	1
n-2	=6	=2	3*	3*	4*	5*	6*	7*	2	1
	...									
	=0	=2	2*	3*	3*	4*	5*	6*	7*	1
n-1	=7	=1	2*	3*	3*	4*	5*	6*	7*	1
	...									
	=0	=1	1*	2*	3*	3*	4*	5*	6*	7*

1d. MergeSort com o vetor a

Em MergeSort, a ordenação funciona de seguinte maneira: devemos pegar o vetor e sempre dividi-los em dois novos vetores(enquanto o vetor possui tamanho maior do que 1), até chegar um ponto em que vamos ter **n** vetores de tamanho 1, após essa divisão, iremos fazer o merge com esses vetores (comparando os valores e juntando eles) até no final sobrar o vetor original novamente já ordenado.

(i) Primeiro fazemos a chamada da função mergeSort, passando o **vetor v** como parâmetro.

(ii) Após a divisão do vetor, há a chamada da função mergeSort passando como parâmetro os subvetores divididos.

(iii) Depois temos de novo a chamada da função mergeSort, assim, fazendo com que tenham N vetores.

(iiii) Após as divisões, agora devemos fazer a junção (merge) dos vetores novamente, só que fazendo a devida ordenação no vetor original de cada iteração.

(i)	[3,6,2,5,4,3,7,1]							
(ii)	[3,6,2,5]				[4,3,7,1]			
(iii)	[3,6]		[2,5]		[4,3]		[7,1]	
(iii)	[3]	[6]	[2]	[5]	[4]	[3]	[7]	[1]

t1	[3]	[6]	[2]	[5]	[4]	[3]	[7]	[1]
	3<6		2<5		3<4		1<7	
	[3]		[2]		[3]		[1]	
	[3]	[6]	[2]	[5]	[4]	[3]	[7]	[1]
	sobrou 6		sobrou 5		sobrou 4		sobrou 7	
	[3,6]		[2,5]		[3,4]		[1,7]	
t2	[3,6]		[2,5]		[3,4]		[1,7]	
	2<3				1<3			

	[2]		[1]	
	[3,6]	[2,5]	[3,4]	[1,7]
	3<5		3<7	
	[2,3]		[1,3]	
	[3,6]	[2,5]	[3,4]	[1,7]
	5<6		4<7	
	[2,3,5]		[1,3,4]	
	[3,6]	[2,5]	[3,4]	[1,7]
	sobrou 6		sobrou 7	
	[2,3,5,6]		[1,3,4,7]	
t3	[2,3,5,6]		[1,3,4,7]	
	...			
	[1,2,3,3,4,5,6,7]			

1e. QuickSort (s/ randomização de pivô) com o vetor b

Nesta ordenação, nós trabalhamos com a recursividade igual ao mergeSort, só que uma grande diferença é que o quickSort trabalha com o vetor in-place, alias do mergeSort que trabalha com out-of-place.

Funciona da seguinte maneira: selecionamos um pivô e particionamos o vetor com base no pivô, todos os valores abaixo do piso vão para a esquerda e os maiores valores vão para a direita. A Partir daí seguimos com a chamada da função novamente, só que particionando o lado direito e esquerdo do vetor.

Grifados em verde e rosa, é a comparação de v[i] <= pivo, caso seja verdadeira haverá a troca entre v[pIndex] e o número marcado em verde.

Os números grifados em azul são os valores já na sua devida posição.

pIndex	pivô	início	fim	vetor							
—-----	—---	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1
0	1	0	7	7	6	5	4	3	3	2	1

troca entre v[pIndex] e pivô				1	6	5	4	3	3	2	7
1	7	1	7	1	6	5	4	3	3	2	7
troca entre 6 e v[pIndex]				1	6	5	4	3	3	2	7
2	7	1	7	1	6	5	4	3	3	2	7
troca entre 5 e v[pIndex]				1	6	5	4	3	3	2	7
3	7	1	7	1	6	5	4	3	3	2	7
troca entre 4 e v[pIndex]				1	6	5	4	3	3	2	7
4	7	1	7	1	6	5	4	3	3	2	7
...											
troca entre 2 e 2 (valor do pIndex)				1	6	5	4	3	3	2	7
7	7	1	7	1	6	5	4	3	3	2	7
troca entre v[pIndex] e pivô				1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
1	2	1	6	1	6	5	4	3	3	2	7
troca entre v[pIndex] e pivô				1	2	5	4	3	3	6	7
...											
troca entre v[pIndex] e pivô				1	2	5	4	3	3	6	7
...											
troca entre v[pIndex] e pivô				1	2	3	4	3	5	6	7
...											
troca entre v[pIndex] e pivô				1	2	3	4	3	5	6	7
...											
troca entre v[pIndex] e pivô				1	2	3	3	4	5	6	7

Questão 2

```
int particiona(int* v, int ini, int fim) {
    int pIndex = ini;


    //Com randomização
    int tamanho = fim - ini + 1;
    int indiceAleatorio = (rand() % tamanho) + ini;
    troca(v,indiceAleatorio,fim);


    int pivo = v[fim];

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

    troca(v, pIndex, fim);
    return pIndex;
}

void quickSort(int* v, int ini, int fim) {
    if (fim>ini) {
        int indexPivo = particiona(v, ini, fim);
        quickSort(v, ini, indexPivo - 1);
        quickSort(v, indexPivo + 1, fim);
    }
}
```

Questão 3

Nesta parte iremos testar todas essas ordenações: SelectionSort (in-place), BubbleSort (melhor versão), InsertionSort (in-place, melhor versão), MergeSort, QuickSort, QuickSort (com seleção randomizada de pivô) e CountingSort. E fazer comparações de performance, destacando a média de tempo de execução e do porquê seria mais efetivo para determinados casos.Todos os testes funcionaram da seguinte forma, vamos criar sempre um vetor de tamanho 10, 1.000 e 100.000, variando entre um vetor ordenado de forma crescente,, um vetor com valores aleatórios e um vetor ordenado de forma decrescente. No fim, teremos uma tabela com as informações e suas devidas médias no tempo de execução e minha opinião sobre o método mais eficiente e mais viável.

Usando como base, esse tipo de código para adquirir os tempos de todos os métodos de ordenação::

```c
int main()
{
    int tamanho = 10000;
    int* v= (int*) malloc (tamanho * sizeof(int));
    double start, stop, elapsed, soma=0, tempos[30];
    for(int n=0; n<30; n++){
        srand (time(NULL));

        for(int i=0; i<tamanho; i++){
            //v[i] = i;
            //v[i] = rand() % 10000;
            v[i] = tamanho - i;
        }
        start = (double) clock () / CLOCKS_PER_SEC;

        //TIPO DE ORDENAÇÃO

        stop = (double) clock () / CLOCKS_PER_SEC;
        elapsed = stop - start;
        tempos[n] = elapsed;
        soma+=elapsed;
        printf("Caso %d:Tempo em segundo: %f \n", n+1 ,elapsed);
    }
    double media = soma / 30;
    printf("Media = %f \n", media);
    ordenar(tempos, 30);
    double mediana = (tempos[14] + tempos[15]) / 2;
    printf("Mediana = %f \n", mediana);
    return 0;
}
```

-SelectionSort (in-place)

SelectionSort (in-place)			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,005100	0,005000
Aleatória	1.000	0,004733	0,005000
Decrescente	1.000	0,005400	0,005000
Crescente	10.000	0,431567	0,430000
Aleatória	10.000	0,436567	0,432000
Decrescente	10.000	0,452400	0,450000

-BubbleSort (melhor versão)

BubbleSort (melhor versão)

Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,000000	0,000000
Aleatória	1.000	0,012200	0,011000
Decrescente	1.000	0,010600	0,010000
Crescente	10.000	0,000133	0,000000
Aleatória	10.000	1,050833	1,047000
Decrescente	10.000	0,996733	0,996000

-InsertionSort (in-place, melhor versão)

InsertionSort (in-place, melhor versão)			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,000000	0,000000
Aleatória	1.000	0,008067	0,008000
Decrescente	1.000	0,018233	0,015000
Crescente	10.000	0,000400	0,000000
Aleatória	10.000	0,878067	0,875000
Decrescente	10.000	1,763533	1,761000

-MergeSort

MergeSort			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000033	0,000000

Crescente	1.000	0,001033	0,001000
Aleatória	1.000	0,000933	0,001000
Decrescente	1.000	0,001300	0,001000
Crescente	10.000	0,012533	0,008000
Aleatória	10.000	0,011633	0,010000
Decrescente	10.000	0,009667	0,008500

-QuickSort

QuickSort			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,003633	0,004000
Aleatória	1.000	0,000133	0,000000
Decrescente	1.000	0,002333	0,002000
Crescente	10.000	0,338867	0,338000
Aleatória	10.000	0,001333	0,001333
Decrescente	10.000	0,231500	0,228000

-QuickSort (com seleção randomizada de pivô)

QuickSort (com seleção randomizada de pivô)			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000000	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,000533	0,000000
Aleatória	1.000	0,000133	0,000000
Decrescente	1.000	0,000067	0,000000
Crescente	10.000	0,001033	0,001000

Aleatória	10.000	0,001633	0,002000
Decrescente	10.000	0,001400	0,001000

-CountingSort

CountingSort			
Vetor original na ordem	Tamanho	tempo de execução (Média em segundo)	Tempo de execução (Mediana em segundo)
Crescente	10	0,000000	0,000000
Aleatória	10	0,000033	0,000000
Decrescente	10	0,000000	0,000000
Crescente	1.000	0,000000	0,000000
Aleatória	1.000	0,000067	0,000000
Decrescente	1.000	0,000000	0,000000
Crescente	10.000	0,000267	0,000000
Aleatória	10.000	0,000267	0,000000
Decrescente	10.000	0,000233	0,000000

Após os testes, se analisarmos bem cada tipo de ordenação, podemos identificar que todas as ordenações são eficientes para vetores com tamanhos bem pequenos, nesses casos, tanto faz a ordenação contanto que sejam realmente vetores pequenos. Agora para valores um pouco acima, tanto o bubble, e o insertion começam a demorar mais para ordenar os valores se comparado ao merge, quick e counting, como o valor do teste de tamanho 100 ainda não é um valor tão superior, o usuário poderia optar por qualquer um deles. Agora para valores muitos maiores, como feito pelo teste, é inviável a utilização de um bubble, selection e insertion se comparado ao programador utilizar um merge, quick ou counting quem possuem uma eficiente em tempo muito superior.

Mas também devemos levar em conta a memória consumida por cada ordenação, visando utilização em vetores bastante grandes, a utilização de um counting ou merge talvez não seria o ideal já que eles trabalham com criação de novos vetores (out-in-place), o mais ideal seria o quick (com pivô aleatório) que trabalha com o próprio vetor (in-place) e possui um boa eficiente em questão de tempo.