CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Algoritmos e Estruturas de Dados I (EDI)
Professor: Eduardo de Lucena Falcão
Aluno: Thiago Theiry de Oliveira
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]

1. Ilustre, em detalhes, o funcionamento dos seguintes algoritmos com os seguintes vetores: **(4.0)**
    a. SelectionSort (in-place) com o vetor a

```
34  void selectionSortIP(int* v, int tamanho) {
35      for (int i = 0; i < (tamanho - 1); i++) {
36
37          int iMenor = i;
38          for (int j = i + 1; j < tamanho ; j++) {
39
40              if (v[j] < v[iMenor]) {
41                  iMenor = j;
42              }
43          }
44          int temp = v[i];
45          v[i] = v[iMenor];
46          v[iMenor] = temp;
47      }
48  }
49
```

O selection sort é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim sucessivamente. O Selection Sort in-place é quando a ordenação é feita rearranjando os elementos no próprio array, ao invés de usar arrays ou outras estruturas auxiliares.

Detalhando o SelectionSort (in-place) no vetor a:

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

Para i= 0 ; iMenor = 0 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

1º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 0	j= 1	6 < 3	iMenor = 0
i= 0	j= 2	2 < 3	iMenor = 2
i= 0	j= 3	5 < 2	iMenor = 2
i= 0	j= 4	4< 2	iMenor = 2

i= 0	j= 5	3< 2	iMenor = 2
i= 0	j= 6	7< 2	iMenor = 2
i= 0	j= 7	1 < 2	iMenor = 7

Ao fim de cada varredura, encontramos o índice que possui o menor valor e fazemos a troca com o valor do índice "i". para fazer essa troca usamos um auxiliador:

```
int temp = v[i];  // temp = 3
    v[i] = v[iMenor]; // v[0] = v[7] // v[0] recebeu o valor de v[7] // v[0] = 1
    v[iMenor] = temp; // v[7] = 3
```

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

Para i= 1 ; iMenor = 1 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

2º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 1	j= 2	2 < 6	iMenor = 2
i= 1	j= 3	5 < 2	iMenor = 2
i= 1	j= 4	4 < 2	iMenor = 2
i= 1	j= 5	3 < 2	iMenor = 2
i= 1	j= 6	7 < 2	iMenor = 2
i= 1	j= 7	3 < 2	iMenor = 2

O valor a ser trocado será o valor do iMenor [2] = 2 ; pelo valor do índice [i]= 6

```
int temp = v[i];  // temp = 6
    v[i] = v[iMenor]; // v[1] = v[2] // v[1] recebeu o valor de v[2] // v[1] = 2
    v[iMenor] = temp; // v[2] = 6
```

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

Para i= 2 ; iMenor = 1 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

3º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 2	j= 3	5 < 6	iMenor = 3
i= 2	j= 4	4 < 5	iMenor = 4
i= 2	j= 5	3 < 4	iMenor = 5
i= 2	j= 6	7 < 3	iMenor = 5
i= 2	j= 7	3 < 3	iMenor = 5

O valor a ser trocado será o valor do iMenor [5] = 3 ; pelo valor do índice [i]= 6

```
int temp = v[i];  // temp = 6
    v[i] = v[iMenor]; // v[2] = v[5] // v[2] recebeu o valor de v[5] // v[2] = 3
    v[iMenor] = temp; // v[5] = 6
```

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

Para i= 3 ; iMenor = 3 ; cada vez que a condição do if for aceita o iMenor recebe índice de j

3º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 3	j= 4	4 < 5	iMenor = 4
i= 3	j= 5	6 < 4	iMenor = 4
i= 3	j= 6	7 < 4	iMenor = 5
i= 3	j= 7	3 < 4	iMenor = 7

O valor a ser trocado será o valor do iMenor [7] = 3 ; pelo valor do índice [i]= 5

```
int temp = v[i];  // temp = 5
    v[i] = v[iMenor]; // v[3] = v[7] // v[3] recebeu o valor de v[7] // v[3] = 3
    v[iMenor] = temp; // v[7] = 5
```

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

Para i= 4 ; iMenor = 4 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

4º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 4	j= 5	6 < 4	iMenor = 4
i= 4	j= 6	7 < 4	iMenor = 4
i= 4	j= 7	5 < 4	iMenor = 4

Nessa varredura o menor valor já estava na posição correta, não precisando de troca

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

Para i= 5 ; iMenor = 5 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

5º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 5	j= 6	7 < 6	iMenor = 5
i= 5	j= 7	5 < 6	iMenor = 7

O valor a ser trocado será o valor do iMenor [7] = 5 ; pelo valor do índice [i]= 6

```
int temp = v[i];  // temp = 6
    v[i] = v[iMenor]; // v[5] = v[7] // v[5] recebeu o valor de v[7] // v[5] = 5
    v[iMenor] = temp; // v[7] = 6
```

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

Para i= 6 ; iMenor = 6 ; cada vez que a condição do if for aceita o iMenor recebe o índice de j

6º Varredura	Valor de j = i + 1	if (v[j] < v[iMenor])	iMenor (índice)
i= 6	j= 7	6 < 7	iMenor = 7

O valor a ser trocado será o valor do iMenor [7] = 6 ; pelo valor do índice [i]= 7

```
int temp = v[i];  // temp = 7
    v[i] = v[iMenor]; // v[6] = v[7] // v[6] recebeu o valor de v[7] // v[6] = 6
    v[iMenor] = temp; // v[7] = 7
```

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

A varredura só irá ate (tamanho-1), pois se todos anteriores estão ordenados, logo o ultimo já estará também.

O vetor a após o uso do algoritmo => a = [1, 2, 3, 3, 4, 5, 6, 7]

b. BubbleSort (melhor versão) com o vetor a

```
86  void bubbleSort(int* v, int n) {
87      for (int i = 0; i < n - 1; i++) {
88          bool houvetroca = false;
89          for (int j = 0; j < n - i - 1; j++) {
90              if (v[j] > v[j + 1]) {
91                  int temp = v[j];
92                  v[j] = v[j + 1];
93                  v[j + 1] = temp;
94                  houvetroca = true;
95              }
96          }
97          if (houvetroca == false)
98              return;
99      }
100 }
```

A função BubbleSort possui o algoritmo de ordenação dos mais simples, já que a ideia de sua utilização é percorrer o vetor diversas vezes (se necessário) fazendo com que o maior elemento da sequencia vá para o topo. Ou seja, o objetivo é ordenar os valores em forma decrescente, então, a posição atual é comparada com a próxima posição e, se a posição atual for maior que a posição posterior, é realizada a troca dos valores nessa posição. Na sua melhor versão, a função verifica se o vetor já esta ordenado e caso esteja, o processo será interrompido.

Detalhando o BubbleSort no vetor a:

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

Se a condição do if for aceita, é realizada a troca dos valores nessa posição
O valor de j será cada vez menor em cada varredura, pois as últimas posições já estarão ordenadas.

1º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 0	j= 0	3 > 6	[3, 6, 2, 5, 4, 3, 7, 1]
i= 0	j= 1	6 > 2	[3, 2, 6, 5, 4, 3, 7, 1]
i= 0	j= 2	6 > 5	[3, 2, 5, 6, 4, 3, 7, 1]
i= 0	j= 3	6 > 4	[3, 2, 5, 4, 6, 3, 7, 1]
i= 0	j= 4	6 > 3	[3, 2, 5, 4, 3, 6, 7, 1]
i= 0	j= 5	6 > 7	[3, 2, 5, 4, 3, 6, 7, 1]
i= 0	j= 6	7 > 1	[3, 2, 5, 4, 3, 6, 1, 7]

Ao final da 1º varredura, temos [3, 2, 5, 4, 3, 6, 1, 7]

2º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 1	j= 0	3 > 2	[2, 3 , 5, 4, 3, 6, 1, 7]
i= 1	j= 1	3>5	[2, 3 , 5, 4, 3, 6, 1, 7]
i= 1	j= 2	5>4	[2, 3 , 4, 5, 3, 6, 1, 7]

i= 1	j= 3	5> 3	[2, 3 , 4, 3, 5, 6, 1, 7]
i= 1	j= 4	5 > 6	[2, 3 , 4, 3, 5, 6, 1, 7]
i= 1	j= 5	6 > 1	[2, 3 , 4, 3, 5, 1, 6, 7]

Ao final da 2º varredura, temos [2, 3 , 4, 3, 5, 1, 6, 7]

3º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 2	j= 0	2 > 3	[2, 3 , 4, 3, 5, 1, 6, 7]
i= 2	j= 1	3 > 4	[2, 3 , 4, 3, 5, 1, 6, 7]
i= 2	j= 2	4 > 3	[2, 3 , 3, 4, 5, 1, 6, 7]
i= 2	j= 3	4 > 5	[2, 3 , 3, 4, 5, 1, 6, 7]
i= 2	j= 4	5 > 1	[2, 3 , 3, 4, 1, 5, 6, 7]

Ao final da 3º varredura, temos [2, 3 , 3, 4, 1, 5, 6, 7]

4º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 3	j= 0	2 > 3	[2, 3 , 3, 4, 1, 5, 6, 7]
i= 3	j= 1	3 > 3	[2, 3 , 3, 4, 1, 5, 6, 7]
i= 3	j= 2	3 > 4	[2, 3 , 3, 4, 1, 5, 6, 7]
i= 3	j= 3	4 > 1	[2, 3 , 3, 1, 4, 5, 6, 7]

Ao final da 4º varredura, temos [2, 3 , 3, 1, 4, 5, 6, 7]

5º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 4	j= 0	2 > 3	[2, 3 , 3, 1, 4, 5, 6, 7]
i= 4	j= 1	3 > 3	[2, 3 , 3, 1, 4, 5, 6, 7]
i= 4	j= 2	3 > 1	[2, 3 , 1, 3, 4, 5, 6, 7]

Ao final da 5º varredura, temos [2, 3 , 1, 3, 4, 5, 6, 7]

6º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 5	j= 0	2 > 3	[2, 3 , 1, 3, 4, 5, 6, 7]
i= 5	j= 1	3 > 1	[2, 1, 3 , 3, 4, 5, 6, 7]

Ao final da 6º varredura, temos [2, 1, 3 , 3, 4, 5, 6, 7]

7º varredura	Valor de j	if (v[j] > v[j + 1])	Vetor
i= 6	j= 0	2 > 1	[1, 2, 3 , 3, 4, 5, 6, 7]

Ao final da 7º varredura, temos [1, 2, 3 , 3, 4, 5, 6, 7]

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

```
void insertionSortIP(int* v, int tamanho) {
    for (int i = 1; i < tamanho; i++) {
        int selecionado = v[i];
        int j;
        for (j = i; j > 0 && v[j - 1] > selecionado ; j--) {
            v[j] = v[j - 1];
        }
        v[j] = selecionado ;
    }
}
```

O Insertion Sort aplica várias vezes a inserção ordenada para ordenar uma sequência. O Insertion Sort in-place é porque a ordenação é feita rearranjando os elementos no próprio array, ao invés de usar arrays ou outras estruturas auxiliares.

Detalhando o InsertionSort no vetor b

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

No 1º for i=1 , então Selecionado = v[i] = 6
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor.

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 1	j= 1	1>0 && 7>6	[7, 7, 5, 4, 3, 3, 2, 1]
i= 1	j= 0	0 > 0	[7, 7, 5, 4, 3, 3, 2, 1]

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 6 e o ao fim da 1º varredura o vetor fica [6, 7, 5, 4, 3, 3, 2, 1]

No 1º for i=2 , então Selecionado = v[i] = 5
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 2	j= 2	2>0 && 7>5	[6, 7, 7, 4, 3, 3, 2, 1]
i= 2	j= 1	1>0 && 6>5	[6, 6, 5, 4, 3, 3, 2, 1]
i= 2	j=0	0 > 0	[6, 6, 5, 4, 3, 3, 2, 1]

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 5 e ao fim da 2º varredura o vetor fica [5, 6, 7, 4, 3, 3, 2, 1]

No 1º for i=3 , então Selecionado = v[i] = 4
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 3	j= 3	3>0 && 7>4	[5, 6, 7, 7, 3, 3, 2, 1]
i= 3	j= 2	2>0 && 6>4	[5, 6, 6, 7, 3, 3, 2, 1]
i= 3	j= 1	1>0 && 5>4	[5, 5, 6, 7, 3, 3, 2, 1]
i= 3	j= 0	0 > 0	[5, 5, 6, 7, 3, 3, 2, 1]

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 4 e ao fim da 3º varredura o vetor fica [4, 5, 6, 7, 3, 3, 2, 1]

No 1º for i=4 , então Selecionado = v[i] = 3
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 4	j= 4	4>0 && 7>3	[4, 5, 6, 7, 7, 3, 2, 1]
i= 4	j= 3	3>0 && 6>3	[4, 5, 6, 6, 7, 3, 2, 1]
i= 4	j= 2	2>0 && 5>3	[4, 5, 5, 6, 7, 3, 2, 1]
i= 4	j= 1	1>0 && 4>3	[4, 4, 5, 6, 7, 3, 2, 1]
i= 4	j= 0	0 > 0	[4, 4, 5, 6, 7, 3, 2, 1

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 3 e ao fim da 4º varredura o vetor fica [3, 4, 5, 6, 7, 3, 2, 1]

No 1º for i=5 , então Selecionado = v[i] = 3
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 5	j = 5	5> 0 && 7>3	[3, 4, 5, 6, 7, 7, 2, 1]
i= 5	j= 4	4>0 && 6>3	[3, 4, 5, 6, 6, 7, 2, 1]
i= 5	j= 3	3>0 && 5>3	[3, 4, 5, 5, 6, 7, 2, 1]
i= 5	j= 2	2>0 && 4>3	[3, 4, 4, 5, 6, 7, 2, 1]
i= 5	j= 1	1>0 && 3>3	[3, 4, 4, 5, 6, 7, 2, 1]

Quando 3>3 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 1. Então v[1] = 3 e ao fim da 5º varredura o vetor fica [3, 3, 4, 5, 6, 7, 2, 1]

No 1º for i=6 , então Selecionado = v[i] = 2
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 6	j = 6	6 > 0 && 7>2	[3, 3, 4, 5, 6, 7, 7, 1]
i= 6	j = 5	5> 0 && 6>2	[3, 3, 4, 5, 6, 6, 7, 1]
i= 6	j= 4	4>0 && 5>2	[3, 3, 4, 5, 5, 6, 7, 1]
i= 6	j= 3	3>0 && 4>2	[3, 3, 4, 4, 5, 6, 7, 1]
i= 6	j= 2	2>0 && 3>2	[3, 3, 3, 4, 5, 6, 7, 1]
i= 6	j= 1	1>0 && 3>2	[3, 3, 3, 4, 5, 6, 7, 1]
i= 6	j = 0	0 > 0	[3, 3, 3, 4, 5, 6, 7, 1]

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 2 e ao fim da 6º varredura o vetor fica [2, 3, 3, 4, 5, 6, 7, 1]

No 1º for i=7 , então Selecionado = v[i] = 1
Se a condição do 2º "for" for aceita, o v[j] recebe o valor de seu antecessor

Valor de i	Valor de j	j>0 && j-1 > selecionado	Vetor
i= 7	j = 7	7 > 0 && 7>1	[2, 3, 3, 4, 5, 6, 7, 7]
i= 7	j = 6	6> 0 && 6>1	[2, 3, 3, 4, 5, 6, 6, 7]
i= 7	j= 5	5>0 && 5>1	[2, 3, 3, 4, 5, 5, 6, 7]
i= 7	j= 4	4>0 && 4>1	[2, 3, 3, 4, 4, 5, 6, 7]
i= 7	j= 3	3>0 && 3>1	[2, 3, 3, 3, 4, 5, 6, 7]
i= 7	j= 2	2>0 && 3>1	[2, 3, 3, 3, 4, 5, 6, 7]
i= 7	j = 1	1 > 0 && 2>1	[2, 2, 3, 3, 4, 5, 6, 7]
i= 7	j = 0	0 > 0	[2, 2, 3, 3, 4, 5, 6, 7]

Quando j=0 a condição não é satisfeita e saímos do 2º for.

Após a varredura e sair do "for" o v[j] recebe o selecionado, como j foi decrementado seu valor é 0. Então v[0] = 1 e ao fim da 6º varredura o vetor fica [1, 2, 3, 3, 4, 5, 6, 7]

d. MergeSort com o vetor a

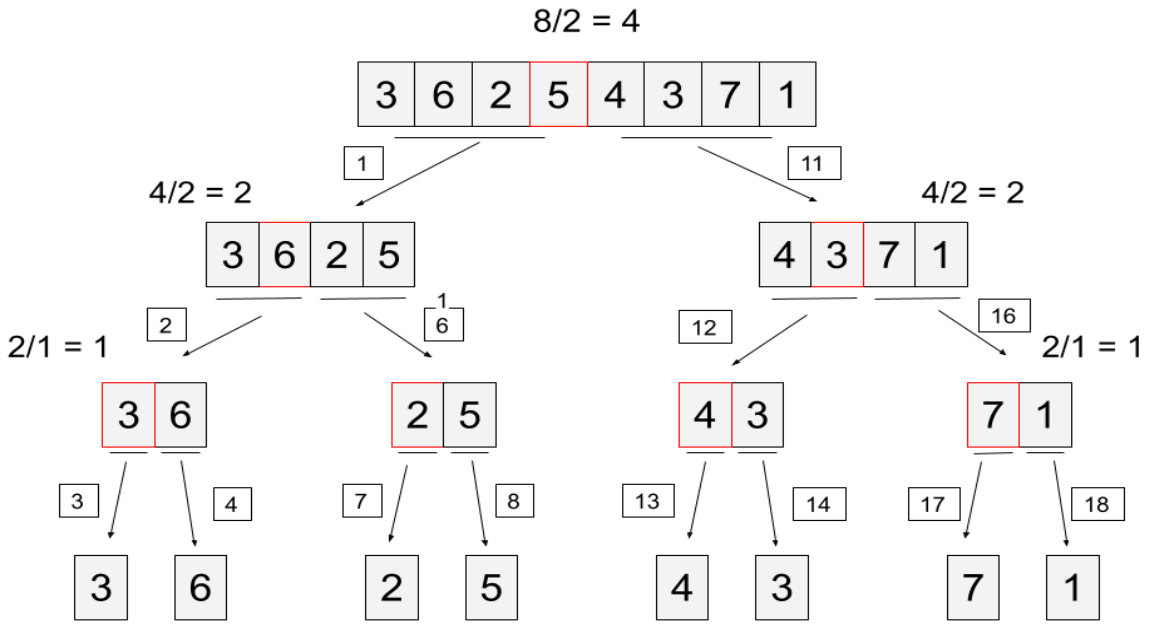
```c
void mergeSort(int* V, int tamV) {
    if (tamV > 1) {
        int metade = tamV / 2;

        int tamV1 = metade;
        int* V1 = (int*)malloc(tamV1 * sizeof(int));
        for (int i = 0; i < metade; i++) {
            V1[i] = V[i];
        }

        int tamV2 = tamV - metade;
        int* V2 = (int*)malloc(tamV2 * sizeof(int));
        for (int i = metade; i < tamV; i++) {
            V2[i - metade] = V[i];
        }

        mergeSort(V1, tamV1);
        mergeSort(V2, tamV2);
        UnirEeD(V, tamV, V1, tamV1, V2, tamV2);

        free(V1);
        free(V2);
    }
}
```

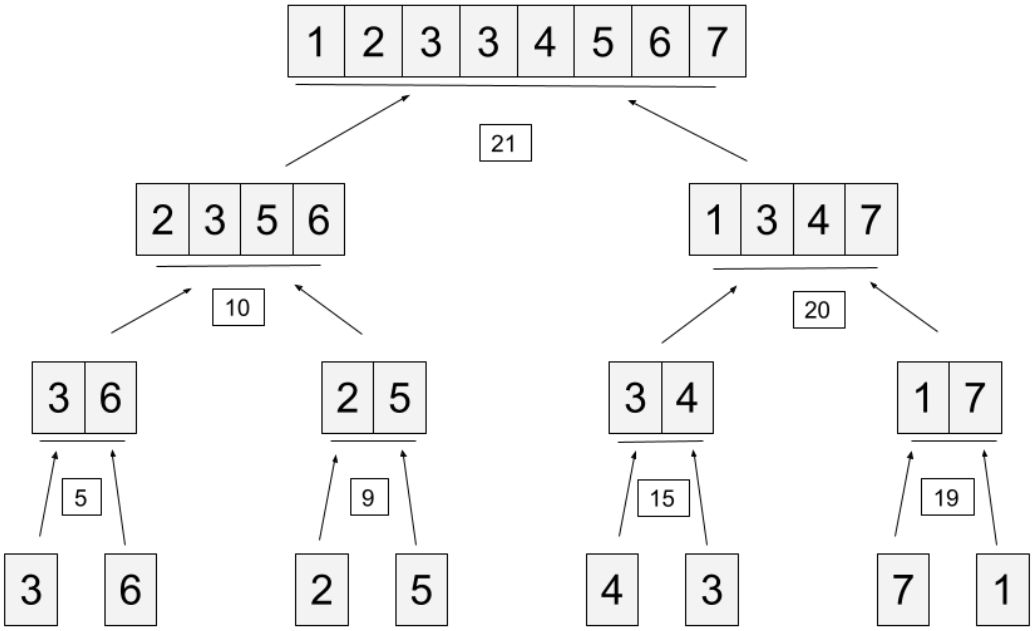
```c
void UnirEeD(int* V, int tamV, int* e, int tamE, int* d, int tamD) {
    int posicaoV = 0;
    int posicaoE = 0;
    int posicaoD = 0;
    while (posicaoE < tamE && posicaoD < tamD) {
        if (e[posicaoE] <= d[posicaoD]) {
            V[posicaoV] = e[posicaoE];
            posicaoE++;
        }
        else {
            V[posicaoV] = d[posicaoD];
            posicaoD++;
        }
        posicaoV++;
    }

    while (posicaoE < tamE) {
        V[posicaoV] = e[posicaoE];
        posicaoE++;
        posicaoV++;
    }
    while (posicaoD < tamD) {
        V[posicaoV] = d[posicaoD];
        posicaoD++;
        posicaoV++;
    }
}
```

Merge Sort é um algoritmo eficiente de ordenação por divisão e conquista, ou seja, temos que fazer o processo de divisão que consiste em dividir o array recursivamente na metade até que sobre apenas um elemento e após isso vem o processo de ordenação que é a rotina de combinar os arrays até que consiga unir todo o array novamente.

Detalhando o MergeSort no vetor a

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





Após o uso das funções fazendo com que todo o array se torne um array unitário, começando a dividir sempre a metade esquerda de cada array, o UnirEeD fará com que esses array unitários vão se juntando em ordenação. Os quadrados menores indicam a sequência de funcionamento do mergeSort com base na separação e união

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

```
int particiona(int* v, int ini, int fim) {
    int Pselecionado = ini;
    int pivo = v[fim];
    for (int i = ini; i < fim; i++) {
        if (v[i] <= pivo) {

            int temp = v[i];
            v[i] = v[Pselecionado];
            v[Pselecionado] = temp;

            Pselecionado++;
        }
    }

    int temp = v[fim];
    v[fim] = v[Pselecionado];
    v[Pselecionado] = temp;

    return Pselecionado;

}

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

O funcionamento do QuickSort baseia-se em uma rotina fundamental cujo nome é particionamento. Particionar significa escolher um número qualquer presente no array, chamado de pivô, e colocá-lo em uma posição tal que todos os elementos à esquerda são menores ou iguais e todos os elementos à direita são maiores. Escolheremos o ultimo elemento do nosso vetor para ser nosso pivô.

Detalhando o InsertionSort no vetor b

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

Se a condição for aceita, fazemos uma troca do valor que esta no Pselecionado com os que estão na posição i e o valor de Pselecionado é incrementado

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
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]

Após a saída do for, trocamos nosso pivô (ultimo elemento do array) com o valor que está no Pselecionado usando um auxiliador:

int temp = v[fim] // pivô // 1
v[fim] = v[Pselecionado] // v[7] = v[0] // v[7] recebeu o valor de v[0] // v[7] = 7
v[Pselecionado] = temp; // v[0] = 1

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

Para continuar com a ordenação, temos duas chamadas quickSort(v, ini, posicaoPivo - 1) e quickSort(v, posicaoPivo + 1, fim). Como nosso pivô ficou v[0], a chamada executada sera: quickSort(v, posicaoPivo + 1, fim) , logo nosso Pselecionado começa em 1.

Se a condição for aceita, fazemos uma troca do valor que esta no Pselecionado com os que estão na posição i e o valor de Pselecionado é incrementado

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
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]

Como a ideia do quicksort é organizar os valores maiores que o pivô de um lado e os menores do outro, sendo o elemento da última posição o maior do vetor, não haverá nenhuma troca, visto que os menores já estão a sua esquerda e após sair do for, também não haverá mudança já que o Pselecionado é igual ao fim.

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

Para continuar com a ordenação, temos duas camadas quickSort(v, ini, posicaoPivo - 1) e quickSort(v, posicaoPivo + 1, fim). Como nosso pivô ficou v[7], a camada executada sera:
(v, ini, posicaoPivo - 1) logo nosso Pselecionado começa em 1.

Se a condição for aceita, fazemos uma troca do valor que esta no Pselecionado com os que estão na posição i e o valor de Pselecionado é incrementado

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
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,]

int temp = v[fim] // pivô // 2
v[fim] = v[Pselecionado] // v[6] = v[1] // v[6] recebeu o valor de v[1] // v[6] = 6
v[Pselecionado] = temp; // v[1] = 2

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

Para continuar com a ordenação, temos duas chamadas quickSort(v, ini, posicaoPivo - 1) e quickSort(v, posicaoPivo + 1, fim). Como nosso pivô ficou v[1], a chamada executada será:
(v, i posicaoPivo + 1, fim). Logo nosso Pselecionado começa em 2.

Se a condição for aceita, fazemos uma troca do valor que esta no Pselecionado com os que estão na posição i e o valor de Pselecionado é incrementado

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
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]

Como a ideia do quicksort é organizar os valores maiores que o pivô de um lado e os menores do outro, sendo o elemento da última posição o maior do vetor, não haverá nenhuma troca, visto que os menores já estão a sua esquerda e saindo do for, também não haverá mudança já que o Pselecionado é igual ao fim.

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

Para continuar com a ordenação, temos duas chamadas quickSort(v, ini, posicaoPivo - 1) e quickSort(v, posicaoPivo + 1, fim). Como nosso pivô ficou v[6], a chamada executada será:
(v, ini, posicaoPivo - 1) logo nosso Pselecionado começa em 2.

Se a condição for aceita, fazemos uma troca do valor que esta no Pselecionado com os que estão na posição i e o valor de Pselecionado é incrementado

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
2	2	3	5<=3	[5, 4, 3, 3]
2	3	3	4<=3	[5, 4, 3, 3]
2	4	3	3<=3	[3, 4, 5, 3]

Pselecionado = [3], pois a condição na ultima linha da tabela foi aceita.

int temp = v[fim] // pivô // 3
v[fim] = v[Pselecionado] // v[5] = v[3] // v[5] recebeu o valor de v[3] // v[5] = 4
v[Pselecionado] = temp;; // v[3] = 3

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

Para continuar com a ordenação, temos duas chamadas quickSort(v, ini, posicaoPivo - 1) e quickSort(v, indexPivo + 1, fim). Como nosso pivô ficou v[3], a chamada (v, ini, posicaoPivo- 1) teria apenas um array unitário, logo já esta ordenado e para a chamada a (v, posicaoPivo + 1, fim), temos:

Pselecionado (índice)	Valor de i	Pivô	v[i] <= pivô	Vetor
4	4	4	5<=4	[5, 4]

int temp = fim // pivô // 4
v[fim] = v[Pselecionado] // v[5] = v[4] // v[5] recebeu o valor de v[4] // v[5] = 5
v[Pselecionado] = temp; // v[4] = 4

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

Para continuar com a ordenação, temos duas chamadas quickSort(v, ini, posicaoPivo- 1) e quickSort(v, posicaoPivo+ 1, fim). Como nosso pivô ficou v[4], a chamada (v, posicaoPivo + 1, fim) teria apenas um array unitário, logo já está ordenado.

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

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

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

Criamos uma função que serve de auxiliador para a troca de valores de variáveis

```
int pivorandomizado(int* v, int ini, int fim) {

    int posicaodopivo = ini + rand() % (fim - ini + 1);

        troca(v, posicaodopivo, fim);

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

    troca(v, Pselecionado, fim);
    return Pselecionado;

}

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

Usamos a função random para poder gerar um numero aleatório entre inicio e fim do vetor. Assim teremos a implementação do pivô randomizado, já que o pivô será escolhido de forma aleatória.

```
Running main() from c:\a\1\s\thirdparty\googletest\googletest\src\gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SortingAlgorithmsTest
[ RUN      ] SortingAlgorithmsTest.QuickSort
[       OK ] SortingAlgorithmsTest.QuickSort (1 ms)
[----------] 1 test from SortingAlgorithmsTest (1 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (51 ms total)
[  PASSED  ] 1 test.

O C:\Users\Pichau\Desktop\algortimos e estrutura de dados\prova2.1\Debug\prova2.1.exe (processo 6592) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

E usamos o teste disponibilizado para verificar se a implementação estava correta

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)**

Tabela referente ao tamanho de 10^1

Algoritmo	Tipo de Vetor	Media da execução (ms)	Mediana de execução(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	Ordenado	0.00	0.00
InsertionSort	Aleatório	0.00	0.00
InsertionSort	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 c/ pivô randomizado	Ordenado	0.00	0.00
QuickSort c/pivô randomizado	Aleatório	0.00	0.00
QuickSort c/pivô randomizado	Decrescente	0.00	0.00
CountingSort	Ordenado	0.00	0.00
CountingSort	Aleatório	0.00	0.00
CountingSort	Decrescente	0.00	0.00

Nessa 1º parte podemos identificar que devido ao tamanho do vetor ser relativamente pequeno, as funções têm um bom e equivalente desempenho independente de como os vetores estejam ordenados.

Tabela referente ao tamanho de 10^3

Algoritmo	Tipo de Vetor	Media da execução(ms)	Mediana de execução(ms)
SelectionSort (in-place)	Ordenado	1.68	1.56
SelectionSort (in-place)	Aleatório	2.83	2.27
SelectionSort (in-place)	Decrescente	1.30	1.25
BubbleSort	Ordenado	0.00	0.00
BubbleSort	Aleatório	2.42	2.27
BubbleSort	Decrescente	2.81	2.52
InsertionSort	Ordenado	0.00	0.00
InsertionSort	Aleatório	1.20	1.12
InsertionSort	Decrescente	1.47	1.27
MergeSort	Ordenado	076	0.73

MergeSort	Aleatório	0.78	0.77
MergeSort	Decrescente	1.2	0.81
QuickSort	Ordenado	3.13	2.45
QuickSort	Aleatório	0.31	0.40
QuickSort	Decrescente	3.79	3.26
QuickSort c/ pivô randomizado	Ordenado	0.13	0.12
QuickSort c/pivô randomizado	Aleatório	0.16	0.16
QuickSort c/pivô randomizado	Decrescente	0.14	0.13
CountingSort	Ordenado	0.02	0.02
CountingSort	Aleatório	0.07	0.05
CountingSort	Decrescente	0.06	0.08

Nessa 2º parte já conseguimos identificar que algumas funções já começam a apresentar um certo nível de atraso dependendo da ordenação do vetor. Destacando o selectionsort, bubblesort, quicksort e um pouco do insertionsort que apresentaram uma media e mediana de tempo maior que as demais funções. Especialmente o quicksort quando está ordenado e ordenado de forma decrescente.

Tabela referente ao tamanho de 10^5

Algoritmo	Tipo de Vetor	Media da execução (ms)	Mediana de execução (ms)
SelectionSort (in-place)	Ordenado	11898.65	11837.18
SelectionSort (in-place)	Aleatório	11873.50	11824.44
SelectionSort (in-place)	Decrescente	12281.48	12221.46
BubbleSort	Ordenado	0.35	0.29
BubbleSort	Aleatório	34641.54	34565.19

BubbleSort	Decrescente	25622.19	26457.32
InsertionSort	Ordenado	0.43	0.40
InsertionSort	Aleatório	6676.98	6602.34
InsertionSort	Decrescente	13212.09	13151.37
MergeSort	Ordenado	111.1	103.6
MergeSort	Aleatório	116.00	105.313
MergeSort	Decrescente	110.65	105.404
QuickSort	Ordenado	O teste falhou	O teste falhou
QuickSort	Aleatório	24.11	22.84
QuickSort	Decrescente	O teste falhou	O teste falhou
QuickSort c/ pivô randomizado	Ordenado	25.30	23.67
QuickSort c/pivô randomizado	Aleatório	34.14	32.39
QuickSort c/pivô randomizado	Decrescente	30.30	28.46
CountingSort	Ordenado	3.26	2.58
CountingSort	Aleatório	5.25	3.83
CountingSort	Decrescente	2.17	1.97

Nesta 3º parte conseguimos de fato identificar a diferença de desempenho das funções. Algumas que apresentaram valor altíssimo e outras, como o quicksort que não conseguiu executar no experimento do teste. Ressaltando também a função coutingsort que foi a melhor de todas, tendo seu melhor desempenho quando está ordenada de forma decrescente e no pior caso de forma aleatória.

No geral temos:

Selection sort: o selection sort teve um desempenho cada vez pior com o aumento do tamanho do vetor, isso se dá pelo fato dele ser O(n^2), sendo assim conforme o vetor for aumentando, o desempenho vai piorando. Mostrou-se de fato ser uma das funções de ordenação mais simples.

Bubblesort: o bubblesort ainda teve um desempenho pior em seus piores casos, em que o vetor esta ordenado de forma aleatória e decrescente em comparação com o selectionsort, porém apresentou um bom desempenho no seu melhor caso. Essa função também é uma das mais simples e apresenta O(n^2) para seu pior caso e O(n) no melhor.

Insertiosort: o insertiosort apresenta o mesmo caminho do bubblesort para seu melhor e pior caso, O(n) e O(n^2) respectivamente. Tem seu melhor desempenho quando já está ordenado e seu pior caso quando este ordenado de forma decrescente e de forma aleatória.

Mergesort: o mergesort já apresenta uma boa melhoria nos desempenhos em comparação aos anteriores, sendo um algoritmo mais eficiente de O(n*log*n) para qualquer situação, seja ordenado, ordenada de forma aleatória ou de forma decrescente.

Quicksort: o quicksort pode ser avaliado de 2 formas, com ou sem o pivô randomizado. No primeiro caso, sem o pivô randomizado ele apresenta falhas em duas formas de ordenação, que é a decrescente e ordenado, sendo O(n^2) nesses casos, não conseguimos determinar o seu tempo para execução na ordenação. Porém na ordenação aleatória já possui um melhor desempenho, sendo O(n*log*n), isso se dá pelo fato que de seu pivô ser escolhido de forma aleatória. E uma forma de garantir isso é fazendo com que o pivô seja randomizado, isso evita o desempenho O(n^2) e tem um desempenho O(n*log*n). Como podemos ver no experimento, o quicksort com o pivô randomizado teve resultados de média e mediana bem melhor, resolvendo o problema do Quicksort inicial.

Coutingsort: O coutingsort entre as funções de ordenação foi sem duvidas o que apresentou o melhor desempenho, diferentes de todos os anteriores, apresentou uma complexidade de O(n) para diferentes tipos de ordenação nas 3 situações, mas é importante entender que o algoritmo tem seu tempo de execução linear em função do tamanho de n e k, ou seja, sua complexidade para pior e melhor caso seria exatamente O (n+k) e não somente do tamanho de n. Esse tempo de execução é substancialmente mais eficiente do que os outros algoritmos que vimos. Contudo, esse algoritmo também tem um custo associado ao uso de memória maior. E por mais que tenha sido o melhor em desempenho, o coutingsort ainda tem o fato em que cada vez que o intervalo entre seus números for maior, seu desempenho piora e usa mais processamento.