UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

ALGORITMOS E ESTRUTURAS DE DADOS I

ALGORITMOS DE ORDENAÇÃO

Avaliação da Unidade II

Igor Michael de Araujo de Macedo

Natal, 10 de agosto de 2021

PARTE 1

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:

- *BubbleSort* (melhor versão) com o vetor a;
- *InsertionSort* (*in-place*, melhor versão) com o vetor b;
- *MergeSort* com o vetor a;
- *QuickSort* (sem randomização de pivô) com o vetor b.

BubbleSort

A ideia do *BubbleSort* é varrer o *array* $n - 1$ vezes, sendo n a quantidade de elementos, e a cada iteração levar o maior valor à posição final. Na primeira iteração o vetor será percorrido do primeiro elemento até o último e será levado o maior valor encontrado para a última posição; na segunda iteração será percorrido do primeiro elemento até o penúltimo e levado o maior elemento encontrado à penúltima posição; e assim por diante até todos elementos estarem ordenados. Para levar o maior valor à última posição, a cada iteração, a ideia é comparar todos os os elementos, dois a dois, deixando sempre o maior entre os dois à direita, sendo assim, fazendo a troca quando necessária. Por fim, uma estratégia de otimização pode ser utilizada: caso seja feita uma varredura em todo o *array* e não seja feita nenhuma troca, significa que ele já está ordenado, então, para isso, utiliza-se uma *flag* booleana para indicar se houve ou não troca.

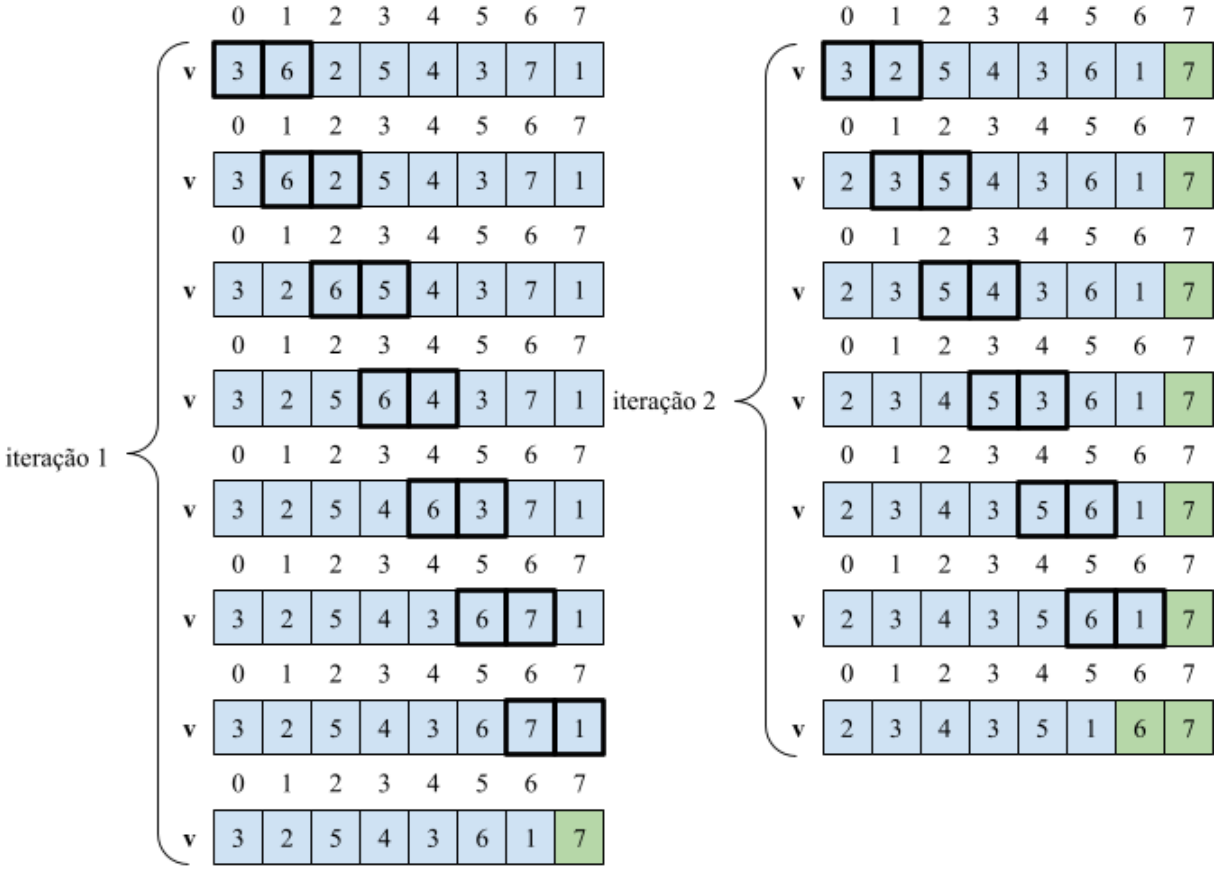
Na Figura 1 é possível ver o código-fonte desenvolvido para o *BubbleSort* e nas Figuras 2, 3 e 4 ilustrações das iterações feitas para o algoritmo rodando no vetor a.

Figura 1 - Código-fonte do *BubbleSort*.

```
6   void bubbleSort(int* v, int n) {
7       for (int j = 0; j < n-1; j++){
8           bool trocou = false;
9
10          for (int i=0; i < n-j-1; i++) {
11              if (v[i] > v[i+1]) {
12                  int temp = v[i];
13                  v[i] = v[i+1];
14                  v[i+1] = temp;
15                  trocou = true;
16              }
17          }
18
19          if (!trocou) return;
20      }
21  }
```

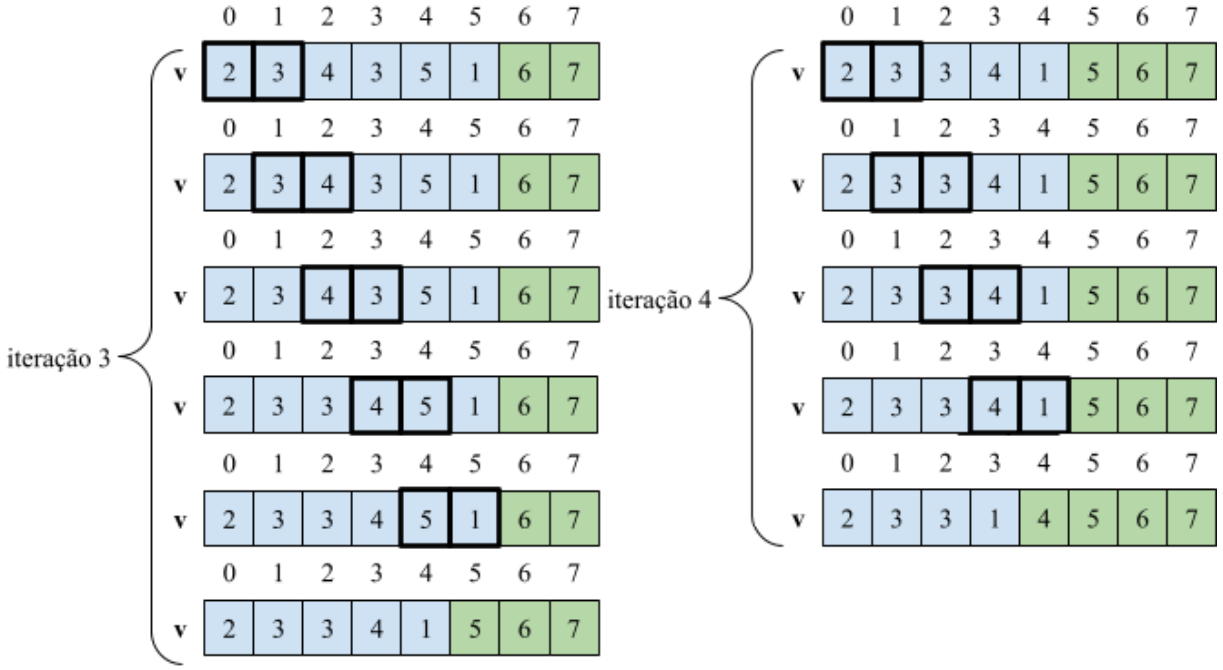
Fonte: Elaboração própria.

Figura 2 - Iterações 1 e 2 do *BubbleSort*.



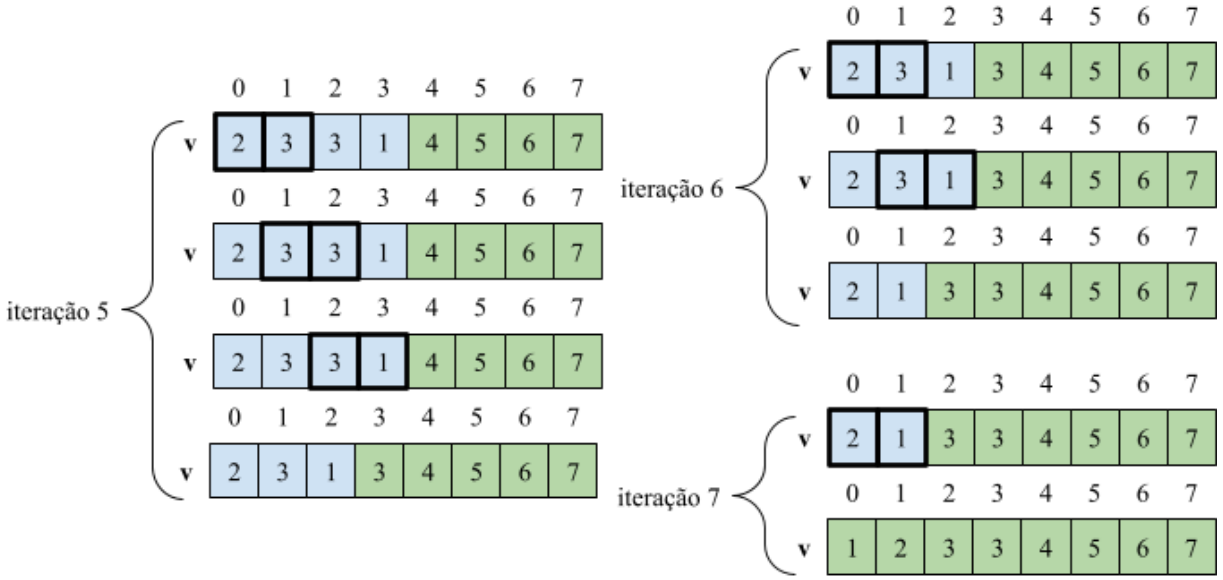
Fonte: Elaboração própria.

Figura 3 - Iterações 3 e 4 do *BubbleSort*.



Fonte: Elaboração própria.

Figura 4 - Iterações 5, 6 e 7 do *BubbleSort*.



Fonte: Elaboração própria.

InsertionSort

A ideia do *InsertionSort* é dividir o *array* em dois, parte direita (desordenada) e esquerda (ordenada). Inicialmente a parte esquerda começará com um elemento, o da primeira posição do *array* e, em seguida, serão adicionados um a um, da mão direita para a mão esquerda, de forma ordenada. O algoritmo pode ser desenvolvido de forma *out-of-place*, ou seja, alocando *arrays* extras para fazer essa divisão, ou de forma *in-place*, usando apenas o

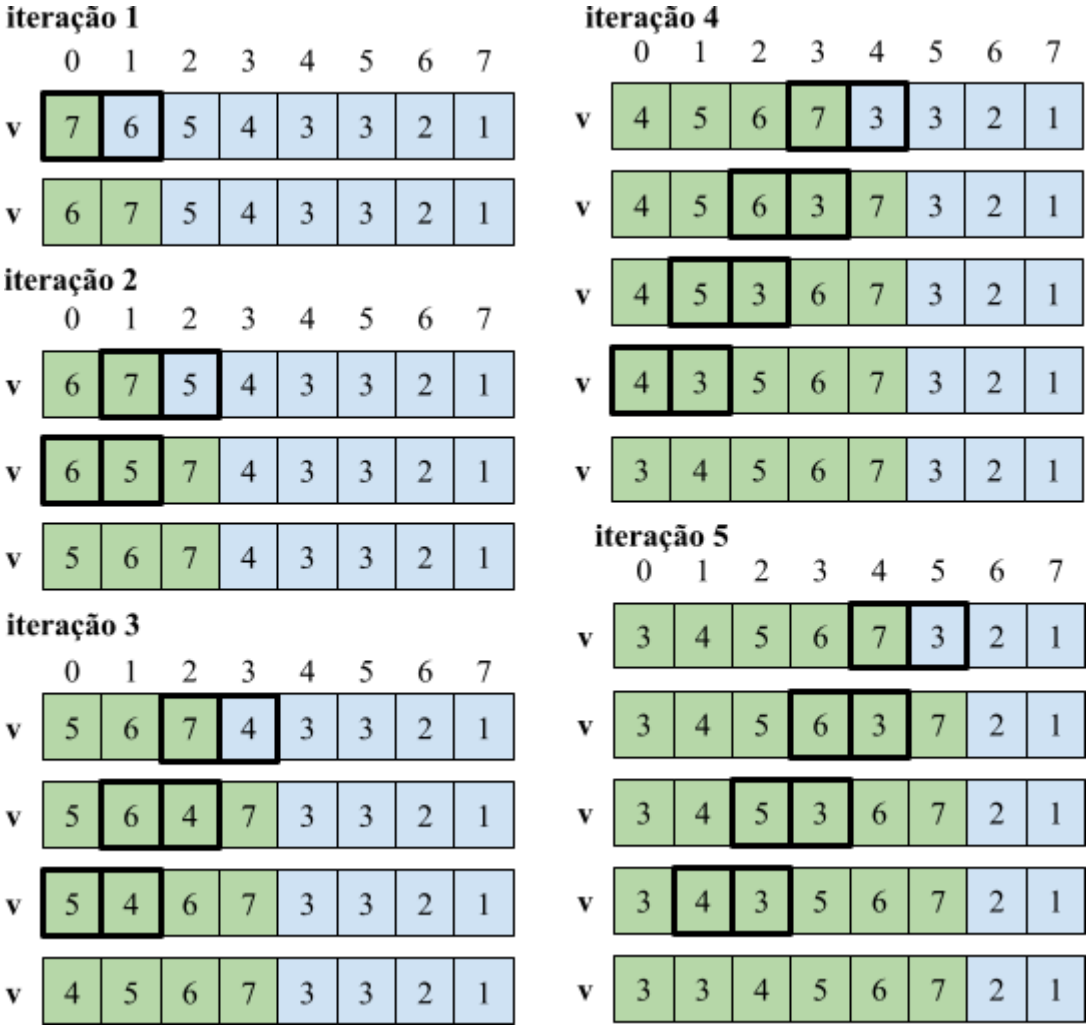
mesmo (que será usado nas ilustrações a seguir). Na Figura 5 é possível ver o código-fonte desenvolvido para o *InsertionSort*. Nas Figuras 6 e 7 ilustrações do algoritmo para o vetor b.

Figura 5 - Código-fonte do *InsertionSort*.

```
6  void insertionSortIP(int* v, int tamanho) {
7      for (int i=1; i < tamanho; i++) {
8          int valor = v[i];
9          int j;
10         for (j=i; j > 0 && v[j-1 ] > valor; j--) {
11             v[j] = v[j-1];
12         }
13         v[j] = valor;
14     }
15 }
```

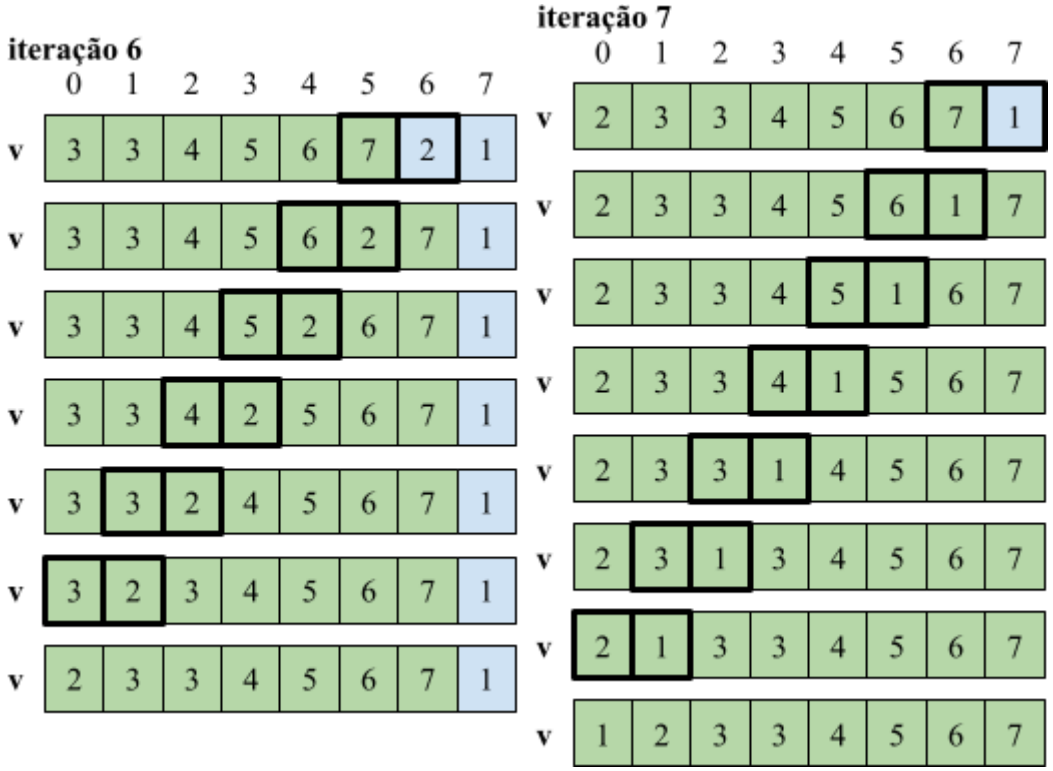
Fonte: Elaboração própria.

Figura 6 - Iterações 1, 2, 3, 4 e 5 do *InsertionSort.*



Fonte: Elaboração própria.

Figura 7 - Iterações 6 e 7 do *InsertionSort.*



Fonte: Elaboração própria.

MergeSort

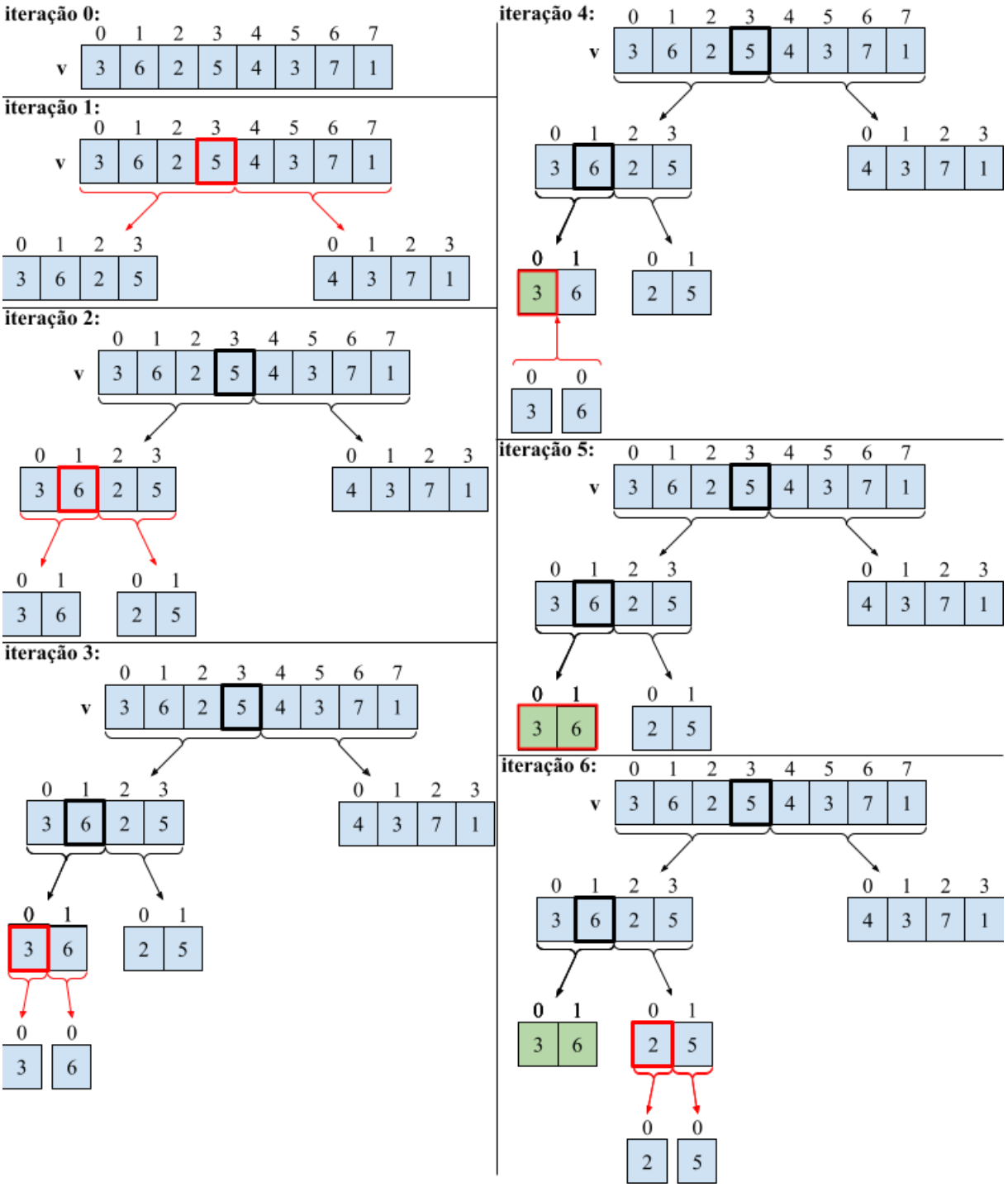
O *MergeSort*, basicamente, utiliza a estratégia “dividir para conquistar”; sendo assim, o *array* será dividido o máximo de vezes possíveis e, quando chegar no máximo, será remontado/fundido ordenadamente até voltar ao tamanho original com todos os elementos. Dessa forma, o algoritmo desenvolvido é composto por duas partes principais: a responsável por fazer essa junção/*merge;* e a parte responsável por fazer a divisão do array, de forma recursiva, e chamar o *merge* no final, como pode ser visto na Figura 8. Por fim, nas Figuras 9, 10, 11, 12 e 13 podem ser vistas as ilustrações de todas as iterações deste algoritmo aplicadas ao vetor a.

Figura 8 - Código-fonte da primeira parte do *MergeSort*.

```
4   void merge(int *v, int tamV, int *e, int tamE, int *d, int tamD){
5       int indexV = 0, indexE = 0, indexD = 0;
6       while(indexE < tamE && indexD < tamD){
7           if(e[indexE] <= d[indexD]) {
8               v[indexV] = e[indexE];
9               indexE++;
10          } else {
11              v[indexV] = d[indexD];
12              indexD++;
13          }
14          indexV++;
15      }
16      while(indexE < tamE){
17          v[indexV] = e[indexE];
18          indexE++;
19          indexV++;
20      }
21      while(indexD < tamD){
22          v[indexV] = d[indexD];
23          indexD++;
24          indexV++;
25      }
26  }
27
28  void mergeSort(int *v, int tamV){
29      if(tamV > 1) {
30          int meio = tamV/2, tamE = meio, tamD = tamV - meio;
31          int *e = (int*)malloc(tamE*sizeof(int));
32          int *d = (int*)malloc(tamD*sizeof(int));
33          for (int i = 0; i < tamD; i++){
34              if ((tamV % 2 != 0) && i == tamD) {
35                  d[i] = v[i + meio];
36                  break;
37              }
38              e[i] = v[i];
39              d[i] = v[i + meio];
40          }
41          mergeSort(e, tamE);
42          mergeSort(d, tamD);
43          merge(v, tamV, e, tamE, d, tamD);
44          free(e);
45          free(d);
46      }
47  }
```

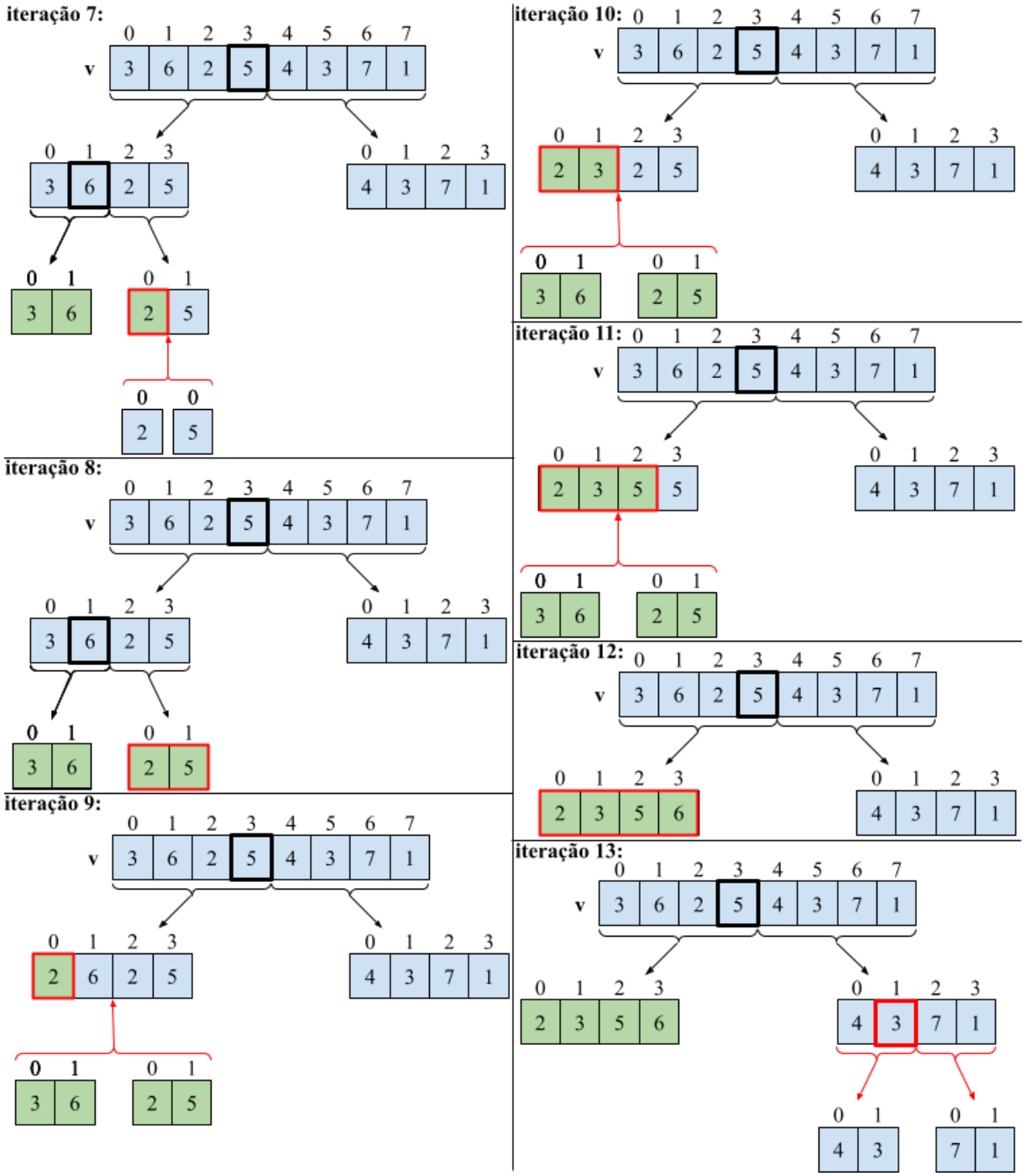
Fonte: Elaboração própria.

Figura 9 - Algumas iterações do *MergeSort*.



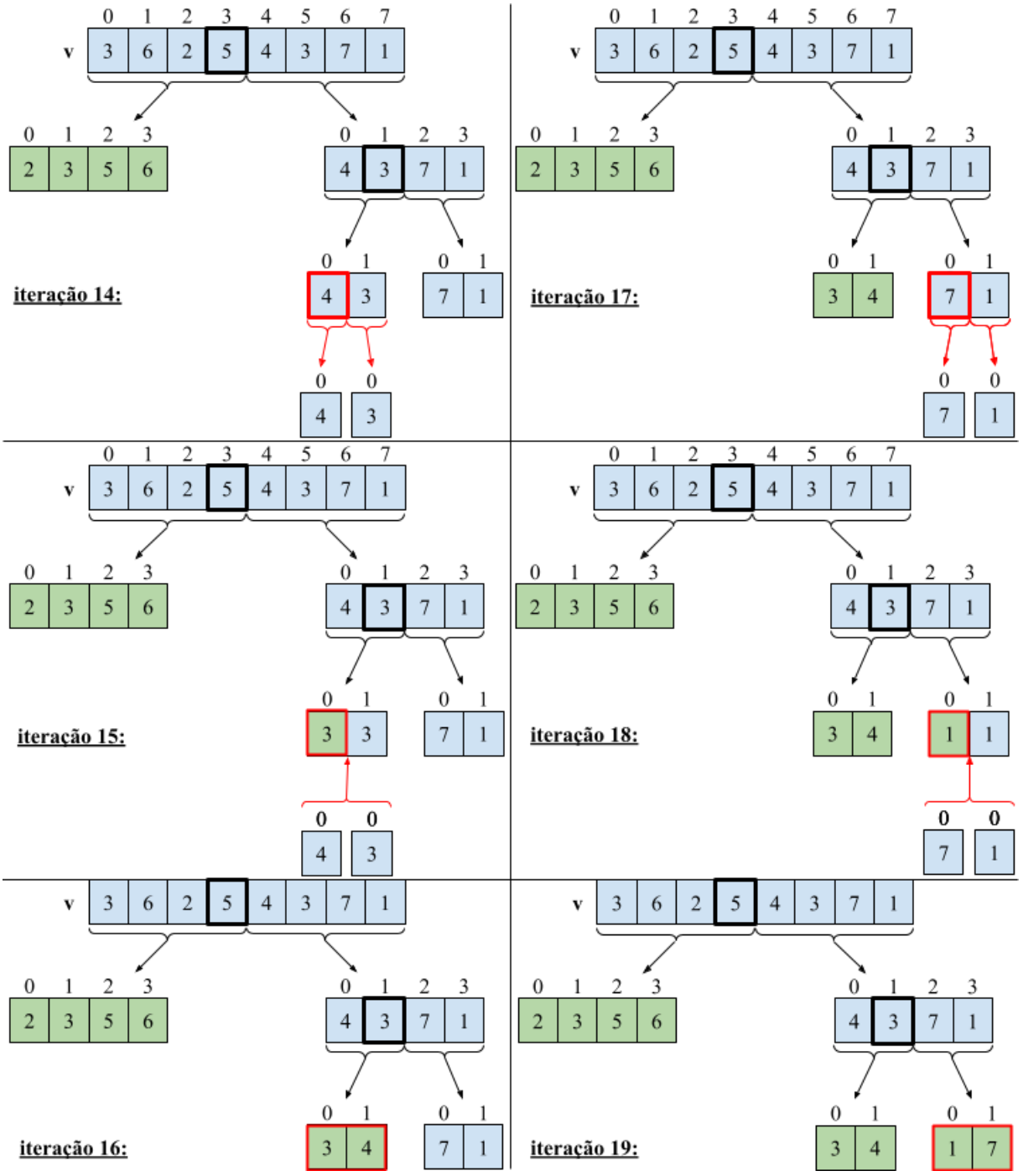
Fonte: Elaboração própria.

Figura 10 - Algumas iterações do *MergeSort*.



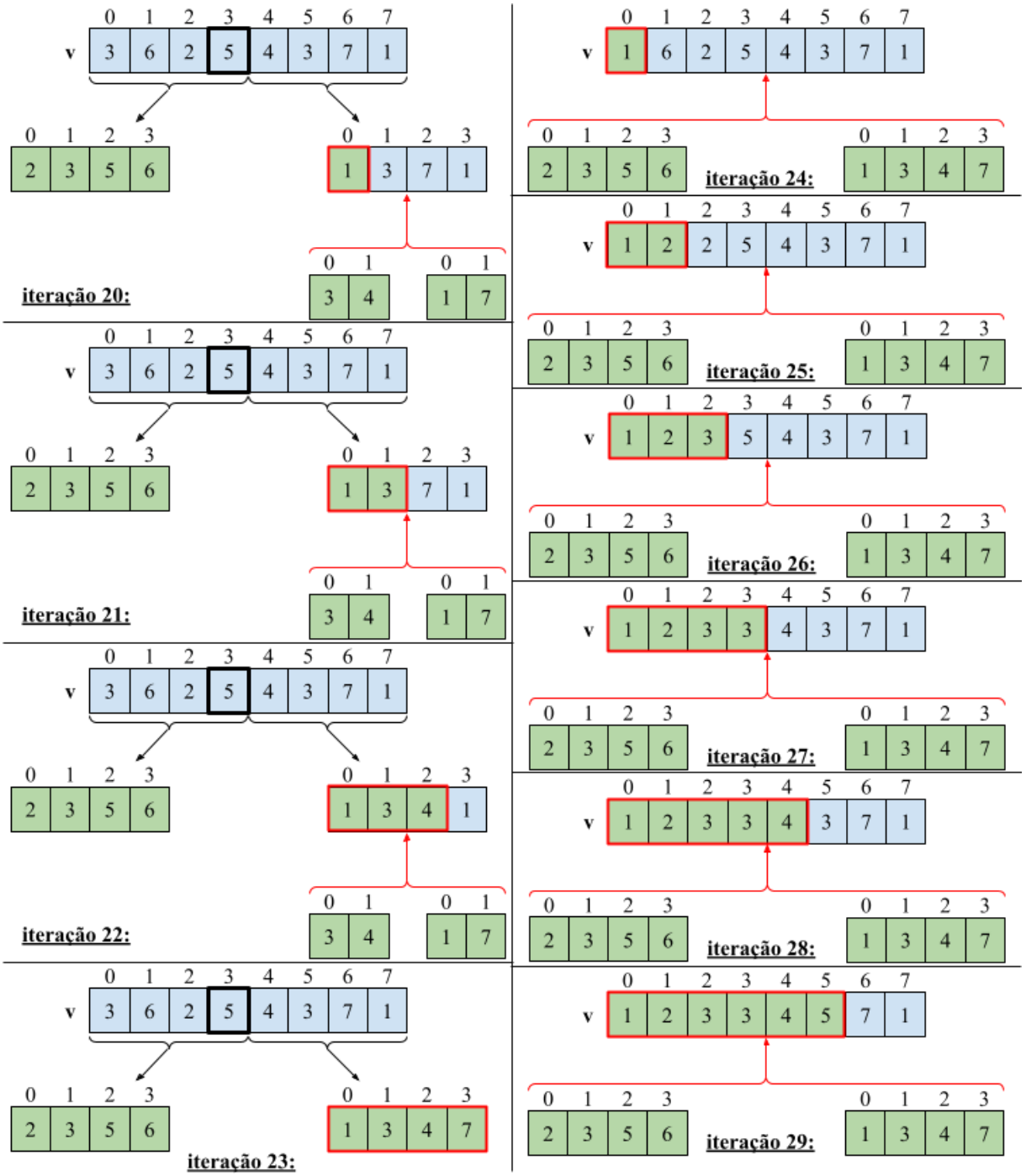
Fonte: Elaboração própria.

Figura 11 - Algumas iterações do *MergeSort*.



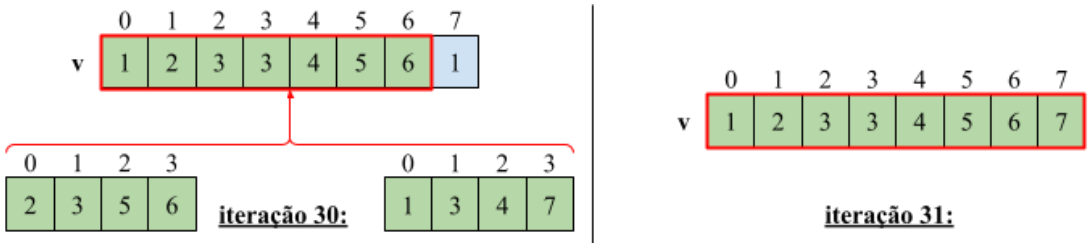
Fonte: Elaboração própria.

Figura 12 - Algumas iterações do *MergeSort*.



Fonte: Elaboração própria.

Figura 13 - Algumas iterações do *MergeSort*.



Fonte: Elaboração própria.

QuickSort

Basicamente, a ideia do *QuickSort* é, a cada iteração, escolher um pivô (escolhendo sempre o último elemento do *array* ou *sub-array*, ou então escolhendo uma posição aleatória e movendo-a para o fim), colocar todos os elementos menores que ele à sua esquerda e os maiores à sua direita. Ao fazer isso, o pivô estará na posição que deveria e existirão dois *sub-arrays*, um à direita e outro à esquerda, com valores maiores e menores, respectivamente, ao pivô (essa divisão será chamada de particionamento). Em seguida, na próxima iteração, escolhe-se um novo pivô e aplica o particionamento para cada *sub-array*. Será aplicado o particionamento em todos os *sub-arrays* até todos existirem apenas os pivôs, que já estarão nas posições corretas.

Na Figura 14 é possível ver o código fonte para o *QuickSort* sem randomização de pivô e, nas Figuras 16, 17, 18, 19 e 20 há ilustrações do processo do algoritmo aplicado ao vetor b. Antes disso, na Figura 15, há uma legenda para diferenciar:

- qual elemento é o pivô (fundo laranja) da iteração;
- qual é o índice que o pivô ficará no final (borda laranja), ou seja, ao fim da iteração essa será a posição real no *array* e que separa em dois *sub-array* com elementos menores e maiores que o pivô;
- por fim, há o iterador (borda preta): caso o elemento na posição do iterador seja menor que o pivô, então faz uma troca entre os elementos da posição do iterador (borda preta) e o elemento da posição do índice do pivô (borda laranja), isso é necessário para fazer a divisão entre elementos maiores e menores que o pivô.

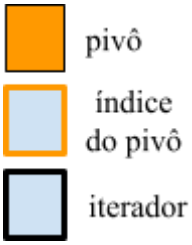
Ao finalizar a iteração, o pivô, que antes estava no fim do *array*, será colocado na posição do índice do pivô (borda laranja).

Figura 14 - Código fonte para o *QuickSort*.

```
6   void swap(int *v, int i, int j) {
7       int temp = v[i];
8       v[i] = v[j];
9       v[j] = temp;
10  }
11
12  int particiona(int *v, int ini, int fim, bool randomizacao){
13      int pivoIndex = ini;
14      if (randomizacao){
15          srand(time(0));
16
17          int i_pivo = rand() % (fim - ini + 1) + ini;
18
19          swap(v, i_pivo, fim);
20      }
21      int pivo = v[fim];
22
23      for(int i = ini; i < fim; i++){
24          if(v[i] <= pivo){
25              swap(v, pivoIndex, i);
26              pivoIndex++;
27          }
28      }
29      swap(v, pivoIndex, fim);
30      return pivoIndex;
31  }
32
33  void quickSort(int *v, int ini, int fim, bool randomizacao) {
34      int tam = fim - ini + 1;
35      if (tam>1) {
36          int pivoIndex = particiona(v, ini, fim, randomizacao);
37          quickSort(v, ini, pivoIndex - 1, randomizacao);
38          quickSort(v, pivoIndex + 1, fim, randomizacao);
39      }
40  }
```

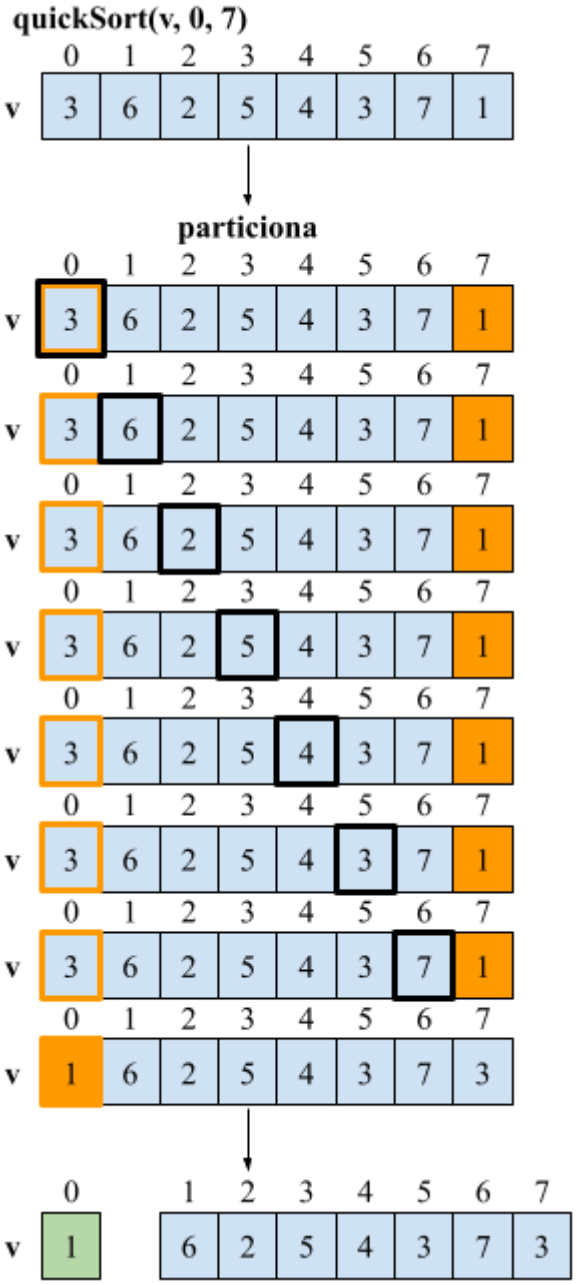
Fonte: Elaboração própria.

Figura 15 - Legenda para as seguintes ilustrações para o *QuickSort*.



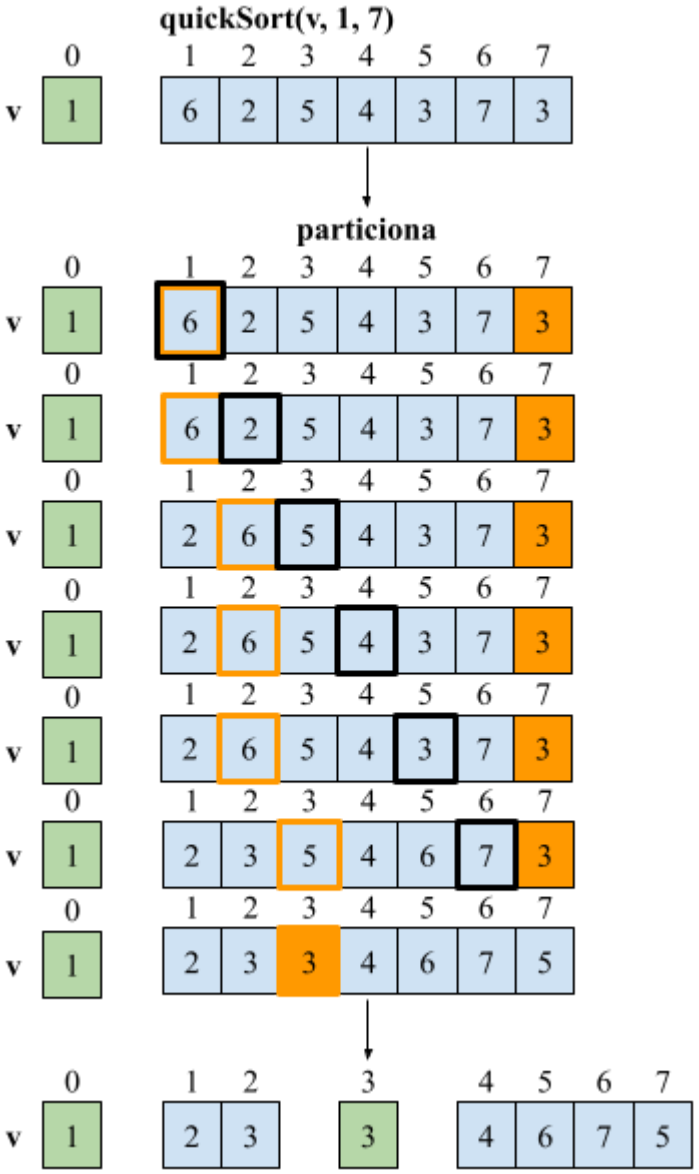
Fonte: Elaboração própria.

Figura 16 - Primeira iteração do *QuickSort* sem randomização de pivô.



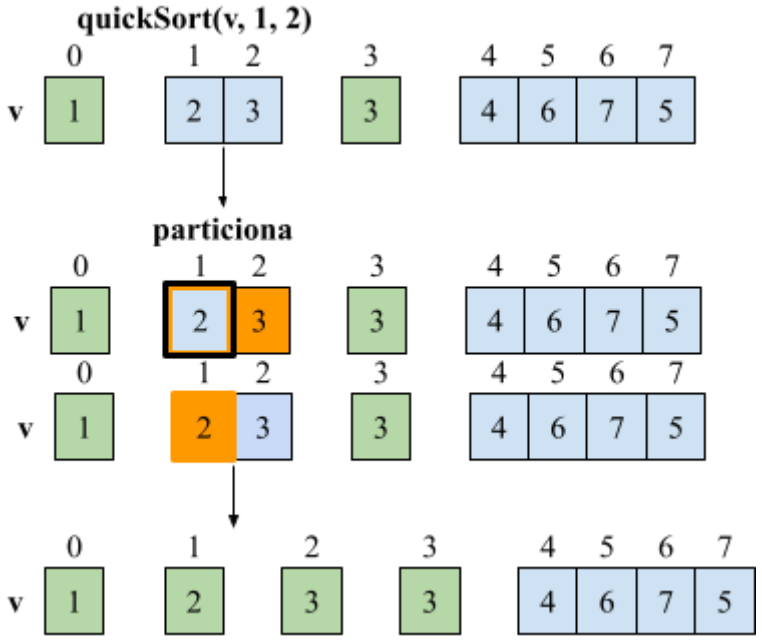
Fonte: Elaboração própria.

Figura 17 - Segunda iteração do *QuickSort* sem randomização de pivô.



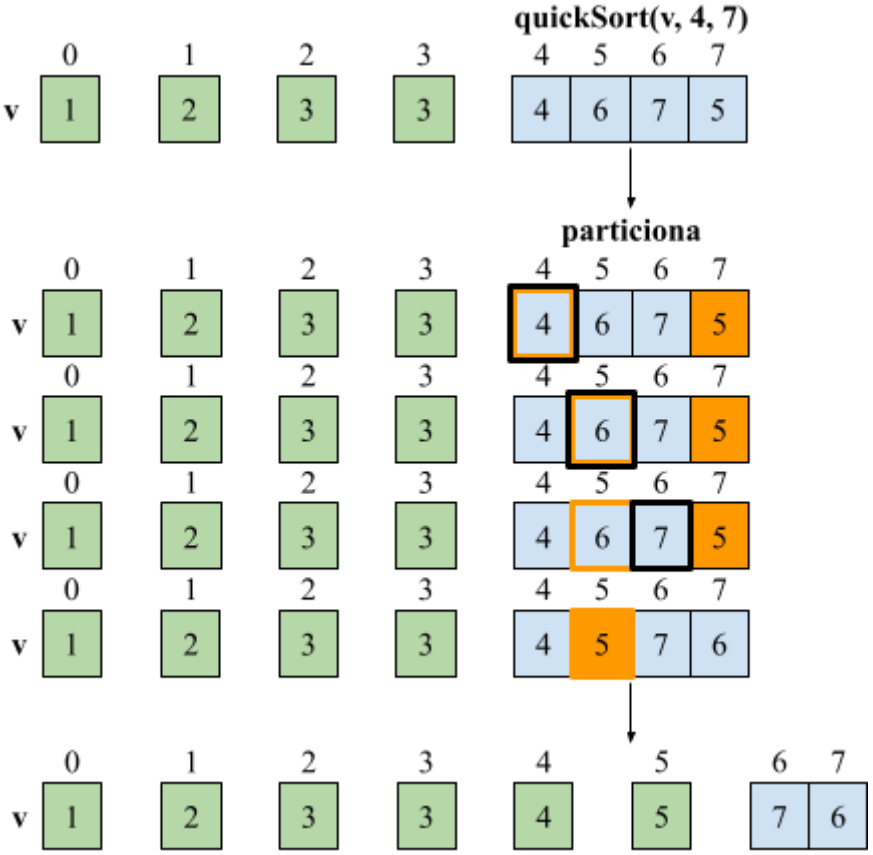
Fonte: Elaboração própria.

Figura 18 - Terceira iteração do *QuickSort* sem randomização de pivô.



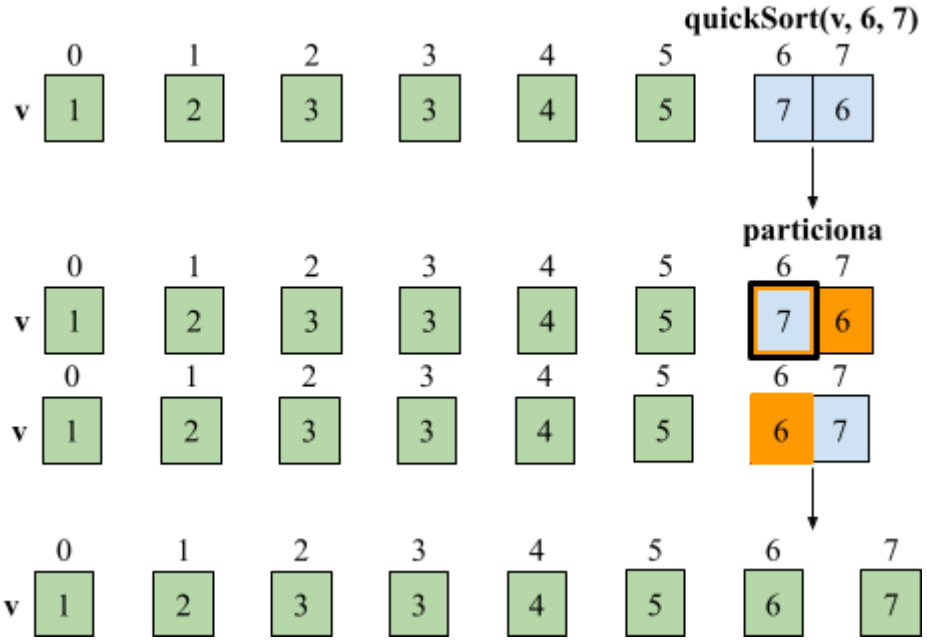
Fonte: Elaboração própria.

Figura 19 - Quarta iteração do *QuickSort* sem randomização de pivô.



Fonte: Elaboração própria.

Figura 20 - Quinta iteração do *QuickSort* sem randomização de pivô.



Fonte: Elaboração própria.

PARTE 2

A parte 2 do relatório pede para implementar o *QuickSort* com seleção randomizada do pivô, mas o código já foi desenvolvido pensando nos dois casos. Foi pensado em adicionar mais um parâmetro à função *quickSort*: um booleano que indica se deseja randomização (*true*) ou não (*false*), como pôde ser visto na Figura 14.

PARTE 3

Para a terceira parte do relatório, serão feitos alguns experimentos com os seguintes algoritmos: *SelectionSort* (*in-place*), *BubbleSort* (melhor versão), *InsertionSort* (*in-place*, melhor versão), *MergeSort*, *QuickSort* (sem e com seleção randomizada de pivô) e *CountingSort*. Os experimentos seguirão da seguinte forma: serão criados vetores com tamanhos 10^1, 10^3, 10^4 e 10^5; para cada tamanho, serão criados três vetores, sendo um aleatório, um ordenado de forma crescente e um vetor ordenado de forma decrescente; para cada combinação, serão executadas 30 repetições, computada a média e mediana dos tempos de execuções e, para isso, serão usadas sementes para assegurar que cada teste seja feito com vetores iguais. Por fim, será feita uma análise dissertativa sobre a performance dos algoritmos para diferentes vetores e tamanhos, explicando quais algoritmos têm boa performance em quais situações.

O código foi desenvolvido para fazer todos os testes e computar todos os dados necessários mas apenas para um tamanho, sendo assim, ele está preparado para ser reutilizado quantas vezes forem desejadas apenas alterando a variável que indica o tamanho dos *arrays*. Ele foi desenvolvido da seguinte forma: serão gerados os três tipos de vetores; eles serão ordenados; computados os tempos de execução; e repetido esse processo 30 vezes (sendo, a cada vez, os mesmos vetores por utilizarem a mesma semente); após os 30 testes, é calculada a média e a mediana dos tempos e imprimidos em tela; toda essa lógica é repetida para cada um dos sete algoritmos de ordenação.

Sendo assim, as análises serão divididas por tamanho dos *arrays* e o código-fonte completo desenvolvido estará no Anexo A.

***Arrays* de tamanho 10^1**

Vale destacar que os dados estão expressos em milissegundos (ms), com 6 casas após a vírgula. No primeiro teste, Figura 21, foi utilizado um limite superior de 1000 para os valores dos elementos; já para o segundo teste, Figura 22, foi utilizado para o limite superior a constante RAND_MAX.

Figura 21 - Testes realizados para *arrays* de tamanho 10^1 e limite superior de 1000.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	0	0	0	0	0	0
bubbleSort	0	0	0	0	0	0
insertionSortIP	0	0	0	0	0	0
mergeSort	0	0	0	0	0	0
quickSort SR	0	0	0	0	0	0
quickSort CR	0	0	0	0	0	0
countingSort	0,133333	0	0	0	0	0

Fonte: Elaboração própria.

Figura 22 - Testes realizados para *arrays* de tamanho 10^1 e limite superior de RAND_MAX.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	0	0	0	0	0	0
bubbleSort	0	0	0	0	0	0
insertionSortIP	0	0	0	0	0	0
mergeSort	0	0	0	0	0	0
quickSort SR	0	0	0	0	0	0
quickSort CR	0	0	0	0	0	0
countingSort	0,133333	0	0,1	0,1	0	0

Fonte: Elaboração própria.

Arrays de tamanho 10^3

Figura 23 - Testes realizados para *arrays* de tamanho 10^3 e limite superior de 1000.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	1,4	1	0,833333	0,833333	1,366667	1,366667
bubbleSort	2,333333	2	0	0	0	0
insertionSortIP	0,533333	0	0	0	0	0
mergeSort	0,333333	0	0,1	0,1	0,233333	0,233333
quickSort SR	0	0	2,833333	2,833333	3,2	3,199999
quickSort CR	0,066667	0	3,133333	3,133333	2,866667	2,866665
countingSort	0,2	0	0	0	0	0

Fonte: Elaboração própria.

Figura 24 - Testes realizados para *arrays* de tamanho 10^3 e limite superior de RAND_MAX.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	2,2	2	0	0	0,033333	0,033333
bubbleSort	0,533333	0	0	0	0	0
insertionSortIP	0	0	0	0	0	0
mergeSort	0,033333	0	0,1	0,1	0,633333	0,633333
quickSort SR	0	0	2,766667	2,766667	3,233333	3,233333
quickSort CR	0,133333	0	2,933333	2,933333	2,933333	2,933333
countingSort	0,2	0	0,133333	0,133333	0,066667	0,066667

Fonte: Elaboração própria.

Arrays de tamanho 10^4

Figura 25 - Testes realizados para *arrays* de tamanho 10^4 e limite superior de 1000.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	117,833333	118	116,433333	116,433289	115,133333	115,133301
bubbleSort	284,366667	283	0	0	0,033333	0,033333
insertionSortIP	60,4	62	0	0	0	0
mergeSort	2,033333	0	0,433333	0,433333	3,033333	3,033333
quickSort SR	0,666667	0	290,633333	290,633301	287,033333	287,033203
quickSort CR	2,3	0	289,5	289,5	288	288
countingSort	0	0	0	0	0,533333	0,533333

Fonte: Elaboração própria.

Figura 25 - Testes realizados para *arrays* de tamanho 10^4 e limite superior de RAND_MAX.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	116,166667	116	115,6	115,599976	115,933333	115,933289
bubbleSort	286,333333	286	0	0	1	1
insertionSortIP	61,366667	61	0,1	0,1	0,433333	0,433333
mergeSort	2,8	3	2,133333	2,133333	2,333333	2,333333
quickSort SR	1,466667	1	289,166667	289,166504	288,4	288,399902
quickSort CR	1,7	2	289,4	289,399902	289,366667	289,366455
countingSort	0,3	0	0,2	0,2	0,166667	0,166667

Fonte: Elaboração própria.

Arrays de tamanho 10^5

Para os *arrays* de tamanho 5, tentou-se fazer um teste com todos os algoritmos, mas o tempo passava de 20 minutos e não houve resultado. Em seguida foi feito um teste removendo o *SelectionSort* (por sempre obter um desempenho $O(n^2)$) e os *QuickSort* (por obter um rendimento ruim para os casos de vetores ordenados). Por fim, foi feito um último teste também removendo o *SelectionSort*, mas agora removendo o *BubbleSort* e o *InsertionSort*, por possuírem rendimento ruim para *arrays* desordenados; também foi removido os vetores ordenados, voltado os *QuickSort* e teste realizado apenas para vetores desordenados. Sendo assim, o teste aconteceu com os melhores algoritmos, segundo os testes feitos.

Figura 26 - Testes realizados para *arrays* de tamanho 10^5 e limite superior de 1000.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	-	-	-	-	-	-
bubbleSort	33611,4	33417	0,1	0,1	2,733333	2,733332
insertionSortIP	6125,266667	6117	0,633333	0,633333	1,3	1.299.999
mergeSort	31,8	32	23,366667	23,366653	23,966667	23,96666
quickSort SR	-	-	-	-	-	-
quickSort CR	-	-	-	-	-	-
countingSort	2,066667	1	2,366667	2,366665	0,766667	0,766666

Fonte: Elaboração própria.

Figura 27 - Testes realizados para *arrays* de tamanho 10^5 e limite superior de 1000.

Algoritmo	Vetor aleatório		Vetor ordenado crescente		Vetor ordenado decrescente	
	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
selectionSortIP	-	-	-	-	-	-
bubbleSort	-	-	-	-	-	-
insertionSortIP	-	-	-	-	-	-
mergeSort	31,633333	32	-	-	-	-
quickSort SR	16,733333	16	-	-	-	-
quickSort CR	20,9	21	-	-	-	-
countingSort	1,266667	1	-	-	-	-

Fonte: Elaboração própria.

Análises

O algoritmo do *SelectionSort* tem um desempenho $O(n^2)$ independente de como o *array* esteja arranjado, sendo assim, obteve bastante dificuldade dificuldade já ao chegar ao tamanho n^4 e se tornando totalmente inviável ao o n^5, onde a espera foi mais de 20 minutos quando incluindo-o nos testes.

O algoritmo *BubbleSort* e o *InsertionSort* têm um desempenho $O(n^2)$ no pior caso (*array* aleatório) e $O(n)$ no melhor caso (*array* ordenado de forma crescente ou decrescente), como pôde ser observado nos testes.

No *QuickSort*, não houve diferença significativa entre com e sem randomização de pivô, mas o algoritmo se saiu ruim nos casos em que os *arrays* estavam ordenados.

Por fim, o *CountingSort*. Os testes foram feitos duplos, com um limite baixo e outro alto, com o objetivo principal de checar o desempenho do *CountingSort* que cria um *array* extra com tamanho $maiorElemento - menorElemento + 1$, sendo assim, se a diferença entre os dois fosse muito grande, poderia haver algum atraso, entretanto, nos testes realizados não houve diferença significativa (em nenhum algoritmo testado), principalmente por esse algoritmo ser $O(n)$.

ANEXO A

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

// função auxiliar: swap
void swap(int *v, int i, int j){
    int temp = v[i];

    v[i] = v[j];
    v[j] = temp;
}

void swapDOUBLE(double *v, int i, int j){
    int temp = v[i];

    v[i] = v[j];
    v[j] = temp;
}

// selectionSort in-place
void selectionSortIP(int *v, int tamanho){
    for(int j=0; j<tamanho-1; j++){
        int iMenor = j;

        for(int i=j+1; i<tamanho; i++){
            if(v[i] < v[iMenor]) iMenor = i;
        }

        int temp = v[j];
        v[j] = v[iMenor];
        v[iMenor] = temp;
    }
}

// bubbleSort
void bubbleSort(int* v, int n){
    for(int j = 0; j < n-1; j++){
        bool trocou = false;

        for(int i=0; i < n-j-1; i++){
            if(v[i] > v[i+1]){
```

```
                swap(v, i, i+1);
                trocou = true;
            }
        }

        if(!trocou) return;
    }
}

// insertionSort in-place
void insertionSortIP(int* v, int tamanho){
    for(int i=1; i < tamanho; i++){
        int valor = v[i];
        int j;

        for(j=i; j > 0 && v[j-1 ] > valor; j--){
            v[j] = v[j-1];
        }

        v[j] = valor;
    }
}

// mergeSort
void merge(int *v, int tamV, int *e, int tamE, int *d, int tamD){
    int indexV = 0, indexE = 0, indexD = 0;

    while(indexE < tamE && indexD < tamD){
        if(e[indexE] <= d[indexD]){
            v[indexV] = e[indexE];
            indexE++;
        } else{
            v[indexV] = d[indexD];
            indexD++;
        }
        indexV++;
    }
    while(indexE < tamE){
        v[indexV] = e[indexE];
        indexE++;
        indexV++;
    }
    while(indexD < tamD){
```

```
            v[indexV] = d[indexD];
            indexD++;
            indexV++;
        }
    }

    void mergeSort(int *v, int tamV){
        if(tamV > 1){
            int meio = tamV/2, tamE = meio, tamD = tamV - meio;
            int *e = (int*)malloc(tamE*sizeof(int));
            int *d = (int*)malloc(tamD*sizeof(int));

            for(int i = 0; i < meio; i++){
                e[i] = v[i];
            }
            for(int i = meio; i < tamV; i++) {
                d[i-meio] = v[i];
            }

            mergeSort(e, tamE);
            mergeSort(d, tamD);

            merge(v, tamV, e, tamE, d, tamD);

            free(e);
            free(d);
        }
    }

    // quickSort
    int particiona(int *v, int ini, int fim, bool randomizacao){
        int pivoIndex = ini;

        if(randomizacao){
            srand(time(0));

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

            swap(v, i_pivo, fim);
        }

        int pivo = v[fim];
```

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

    swap(v, pivoIndex, fim);

    return pivoIndex;
}

void quickSort(int *v, int ini, int fim, bool randomizacao){
    int tam = fim - ini + 1;
    if (tam>1){
        int pivoIndex = particiona(v, ini, fim, randomizacao);
        quickSort(v, ini, pivoIndex - 1, randomizacao);
        quickSort(v, pivoIndex + 1, fim, randomizacao);
    }
}

int particionaDOUBLE(double *v, int ini, int fim, bool randomizacao){
    int pivoIndex = ini;

    if(randomizacao){
        srand(time(0));

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

        swapDOUBLE(v, i_pivo, fim);
    }

    int pivo = v[fim];

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

    swapDOUBLE(v, pivoIndex, fim);
```

```
    return pivoIndex;
}

void quickSortDOUBLE(double *v, int ini, int fim, bool randomizacao){
    int tam = fim - ini + 1;
    if (tam>1){
        int pivoIndex = particionaDOUBLE(v, ini, fim, randomizacao);
        quickSortDOUBLE(v, ini, pivoIndex - 1, randomizacao);
        quickSortDOUBLE(v, pivoIndex + 1, fim, randomizacao);
    }
}

// countingSort
void countingSort(int *v, int tamanho){
    int iMenor = 0, iMaior = 0;

    for(int i = 0; i < tamanho; i++){
        if(v[i] < v[iMenor]) iMenor = i;
        if(v[i] > v[iMaior]) iMaior = i;
    }

    int vMenor = v[iMenor], vMaior = v[iMaior];
    int tamanho_c = vMaior - vMenor + 1;
    int* c = (int*)calloc(tamanho_c, sizeof(int));

    for(int i = 0; i < tamanho; i++){
        c[v[i] - vMenor]++;
    }

    int index_v = 0;

    for(int i=0; i < tamanho_c; i++){
        int j = c[i];
        while(j > 0){
            v[index_v] = i + vMenor;
            index_v++;
            j--;
        }
    }
}

// função auxiliar: instancia array
int* instanciaINT(int tamanho){
```

```
    int *v = (int*)malloc(tamanho * sizeof(int));

    return v;
}

double* instanciaDOUBLE(int tamanho){
    double *v = (double*)malloc(tamanho * sizeof(double));

    return v;
}

// funções auxiliares: exibir arrays
void exibirArrayINT(int *v, int tamV) {
    printf("[");

    for (int i = 0; i < tamV; i++) {
        printf("%d", v[i]);

        if (i < tamV - 1) printf(", ");
    }

    printf("]\n");
}

void exibirArrayDOUBLE(double *v, int tamV) {
    printf("[");

    for (int i = 0; i < tamV; i++) {
        printf("%lf", v[i]);

        if (i < tamV - 1) printf(", ");
    }

    printf("]\n");
}

// função auxiliar: gera arrays
void    geraArrays(int*    v_random,    int*    v_ordenado_cre,    int*
v_ordenado_dec, int tamanho, int limite_superior, time_t seed){
    srand(seed);
    for(int i = 0; i < tamanho; i++){
        v_random[i] = rand();
        if(i == 0){
```

```
            v_ordenado_cre[i] = rand() % limite_superior;
            v_ordenado_dec[i] = rand() % limite_superior;
        } else{
                    v_ordenado_cre[i] = rand() % (limite_superior -
v_ordenado_cre[i-1] + 1) + v_ordenado_cre[i-1];
            v_ordenado_dec[i] = rand() % v_ordenado_dec[i-1] + 1;
        }
    }
}

// função auxiliar: calcula tempo
double ordenaEComputaTempo(int* v, int tamanho, int code){
    /*
    * Códigos:
    * 0 - selectionSort in-place
    * 1 - bubbleSort
    * 2 - insertionSort in-place
    * 3 - mergeSort
    * 4 - quickSort sem randomização de pivô
    * 5 - quickSort com randomização de pivô
    * 6 - countingSort
    */
    clock_t begin, end;

    begin = clock();

    switch(code){
        case 0:
            selectionSortIP(v, tamanho);
            break;
        case 1:
            bubbleSort(v, tamanho);
            break;
        case 2:
            insertionSortIP(v, tamanho);
            break;
        case 3:
            mergeSort(v, tamanho);
            break;
        case 4:
            quickSort(v, 0, tamanho - 1, false);
            break;
        case 5:
```

```
            quickSort(v, 0, tamanho - 1, true);
            break;
        case 6:
            countingSort(v, tamanho);
            break;
        default:
            printf("Erro: código do ordenador inválido (1-7)!\n");
            break;
    }

    end = clock();

    double tempo = (double)(end - begin)/(CLOCKS_PER_SEC/1000);

    //printf("Tempo total: %lf\n", tempo);

    return tempo;
}

int main(){
    int limite_superior = 1000;
    int tamanho = 100000;
    int* v_random = instanciaINT(tamanho);
    int* v_ordenado_cre = instanciaINT(tamanho);
    int* v_ordenado_dec = instanciaINT(tamanho);
    double* tempos_v_random = instanciaDOUBLE(30);
    double* tempos_v_ordenado_cre = instanciaDOUBLE(30);
    double* tempos_v_ordenado_dec = instanciaDOUBLE(30);

    time_t seed = time(NULL);


printf("-----------------------------------------------------------\n");
    for(int i = 0; i < 7; i++){  // Laço para 7 algoritmos
                double media_v_random = 0, media_v_ordenado_cre = 0,
media_v_ordenado_dec = 0;
        // if(i == 0 || i == 1 || i == 2) continue;
         for(int j = 0; j < 30; j++){ // Laço para preencher os vetores
de tempo
                    geraArrays(v_random, v_ordenado_cre, v_ordenado_dec,
tamanho, limite_superior, seed);
            // Vetor com valores aleatórios:
```

```
            tempos_v_random[j] = ordenaEComputaTempo(v_random, tamanho,
i);
            media_v_random += tempos_v_random[j];
            // Vetor ordenado crescente:
                                        tempos_v_ordenado_cre[j]    =
ordenaEComputaTempo(v_ordenado_cre, tamanho, i);
            media_v_ordenado_cre += tempos_v_ordenado_cre[j];
            // Vetor ordenado decrescente:
                                        tempos_v_ordenado_dec[j]    =
ordenaEComputaTempo(v_ordenado_dec, tamanho, i);
            media_v_ordenado_dec += tempos_v_ordenado_dec[j];
        }
        // Ordena os vetores de tempo para cálculo de mediana
        quickSortDOUBLE(tempos_v_random, 0, 29, true);
        quickSortDOUBLE(tempos_v_ordenado_cre, 0, 29, true);
        quickSortDOUBLE(tempos_v_ordenado_dec, 0, 29, true);

        switch(i){
            case 0:
                printf("selectionSortIP:\n");
                break;
            case 1:
                printf("bubbleSort:\n");
                break;
            case 2:
                printf("insertionSortIP:\n");
                break;
            case 3:
                printf("mergeSort:\n");
                break;
            case 4:
                printf("quickSort sem randomização de pivô:\n");
                break;
            case 5:
                printf("quickSort com randomização de pivô:\n");
                break;
            case 6:
                printf("countingSort:\n");
                break;
            default:
                printf("Erro: código do ordenador inválido (1-7)!\n");
                break;
        }
```

```
            printf("Média, em milissegundos, para array com elementos
aleatórios: %lf\n", media_v_random /= 30);
          printf("Mediana, em milissegundos, para array com elementos
aleatórios: %lf\n", tempos_v_random[15]);

        printf("Média, em milissegundos, para array ordenado crescente:
%lf\n", media_v_ordenado_cre /= 30);
          printf("Mediana, em milissegundos, para array com elementos
aleatórios: %lf\n", v_ordenado_cre[tamanho/2]);

              printf("Média, em milissegundos, para array ordenado
decrescente: %lf\n", media_v_ordenado_dec /= 30);
          printf("Mediana, em milissegundos, para array com elementos
aleatórios: %lf\n", v_ordenado_dec[tamanho/2]);

printf("-------------------------------------------------------\n");
    }

    return 0;
}
```