UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
ALGORITMOS E ESTRUTURA DE DADOS I

Discente: Luiz Henrique Araújo Dantas - 20180109005

ATIVIDADE AVALIATIVA II

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

Questão 1

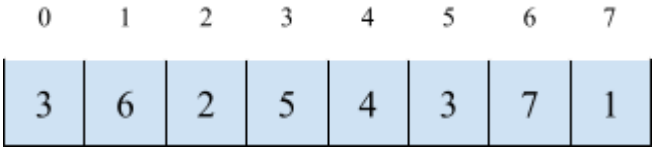
a. SelectionSort in-place com vetor a

O SelectionSort é um dos algoritmos de ordenação mais simples que existem, ele pode ser descrito da seguinte forma:

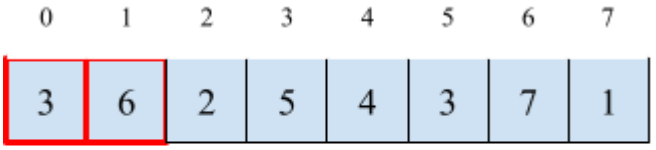
I. Selecione o menor elemento do vetor;
II. Troque de posição com o elemento que está no início do vetor;
III. Repita essas operações com os *n-1* elementos que restam, depois com *n-2* e assim por diante, até que sobre apenas um elemento.

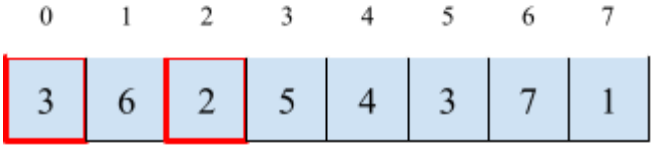
O SelectionSort em si é um dos piores algoritmos de ordenação existentes, pois ele possui uma complexidade de tempo na ordem de $O(n^2)$, apesar da sua ordenação ser muita intuitiva é necessário passar diversas varreduras no vetor para que possamos ordenar todos os elementos.

Nas imagens abaixo utilizamos o SelectionSort para ordenar o vetor A se baseando nos passos descritos anteriormente, o processo em si é massante e possui muitos passos para que o vetor seja completamente organizado.

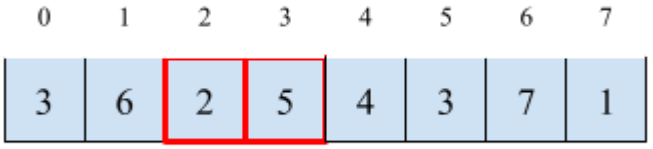


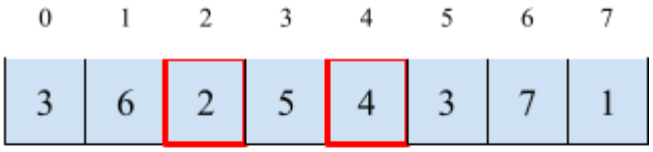
Vetor A inicialmente desordenado.

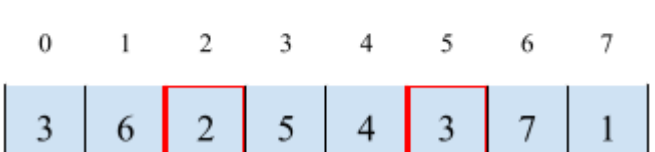


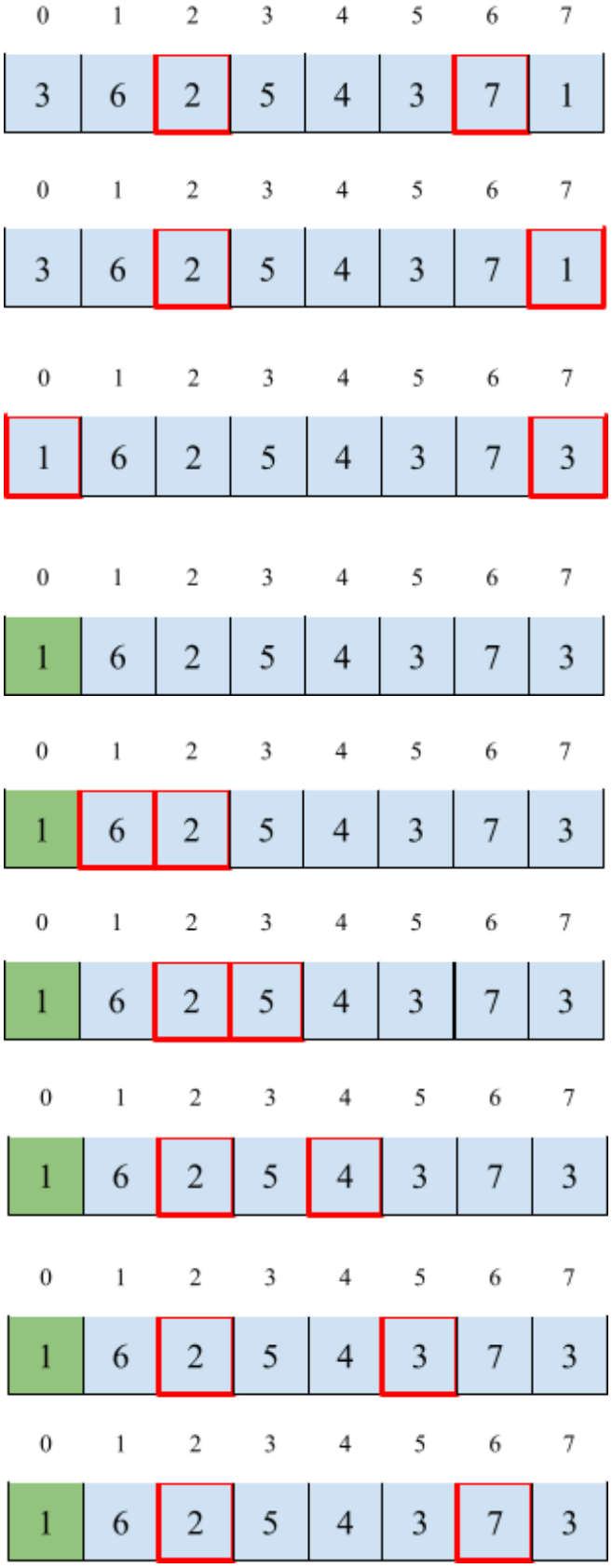


Começamos a buscar o menor elemento presente no vetor, serão feitas diversas verificações entre os elementos para que seja possível encontrar o menor elemento, para que, assim, consiga ser feita a troca de posição com o elemento no início do vetor.



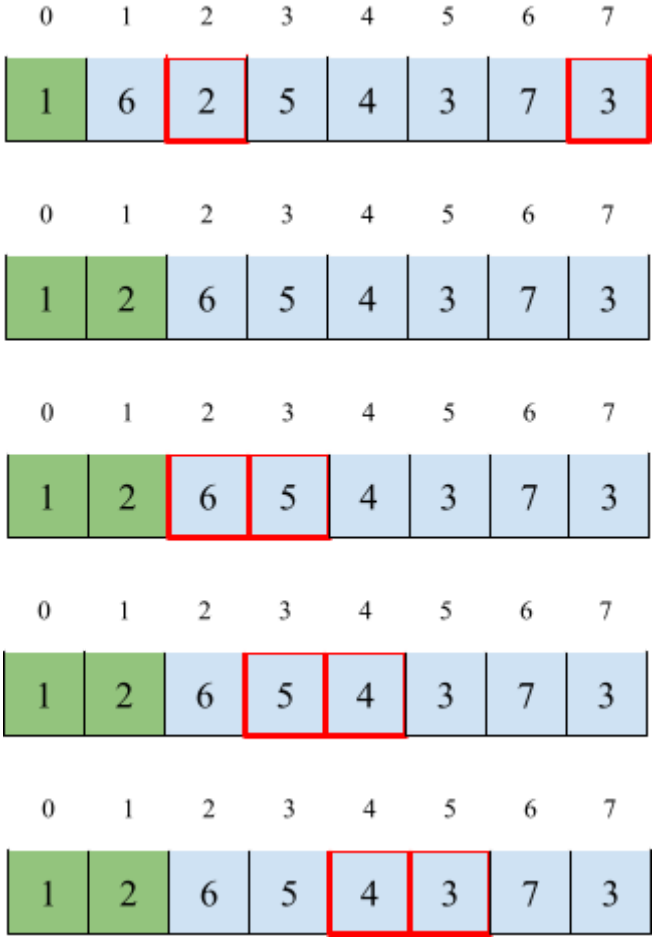


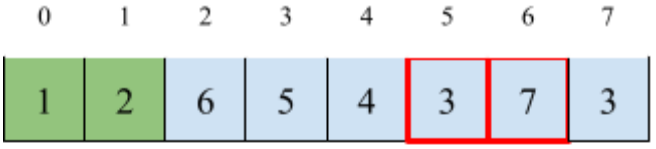


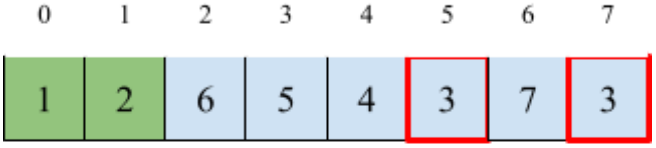


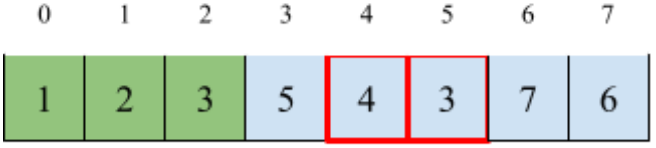
Encontrado o menor elemento no vetor, é feita a troca de posição para o início do vetor.

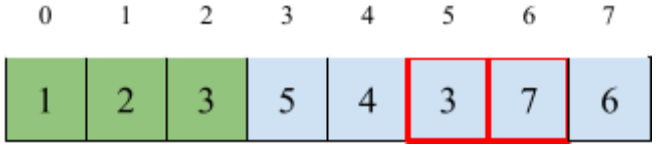
O processo continua, pois o vetor ainda não está completamente ordenado, com isso o processo se repete, encontrando o menor valor do vetor e trocando de posição.


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

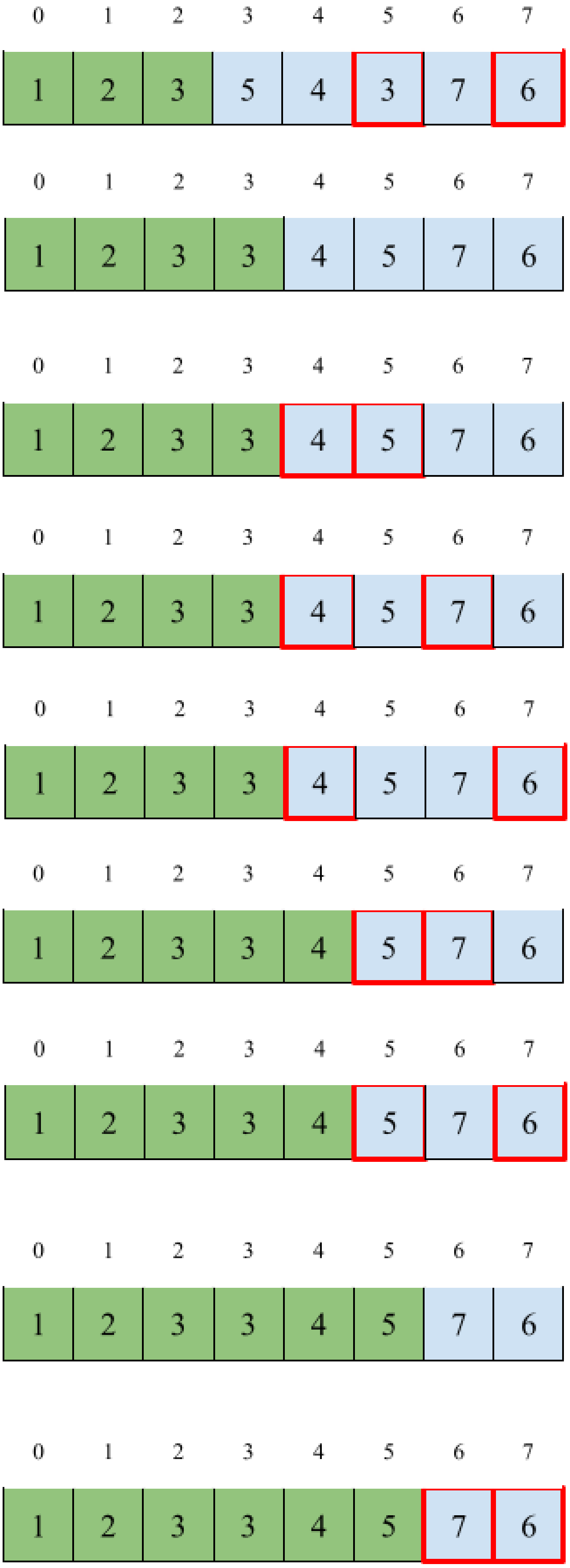




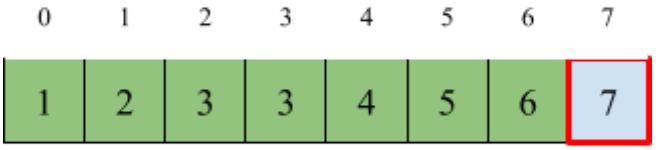




O processo continua, pois o vetor ainda não está completamente ordenado, com isso o processo se repete, encontrando o menor valor do vetor e trocando de posição.



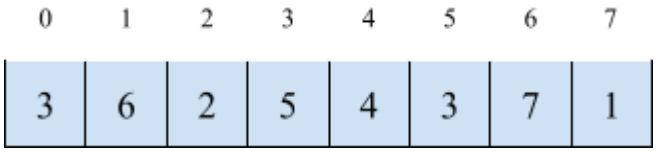
O processo continua, pois o vetor ainda não está completamente ordenado, com isso o processo se repete, encontrando o menor valor do vetor e trocando de posição.



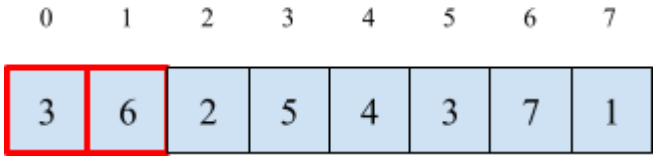
Vetor A ordenado.

b. BubbleSort (melhor versão) com o vetor a

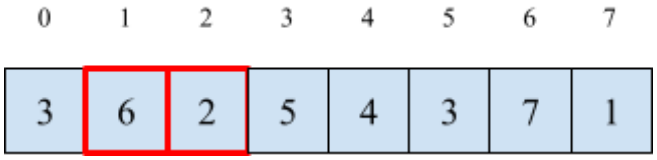
O algoritmo Bubblesort realiza diversas varreduras no vetor verificando se o valor à direita da posição atual é maior ou menor, caso seja menor é realizado a troca entre os valores mantendo sempre o menor à esquerda e o maior à direita do vetor, a seguir será demonstrado a ordenação do vetor a utilizando o algoritmo Bubblesort passo a passo.



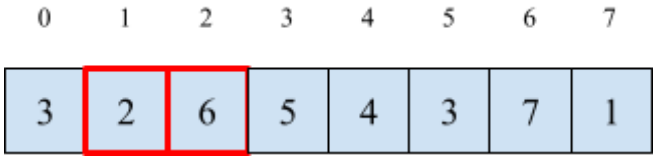
Vetor A inicialmente desordenado



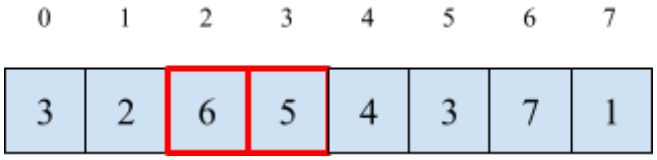
Iniciamos a varredura da esquerda para a direita, verificamos o primeiro elemento com o seguinte. Como 3 < 6, não trocamos de posição.



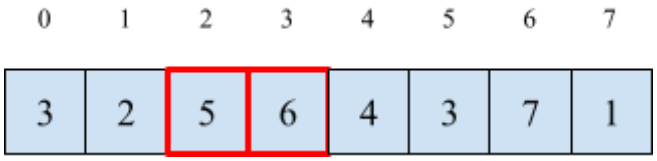
Agora, verificamos que 6 > 2, então trocamos a posição dos elementos, pois 2 é menor que 6.



A varredura continua após a troca de posições.

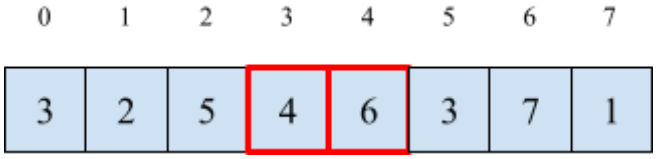


Analisando os elementos, 5 é menor que 6, então há troca de posições.

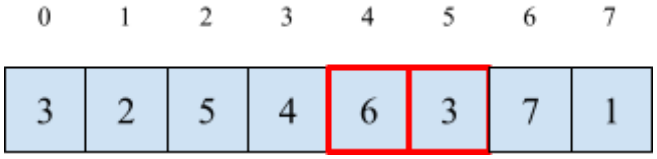


A varredura prossegue assim que é feita a troca de posições.

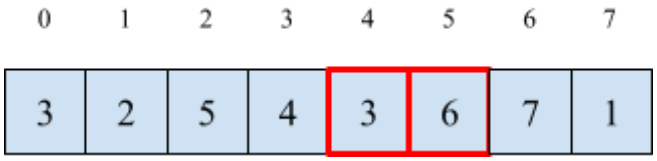
É verificado que 4 < 6, então há troca de posição entre os valores.



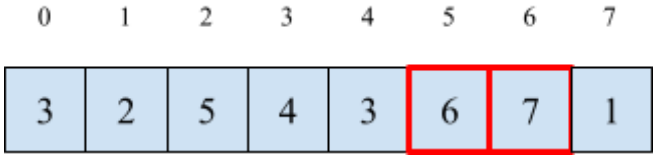
Após realizar a troca de posições, a varredura continua.



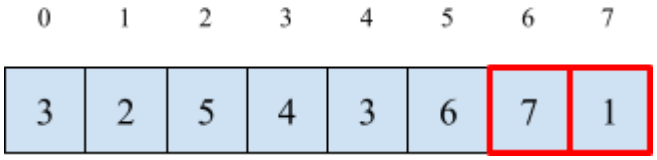
É verificado que 3 é menor que 6, então trocamos a posição dos valores.



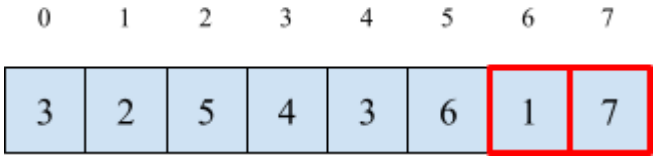
Após realizar a troca de posições, a varredura continua.



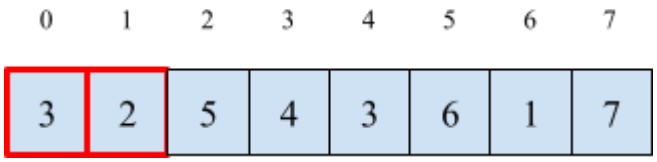
É verificado que 6 é menor que 7, então não há troca de posição entre os valores.



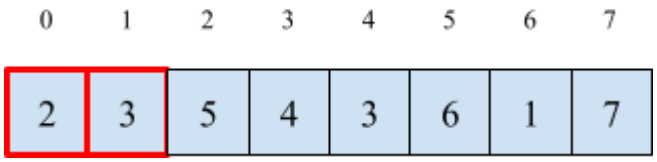
Agora verificamos se 7 < 1, como não é verdade, trocamos as posições dos valores.



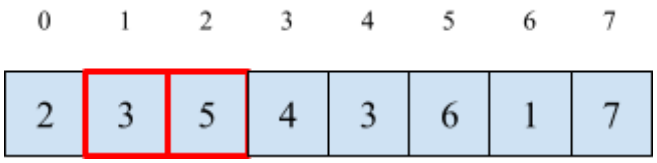
Após realizar a troca de posições, a varredura continua. Podemos verificar que o valor 7 já encontra-se ordenado.



Começamos então a segunda varredura, pois o vetor não está completamente ordenado. Verifica-se que 3 é maior que 2, então há troca de posições.



Após realizar a troca de posições, a varredura continua.



Agora verifica-se que 3 < 5, então não há troca de posições, pois o valor 5 é maior que 3.

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

É verificado se 5 < 4, como não é verdade, trocamos os valores de posição.

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

Após realizar a troca de posições, a varredura continua.

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

É verificado se 5 < 3, como não é verdade, trocamos os valores de posição.

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

Após realizar a troca de posições, a varredura continua.

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

É verificado se 5 < 6, como 6 é maior que 5, não há troca de posições.

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

Seguindo adiante, verificamos se 6 < 1, como não é verdade, é feita a troca de posições entre os valores.

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

Após realizar a troca de posições, a varredura continua.

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

Verificamos se 6 < 7, como é verdade, não há troca de posições, verificamos também que 6 encontra-se ordenado, porém o vetor não está completamente ordenado sendo necessário uma nova varredura.

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

Iniciamos a 3ª varredura.

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

É verificado se 2 < 3, como é verdade não há troca de posições entre os valores.

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

É verificado se 3 < 4, como é verdade não há troca de posições entre os valores.

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

É verificado se 4 < 3, como não é verdade, há troca de posições entre os valores.

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

É verificado se 4 < 5, como é verdade, não há troca de posições entre os valores.

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

É verificado se 5 < 1, como não é verdade, há troca de posições entre os valores.

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

Seguindo adiante, verifica se 5 < 6, como não é verdade, não há troca de posições.

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

Como o vetor não está totalmente ordenado, iniciamos a 4ª varredura, notamos que o valor 5 já está ordenado, junto aos valores 6 e 7.

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

É verificado se 2 < 3, como é verdade, não há troca de posições.

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

É verificado se 3 < 3, como os valores são iguais, não há troca de posições entre os valores.

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

É verificado se 3 < 4, como é verdade, não há troca de posições entre os valores.

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

É verificado se 4 < 1, como 1 é menor que 4, os valores trocam de posição.

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

Após realizar a troca de posições, a varredura continua.

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

Finalizada a 4ª varredura, verificamos que o valor 4 também encontra-se ordenado junto aos outros valores.

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

Iniciamos a 5ª varredura do vetor, é verificado se 2 < 3, como é verdade, não há troca de posição entre os valores.

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

É verificado se 3 < 3, como os valores são iguais, não há troca de posições entre os valores.

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

É verificado se 3 < 1, como não é verdade, os valores trocam de posição.

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

Após realizar a troca de posições, a varredura continua.

Finalizada a 5ª varredura, porém o vetor não está completamente ordenado, sendo necessário fazer outra varredura.

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

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

Iniciada a 6ª varredura. É verificado se 2 < 3, como é verdade, não há troca de posição.

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

É verificado se 3 < 1, como não é verdade, os valores trocam de posição.

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

Após realizar a troca de posições, a varredura continua.

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

Finalizada a 6ª varredura, o vetor ainda não está totalmente ordenado, sendo necessário uma nova varredura no vetor.

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

Iniciada a 7ª varredura, é verificado se 2 < 1, como não é verdade, os valores trocam de posição.

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

Após realizar a troca de posições, a varredura continua.

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

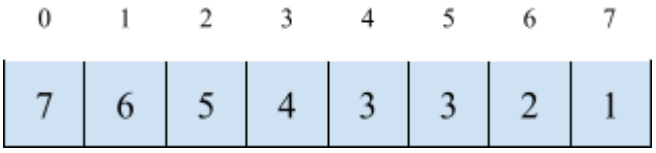
Finalizada a 7ª varredura, o valores do vetor estão todos ordenados.

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

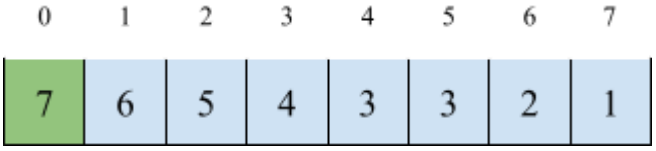
Vetor A ordenado pelo algoritmo BubbleSort.

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

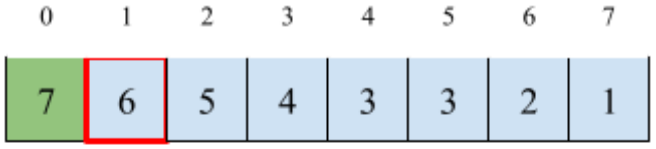
O algoritmo InsertionSort realiza a ordenação do vetor subdividindo o vetor em dois sub-vetores, um ordenado e outro desordenado. O vetor ordenado começa com o primeiro valor ordenado enquanto os outros valores se mantém no sub-vetor desordenado, com isso os valores são ordenados de um em um e irão completar o vetor ordenado. Logo abaixo é apresentado a maneira de ordenação do vetor utilizando InsertionSort.



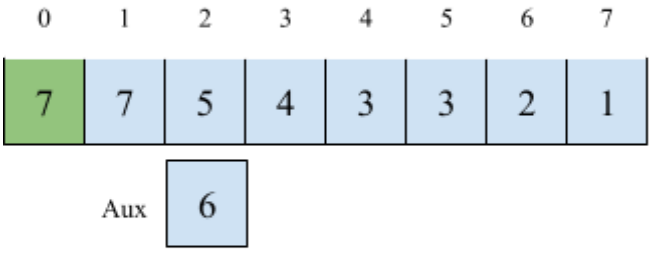
Vetor B inicialmente ordenado reversamente.



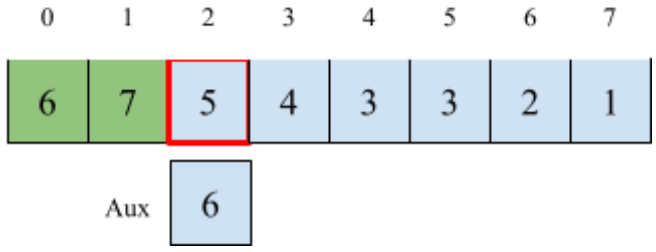
O vetor em verde é o vetor ordenado enquanto o vetor em azul é o vetor desordenado.



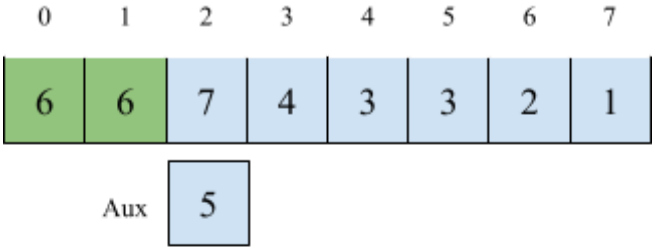
Selecionamos o primeiro índice do sub-vetor em azul e vamos inserir no sub-vetor ordenado.



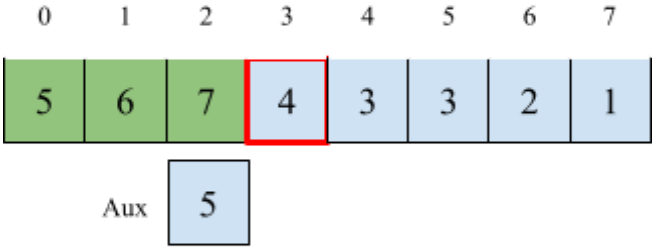
Como 6 < 7, ele ficará antes do 7, o valor 6 é passado para uma variável auxiliar e o valor 7 é copiado para o índice em que o valor 6 estava no sub-vetor desordenado. O índice 0 receberá o valor 6.



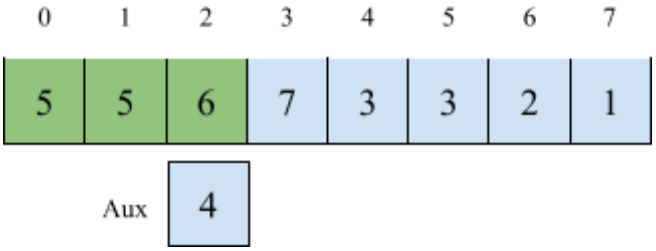
Selecionamos o primeiro índice do sub-vetor desordenado e vamos inserir no sub-vetor ordenado.



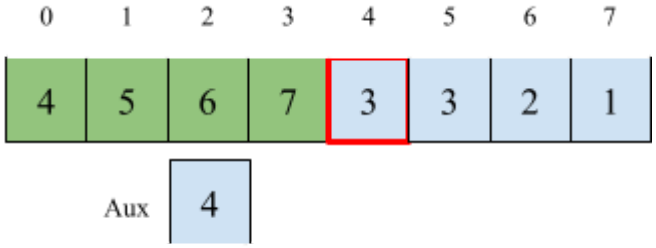
Como 5 < 6, ele ficará antes do 6, o valor 5 é passado para uma variável auxiliar e os valores 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 5.



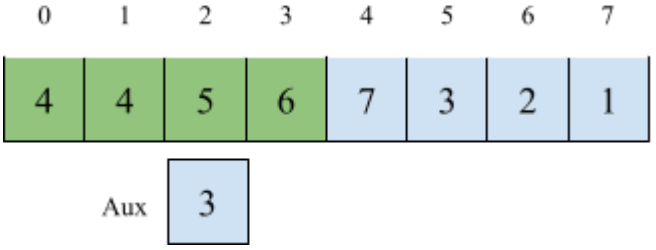
Selecionamos o primeiro índice do sub-vetor desordenado e vamos inserir no sub-vetor ordenado.



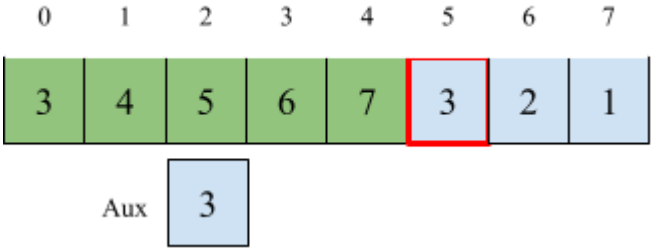
Como 4 < 5, ele ficará antes do 5, o valor 4 é passado para uma variável auxiliar e os valores 5, 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 4.



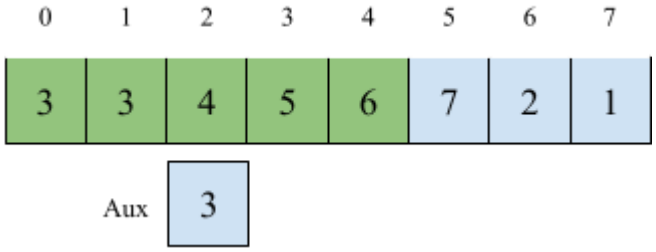
Seguindo adiante, pegamos o primeiro índice do sub-vetor desordenado e vamos inserir no sub-vetor ordenado.



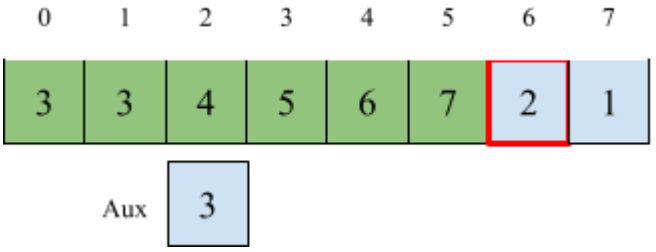
Como 3 < 4, ele ficará antes do 4, o valor 3 é passado para uma variável auxiliar e os valores 4, 5, 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 3, que estava armazenado na variável auxiliar.



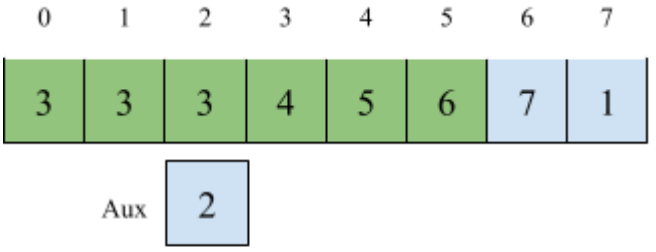
Selecionamos o primeiro índice do sub-vetor em azul e vamos inserir no sub-vetor ordenado.



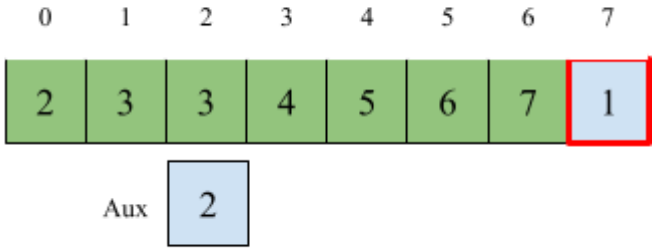
Semelhante na iteração anterior, como 3 < 4, ele ficará antes do 4, o valor 3 é passado para uma variável auxiliar e os valores 3, 4, 5, 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 3, que estava armazenado na variável auxiliar.



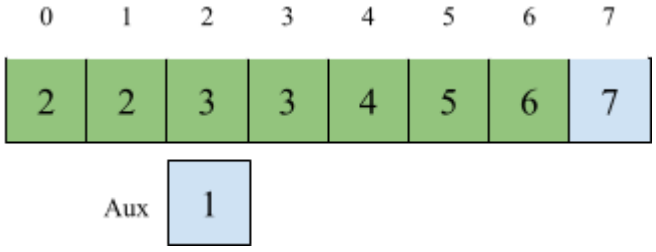
Selecionamos o primeiro índice do sub-vetor desordenado e vamos inserir no sub-vetor ordenado.



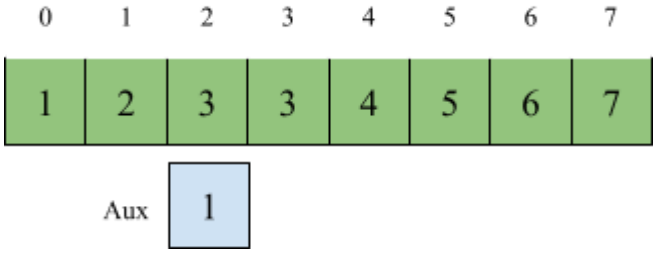
Como 2 < 3, ele ficará antes do 3, o valor 2 é passado para uma variável auxiliar e os valores 3, 3, 4, 5, 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 2, que estava armazenado na variável auxiliar.



Selecionamos o último valor do sub-vetor desordenado e vamos inserir no sub-vetor ordenado.



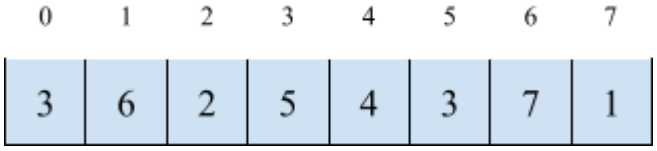
Como 1 < 2, ele ficará antes do 2, o valor 1 é passado para uma variável auxiliar e os valores 2, 3, 3, 4, 5, 6 e 7 são copiados para os próximos índices. O índice 0 receberá o valor 1, que estava armazenado na variável auxiliar.



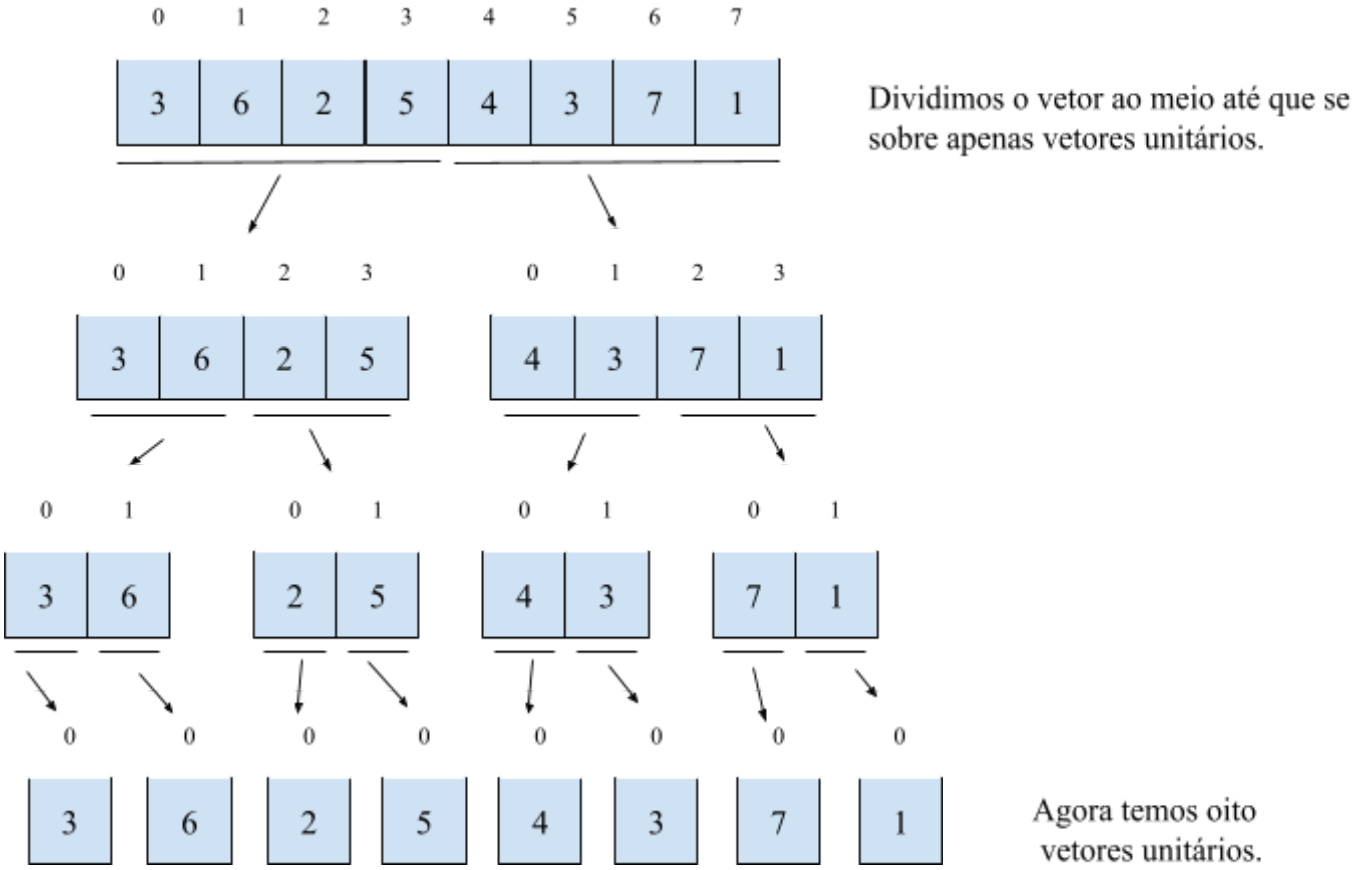
Finalmente temos o vetor B totalmente ordenado a partir do algoritmo InsertionSort.

d. MergeSort com o vetor a

O MergeSort é um algoritmo baseado na ideia de "dividir para conquistar", em outras palavras o vetor será subdividido recursivamente até que seus elementos se tornem vetores unitários e assim, o vetor será ordenado no processo de merge (junção) dessas pequenas partes, gerando o vetor completamente ordenado.

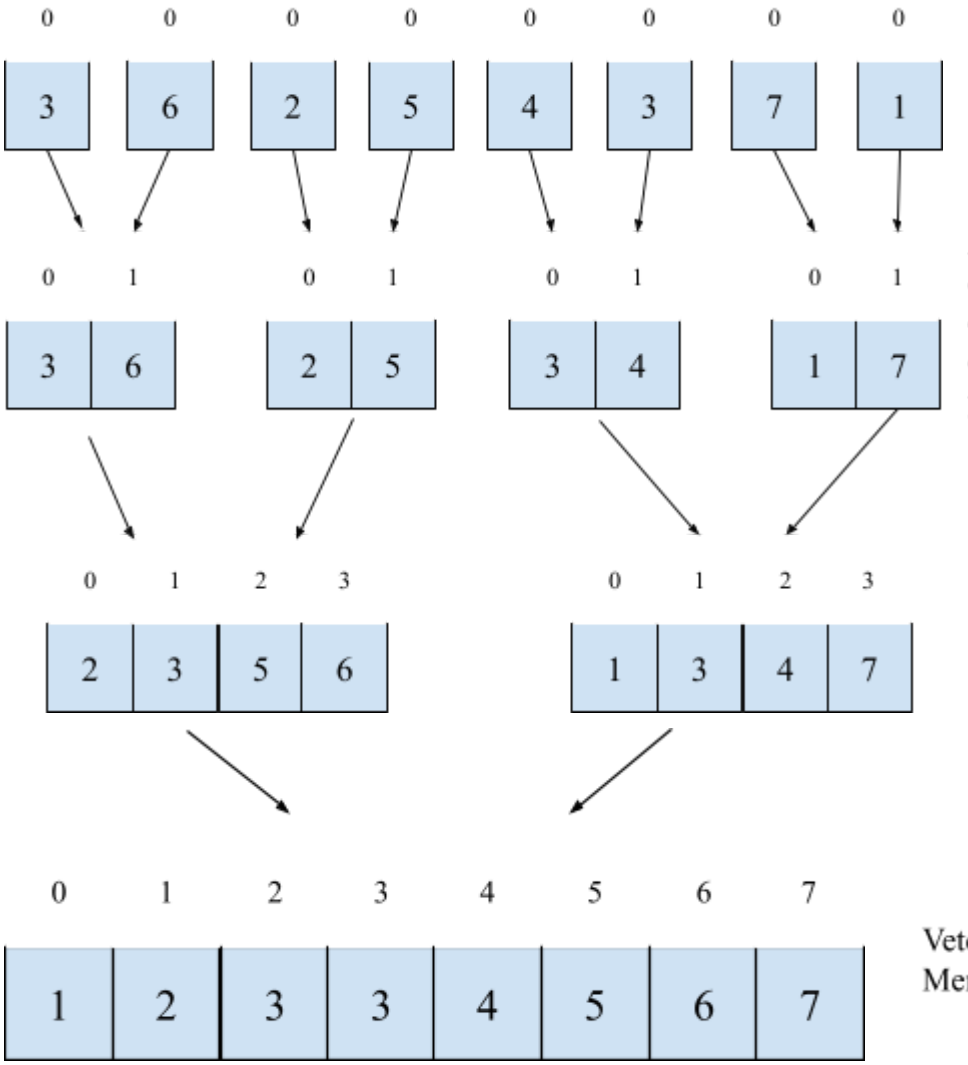


Vetor A inicialmente desordenado



Dividimos o vetor ao meio até que se sobre apenas vetores unitários.

Agora temos oito vetores unitários.

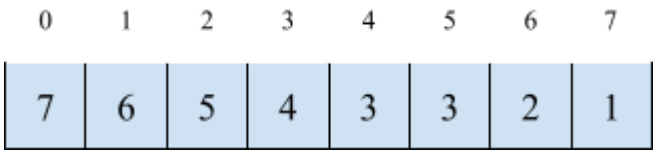


A partir desse momento é feito o caminho inverso, iremos unir os vetores e vamos ordenando os valores dos vetores do menor para o maior.

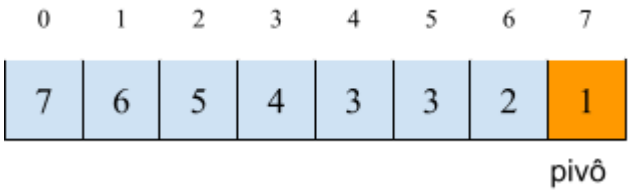
Vetor A ordenado pelo algoritmo MergeSort.

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

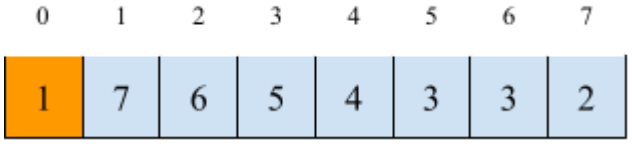
O QuickSort funciona de maneira que, à cada iteração, ele escolhe um dos valores do vetor como pivô (sempre será o último valor do vetor ou sub-vetor), o pivô têm como função de organizar todos os elementos do vetor, colocando os menores à sua esquerda e os maiores à sua direita, criando assim dois sub-vetores, e o pivô se encontra na divisão desses sub-vetores. Na próxima iteração, será escolhido um novo pivô e será novamente aplicado esse método de particionamento até que todos os sub-vetores estejam organizados em pivôs e assim, ordenados.



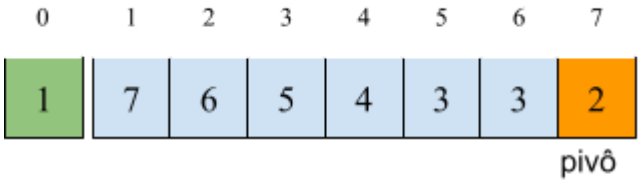
Vetor B inicialmente desordenado



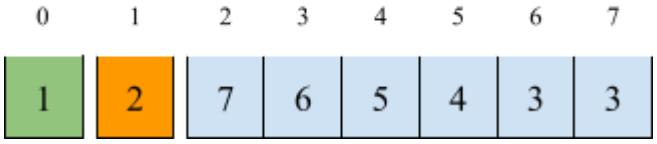
É escolhido um pivô para iniciar a ordenação do vetor. O pivô é o último elemento do vetor.

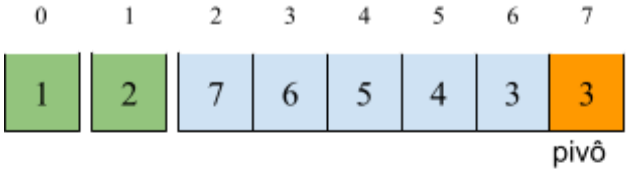


O processo de particionamento já é realizado e assim, devemos selecionar o próximo pivô, que deverá ser ordenado.

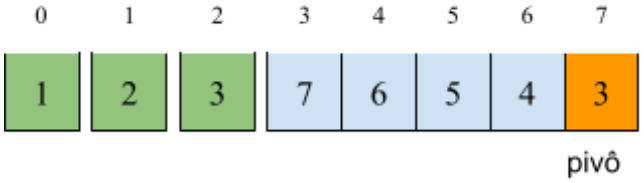


O sub-vetor particionado recebe o valor 2 e deve-se selecionar um novo pivô.



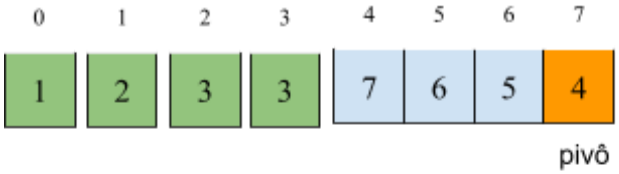


Selecionamos novamente o último elemento do vetor como pivô e devemos realizar novamente o particionamento de ordenação.



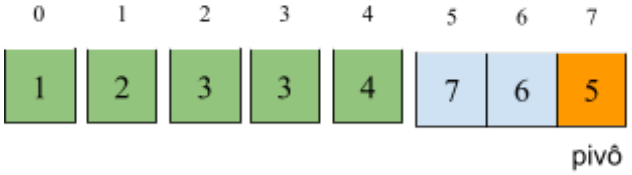
Após realizado o particionamento, devemos selecionar um novo pivô, para continuar o processo de particionamento.

Particionamos o pivô e continuamos ordenando o vetor.

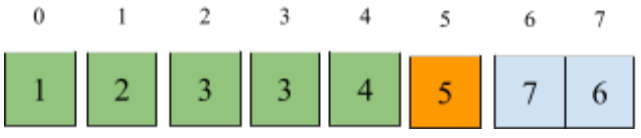


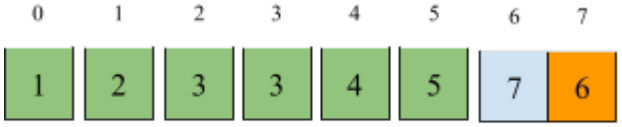
Selecionamos novamente o último elemento do vetor como pivô e devemos realizar novamente o particionamento de ordenação.

O processo de particionamento já é realizado e assim, devemos selecionar o próximo pivô, que deverá ser ordenado.



Sistematicamente o processo é repetido de forma semelhante aos passos anteriores, no qual escolhemos o último elemento do vetor como pivô e realizamos o processo de particionamento do vetor.





É selecionado o último elemento do vetor como pivô, para que seja feito o processamento de particionamento, semelhante aos processos anteriores.

Particionamos o pivô.

Escolhemos o último elemento do vetor como pivô e devemos realizar o processo de particionamento, notamos que ele está ordenado.

Por fim, temos todos os pivôs ordenados e podemos observar que o vetor original encontra-se organizado.

Vetor B ordenado

Questão 2

Como mostrado em aula, para poder randomizar o pivô, é necessário que seja feito um sorteio entre os valores do início e o fim do vetor.

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

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

int particiona(int* v, int ini, int fim){
    srand(time(NULL));
    int pivoSorteado = rand() % fim;
    pivoSorteado = pivoSorteado % (fim - ini + 1) + ini;
    int pIndex = ini;
    int pivo = v[pivoSorteado];
    int i = 0;
    for(int i = ini; i < fim; i++){
        if(v[i] <= pivo){
            swap(v,i,pIndex);
            pIndex++;
        }
    }
    swap(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

Para desenvolver os experimentos com os algoritmos de ordenação vamos considerar vetores de tamanhos 10^1, 10^3 e 10^5, os algoritmos serão submetidos a diversas ordenações com esses tamanhos de vetores, utilizamos três tipos de vetores: vetor ordenado, vetor com valores aleatórios e vetor ordenado de forma decrescente, os algoritmos que iremos realizar os testes são: SelectionSort, BubbleSort, InsertionSort, MergeSort, QuickSort, QuickSort com randomização de pivô e CountingSort.

Iremos analisar os algoritmos de acordo com o tamanho de cada array, sendo assim serão criadas seções para os tamanhos que iremos utilizar.

1. Array com tamanho 10^1

Para vetores dessa dimensão, todos os algoritmos de ordenação obtém bom desempenho. Por se tratar de um vetor de tamanho pequeno todos os algoritmos de ordenação possuem facilidade em ordenar, pois, independente de sua organização os elementos que são colocados em ordem quase instantaneamente, isso apresentará apenas nessa configuração, na próxima configuração, algoritmos como SelectionSort, BubbleSort e InserctionSort terão um declínio em seu desempenho por se tratar de vetores com uma quantidade bem maior do que foi realizado nesse teste.

	Vetor ordenado		Vetor decrescente		Vetor aleatório	
Alg. de Ordenação	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
SelectionSort	0	0	0	0	0	0
BubbleSort	0	0	0	0	0	0
InsertionSort	0	0	0	0	0	0
MergeSort	0	0	0	0	0	0
QuickSort	0	0	0	0	0	0
QuickSort Rand.	0	0	0	0	0	0
CountingSort	0,1	0	0	0	0	0

2. Array com tamanho 10^3

Como mencionado anteriormente, os algoritmos mais simples como SelectionSort, BubbleSort e InserctionSort começam a ter um desempenho questionável para uma aplicação para vetores dessa dimensão e até maiores, pois eles já começam a atuar em $O(n^2)$, que é a situação que não é desejável para os algoritmos de ordenação.

Para os outros algoritmos de ordenação eles vão apresentar uma performance inquestionável, pois possuem uma organização dos vetores de forma dinâmica para esse tamanho de vetor, um dos casos mais lentos podemos citar o QuickSort com/sem randomização, pois quando operam ordenando vetores decrescentes eles tendem à uma performance $O(n^2)$ porém o algoritmo tenta colocar a performance O(n log (n)) (QuickSort com randomização de pivô) para que o algoritmo não entre em seu pior estado.

Destaque para o CountingSort, pois é o algoritmo mais poderoso de todos os que estão sendo analisados, ele possui complexidade O(n) estável (linear), sendo de fato o melhor algoritmo de ordenação dentre os que estão sendo analisados.

	Vetor ordenado		Vetor decrescente		Vetor aleatório	
Alg. de Ordenação	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
SelectionSort	4	3	1	1	1,3	1,3
BubbleSort	2,5	2	0	0	0	0
InsertionSort	0,5	0	0	0	0	0
MergeSort	0,4	0	0,1	0,1	0,2	0,2
QuickSort	0	0	2	2	3	2,999
QuickSort Rand.	0,02	0	3	3	2,8	2,8
CountingSort	0,1	0	0	0	0	0

3. Array com tamanho 10^5

Nessa configuração, ficou impossível a medição dos algoritmos mais simples, no qual o software não respondia mais às tentativas para ordenação, então elenquei esses algoritmos com dados nulos, pois não foi possível obter dados referentes aos vetores.

Destaque também para os algoritmos QuickSort com/sem randomização e MergeSort, no qual só foi possível obter dados para vetores ordenados, pois não foi possível continuar analisando para as situações de vetores decrescentes e aleatório devido à limitação do computador que não conseguiu “segurar a barra” para essa configuração.

O CountingSort foi o algoritmo superior nesse teste, ele foi o único em que foi possível obter todos os parâmetros de teste para os vetores de 1 milhão.

	Vetor ordenado		Vetor decrescente		Vetor aleatório	
Alg. de Ordenação	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)	Média (ms)	Mediana (ms)
SelectionSort	NaN	NaN	NaN	NaN	NaN	NaN
BubbleSort	NaN	NaN	NaN	NaN	NaN	NaN
InsertionSort	NaN	NaN	NaN	NaN	NaN	NaN
MergeSort	245	245	NaN	NaN	NaN	NaN
QuickSort	92,99	87,878	NaN	NaN	NaN	NaN
QuickSort Rand.	62,334	61,6667	NaN	NaN	NaN	NaN
CountingSort	1,99908	1,8883	3,10109	3,0	21,2333	21,2

Para concluir sobre as análises feitas acima, a escolha do melhor algoritmo vai depender da necessidade que o usuário irá enfrentar no dia a dia, pois dentre todos os algoritmos que foram utilizados podemos citar o CountingSort como o único que conseguiu ser efetivo para todas as dimensões de vetor que foram testadas (devido às suas características quase lineares), porém há diversos cenários em que os outros algoritmos podem ser empregados e podem desempenhar performances interessantes e entregar a melhor solução para o problema em questão.