

Projeto e Análise de Algoritmos

Aula 05 – Algoritmos de Ordenação

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Ordenação

- **Problema:**
 - **Entrada:** conjunto de itens a_1, a_2, \dots, a_n ;
 - **Saída:** conjunto de itens permutados em uma ordem $a_{k1}, a_{k2}, \dots, a_{kn}$, tal que, dada uma função de ordenação f , tem-se a seguinte relação: $f(a_{k1}) < f(a_{k2}) < \dots < f(a_{kn})$.
- Ordenar consiste no processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado.

Ordenação

- Qualquer tipo de campo **chave**, sobre o qual exista uma relação de ordem total $<$, para uma dada função de ordenação, pode ser utilizado.
- A relação $<$ deve satisfazer as condições:
 - Apenas uma das seguintes expressões pode ser verdadeira: $a < b$, $a = b$, $a > b$;
 - Se $a < b$ e $b < c$ então $a < c$;
- Um método de ordenação é dito **estável** se a ordem relativa dos itens com chaves iguais mantém-se inalterada pelo processo de ordenação.

Ordenação

Vetor

10	20	30	40	50	60	70	80	90
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 500,00	R\$ 600,00	R\$ 600,00	R\$ 500,00	R\$ 400,00

Estável

10	20	30	40	90	50	80	60	70
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 400,00	R\$ 500,00	R\$ 500,00	R\$ 600,00	R\$ 600,00

Instável

20	10	30	90	40	50	80	70	60
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 400,00	R\$ 500,00	R\$ 500,00	R\$ 600,00	R\$ 600,00

Ordenação

- Os métodos de ordenação são classificados em dois grandes grupos:
 - **Ordenação Interna:** São os métodos que não necessitam de uma memória secundária para o processo, a ordenação é feita na memória principal do computador;
 - Qualquer registro pode ser acessado em tempo $O(1)$;
 - **Ordenação Externa:** Quando o arquivo a ser ordenado não cabe na memória principal e, por isso, tem de ser armazenado em fita ou disco.
 - Registros são acessados sequencialmente ou em grandes blocos;

Ordenação

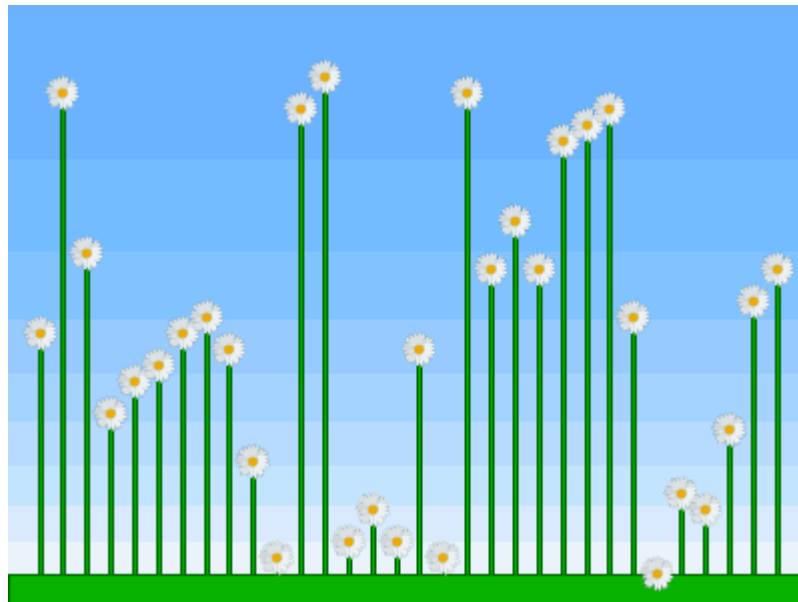
- **Métodos de Ordenação:**
 - Bubble Sort;
 - Selection Sort;
 - Insertion Sort;
 - Merge Sort;
 - Quick Sort;
- Limite assintótico mínimo para algoritmos de ordenação baseados em comparações:

$$O(n \log n)$$

Bubble Sort

- **Algoritmo:**

- Quando dois elementos estão fora de ordem, troque-os de posição até que o i -ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor;
- Continue o processo até que todo o vetor esteja ordenado;



Bubble Sort

16	12	22	23	27	27
---------------	---------------	---------------	---------------	---------------	---------------

6	12	14	14	17	22
---	----	---------------	---------------	----	----

Bubble Sort

6	12	14	8	17	22
---	----	----	---	----	----

6	12	8	14	17	22
---	----	---	----	----	----

6	12	12	14	17	22
---	---------------	---------------	----	----	----

6	8	12	14	17	22
---	---	----	----	----	----

Bubble Sort - Implementação Iterativa (I)

```
void bubblesort(int n, int* v)
{
    int fim, i, temp;
    for (fim = n-1; fim > 0; fim--)
    {
        for (i=0; i<fim; i++)
        {
            if (v[i]>v[i+1])
            {
                temp = v[i];
                v[i] = v[i+1];
                v[i+1] = temp;
            }
        }
    }
}
```

Bubble Sort - Implementação Iterativa (II)

```
void bubblesort(int n, int* v)
{
    int i, fim, troca, temp;
    for (fim = n-1; fim > 0; fim--){
        troca = 0;
        for (i=0; i<fim; i++){
            if (v[i]>v[i+1]){
                temp = v[i];
                v[i] = v[i+1];
                v[i+1] = temp;
                troca = 1;
            }
        }
        if (troca == 0)
            return;
    }
}
```

Implementação mais otimizada:
para a busca quando ocorre
uma passada sem trocas.

Bubble Sort - Complexidade

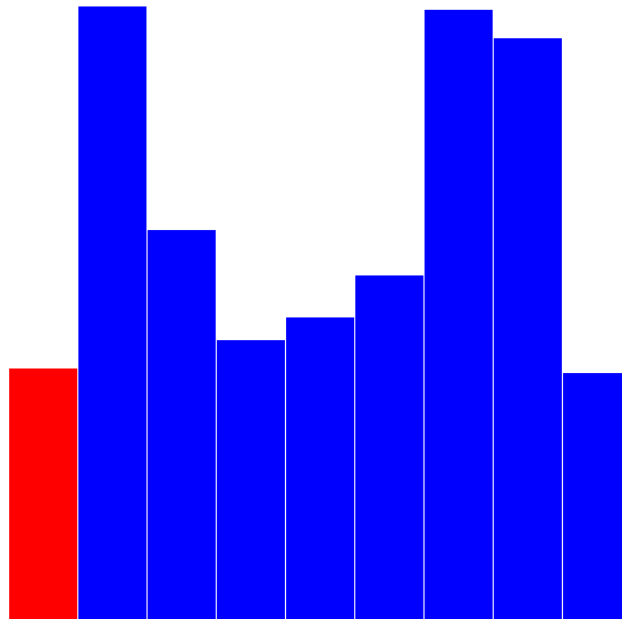
- Esforço computacional \cong número de comparações
 \cong número máximo de trocas
 - primeira passada: $n-1$ comparações
 - segunda passada: $n-2$ comparações
 - terceira passada: $n-3$ comparações
- Tempo total gasto pelo algoritmo:
 - $T(n) = (n-1) + (n-2) + \dots + 2 + 1$
 - $T(n) = \frac{n^2 - n}{2}$
 - Algoritmo de ordem quadrática: **$O(n^2)$**

Bubble Sort

- Complexidade: $O(n^2)$
- Vantagens:
 - Fácil Implementação;
 - É estável;
- Desvantagens:
 - O fato do vetor já estar ordenado não ajuda em nada;
 - Ordem de complexidade quadrática;

Selection Sort

- **Algoritmo:**
 - Em cada etapa, em selecionar o maior (ou o menor) elemento e colocá-lo em sua posição correta dentro da futura lista ordenada.



Selection Sort

8	5	7	1	9	3
---	---	---	---	---	---

1	5	7	8	9	3
---	---	---	---	---	---

1	3	7	8	9	5
---	---	---	---	---	---

1	3	5	8	9	7
---	---	---	---	---	---

1	3	5	7	9	8
---	---	---	---	---	---

1	3	5	7	8	9
---	---	---	---	---	---


Selection Sort - Implementação

```
void selectSort(int arr[], int n)
{
    int pos_min, temp, i, j;
    for (i=0; i < n-1; i++){
        pos_min = i;
        for (j=i+1; j < n; j++){
            if (arr[j] < arr[pos_min])
                pos_min=j;
        }
        if (pos_min != i){
            temp = arr[i];
            arr[i] = arr[pos_min];
            arr[pos_min] = temp;
        }
    }
}
```


Selection Sort – Complexidade

- Esforço computacional \cong número de comparações
 \cong número máximo de trocas
- Tempo total gasto pelo algoritmo:
 - $T(n) = (n-1) + (n-2) + \dots + 2 + 1$
 - $T(n) = \frac{n^2 - n}{2}$
 - Algoritmo de ordem quadrática: **$O(n^2)$**

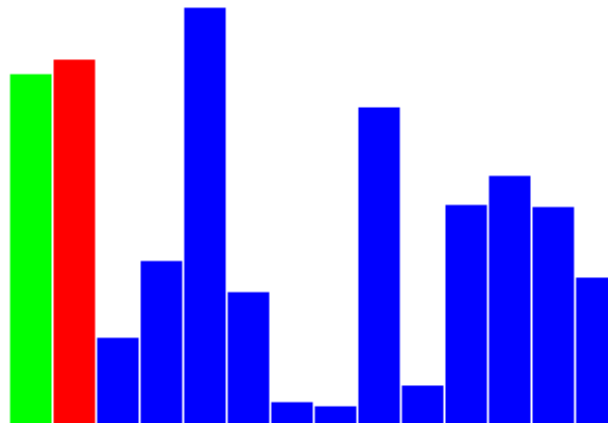
Selection Sort

- Complexidade: $O(n^2)$
 - Vantagens:
 - Fácil Implementação;
 - Pequeno número de movimentações;
 - Interessante para arquivos pequenos;
 - Desvantagens:
 - O fato de o arquivo já estar ordenado não influencia em nada;
 - Ordem de complexidade quadrática;
 - Algoritmo não estável;
- 

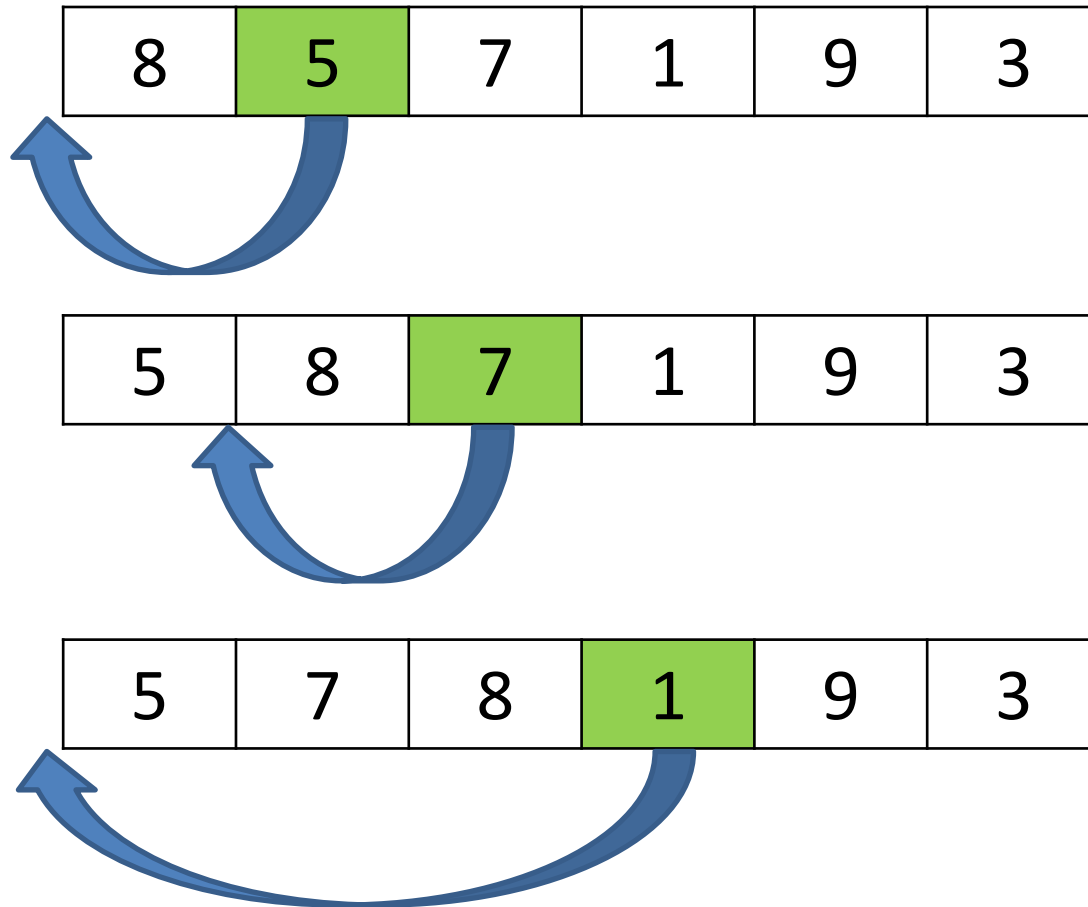
Insertion Sort

- **Algoritmo:**

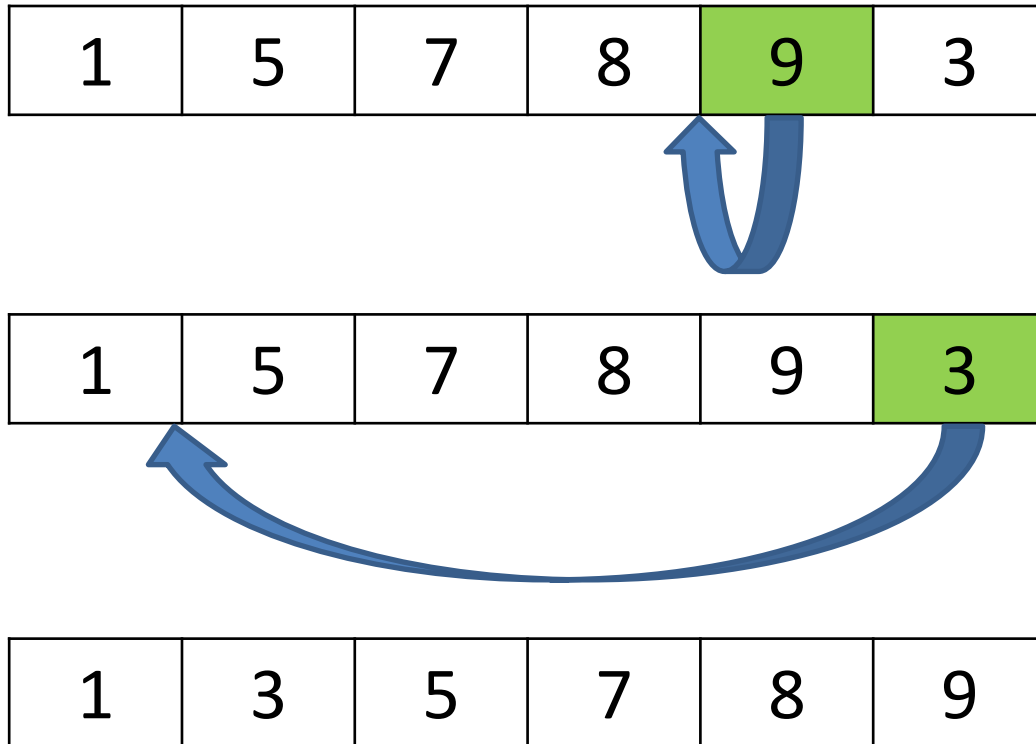
- Em cada passo, a partir de $i=2$, o i -ésimo item da sequência fonte é apanhado e transferido para a sequência destino, sendo colocado na posição correta.
- A inserção do elemento no lugar de destino é efetuado movendo-se os itens com chave maiores para a direita e então inserindo-o na posição que ficou vazia.



Insertion Sort



Insertion Sort



Insertion Sort – Implementação

```
void insertion_sort (int arr[], int length)
{
    int j, i, temp;
    for (i = 1; i < length; i++)
    {
        j = i;
        while (j > 0 && arr[j] < arr[j-1])
        {
            temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
            j--;
        }
    }
}
```

Insertion Sort – Complexidade

- Esforço computacional \cong número de comparações
 \cong número máximo de trocas
- Tempo total gasto pelo algoritmo:
 - $T(n) = (n-1) + (n-2) + \dots + 2 + 1$
 - $T(n) = \frac{n^2 - n}{2}$
 - Algoritmo de ordem quadrática: **$O(n^2)$**

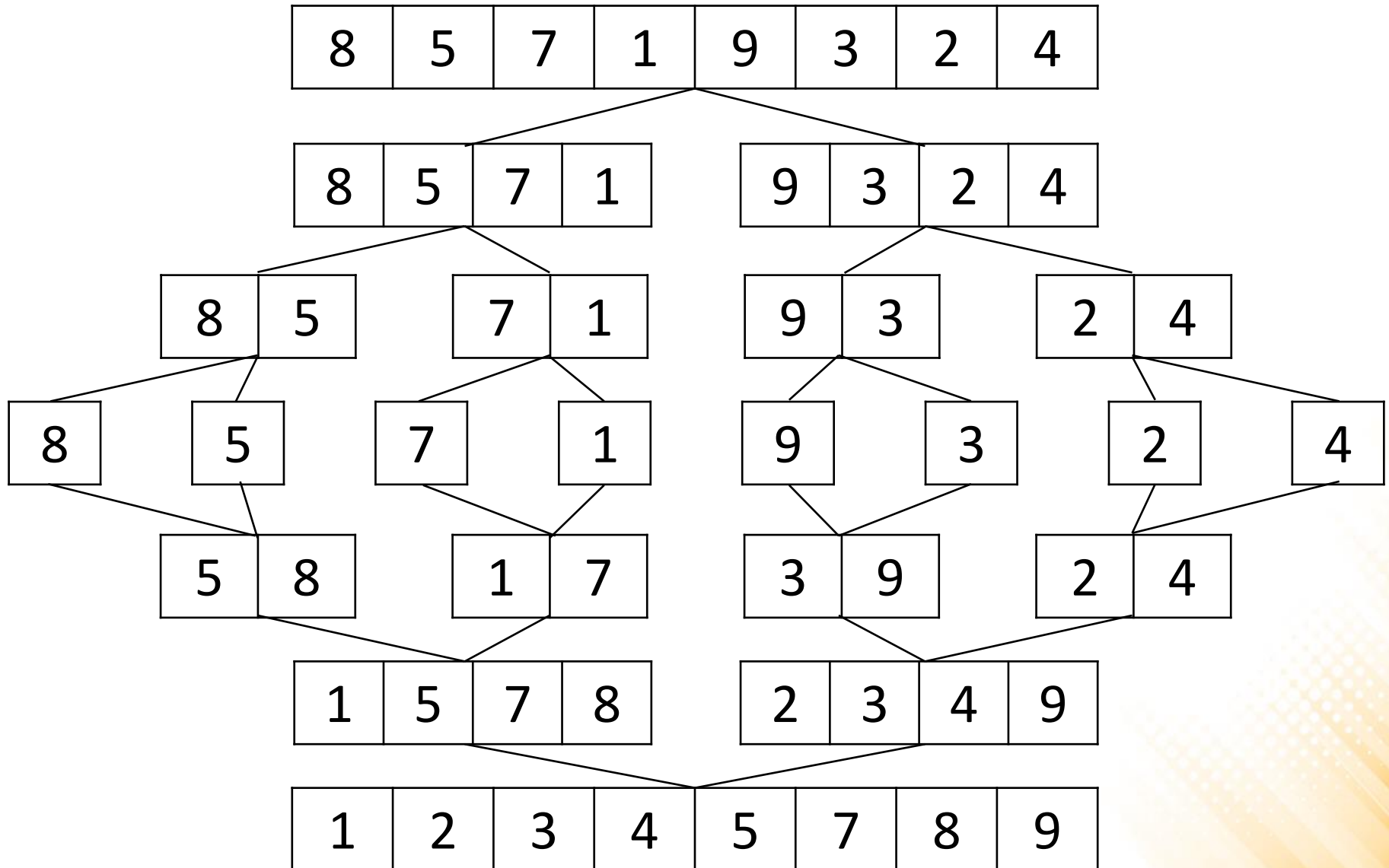
Insertion Sort

- Complexidade: $O(n^2)$
- Vantagens:
 - Fácil Implementação;
 - Algoritmo estável;
 - O vetor já ordenado favorece a ordenação: $O(n)$;
- Desvantagens:
 - Ordem de complexidade quadrática;
 - Ineficiente quando o vetor está ordenado inversamente (algo natural em um problema de ordenação);

Merge Sort

- **Algoritmo:**
 - Criar uma sequência ordenada a partir de duas outras também ordenadas.
 - Para isso, divide-se a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes.
- É um algoritmo de ordenação do tipo dividir para conquistar:
 - **Dividir:** Dividir os dados em subsequências pequenas;
 - **Conquistar:** Classificar as duas metades recursivamente aplicando o merge sort;
 - **Combinar:** Juntar as duas metades em um único conjunto já classificado.

Merge Sort



Merge Sort – Implementação

```
void mergeSort_ordena(int *v, int esq, int dir)
{
    if (esq == dir)
        return;

    int meio = (esq+dir) / 2;
    mergeSort_ordena(v, esq, meio);
    mergeSort_ordena(v, meio+1, dir);
    mergeSort_intercala(v, esq, meio, dir);
}
```

Merge Sort – Implementação

```
void mergeSort_intercala(int* v, int esq, int meio, int dir)
{
    int i, j, k;
    int a_tam = meio - esq + 1;
    int b_tam = dir - meio;
    int *a = (int*)malloc(sizeof(int) * a_tam);
    int *b = (int*)malloc(sizeof(int) * b_tam);
    for(i = 0; i < a_tam; i++)
    {
        a[i] = v[i+esq];
    }
    for(i = 0; i < b_tam; i++)
    {
        b[i] = v[i+meio+1];
    }
}
```

[continua...]

Merge Sort – Implementação

[... continuação]

```
...
for(i = 0, j = 0, k = esq; k <= dir; k++)
{
    if(i == a_tam)
        v[k] = b[j++];
    else if (j == b_tam)
        v[k] = a[i++];
    else if (a[i] < b[j])
        v[k] = a[i++];
    else
        v[k] = b[j++];
}
free(a);
free(b);
}
```

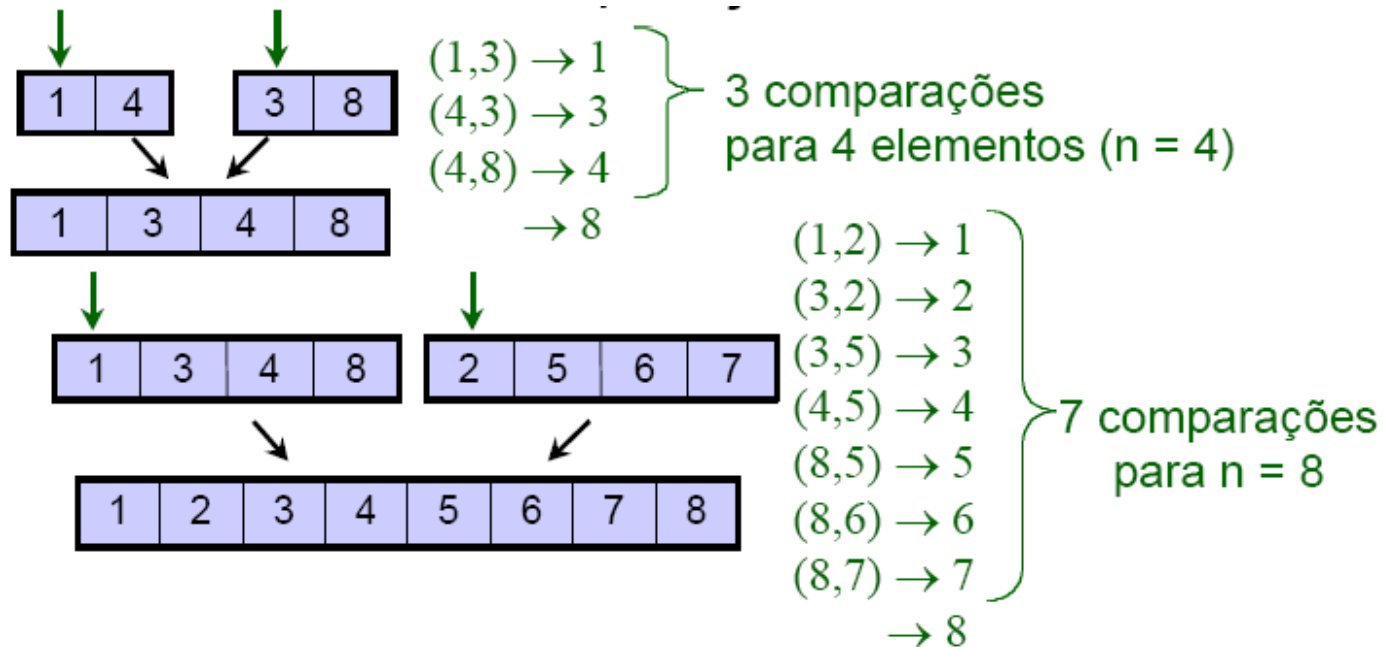
Merge Sort – Complexidade

- Tempo gasto para dividir: $O(\log n)$

Repetição	Tamanho do Problema
1	n
2	$n/2$
3	$n/4$
...	...
$\log n$	1

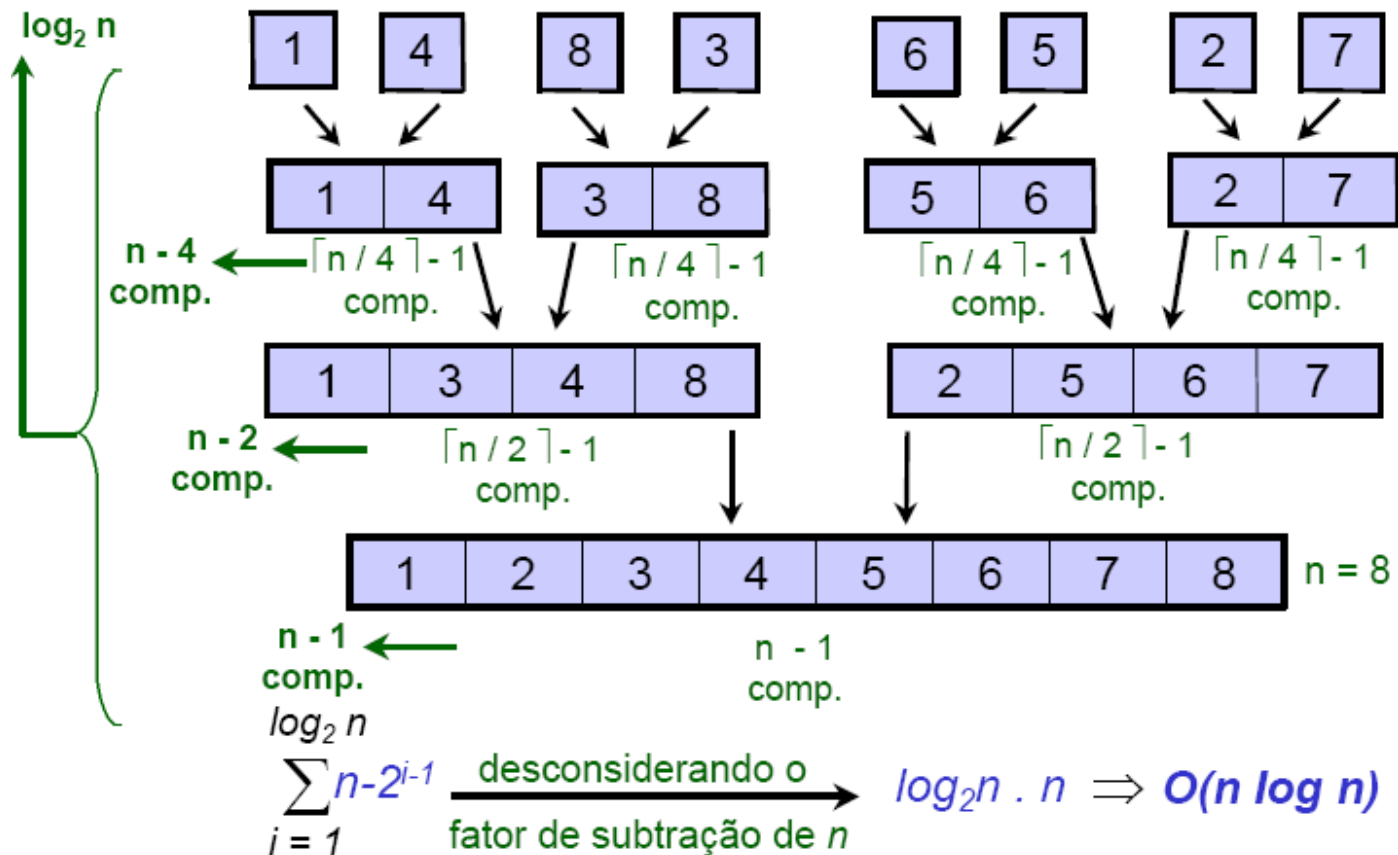
Merge Sort – Complexidade

- Tempo gasto para combinar: $O(n)$



Merge Sort – Complexidade

- Tempo total gasto pelo algoritmo: $O(n \log n)$



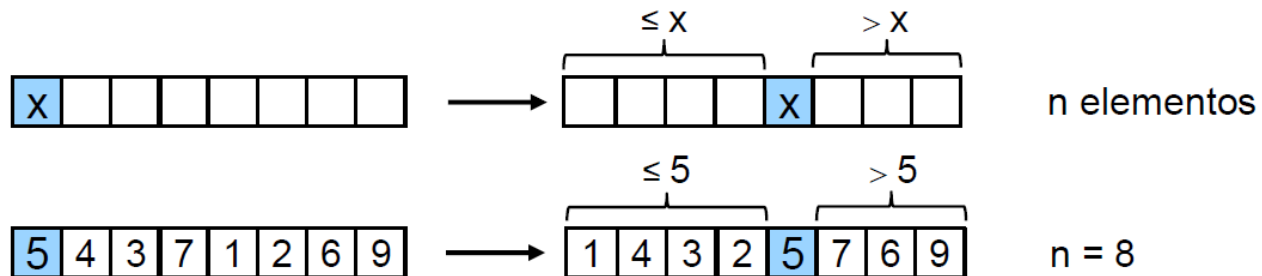
Merge Sort

- Complexidade: $O(n \log n)$
- Vantagens:
 - Complexidade $O(n \log n)$;
 - É estável;
- Desvantagens:
 - Tradicionalmente baseia-se em chamadas recursivas, mas é possível implementá-lo sem utilizar recursão;
 - Utiliza memória auxiliar;

Quick Sort

- **Algoritmo:**

1. Escolha um elemento arbitrário x , o **pivô**;
2. **Particione** o vetor de tal forma que x fique na posição correta $v[i]$:
 - x deve ocupar a posição i do vetor sse:
 - todos os elementos $v[0], \dots, v[i-1]$ são menores que x ;
 - todos os elementos $v[i+1], \dots, v[n-1]$ são maiores que x ;



3. Chame **recursivamente** o algoritmo para ordenar os subvetores $v[0], \dots, v[i-1]$ e $v[i+1], \dots, v[n-1]$ (vetor da esquerda e vetor da direita)
 - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento

Quick Sort

quicksort do vetor de tamanho n

se $n > 1$ então

PARTIÇÃO com pivô x

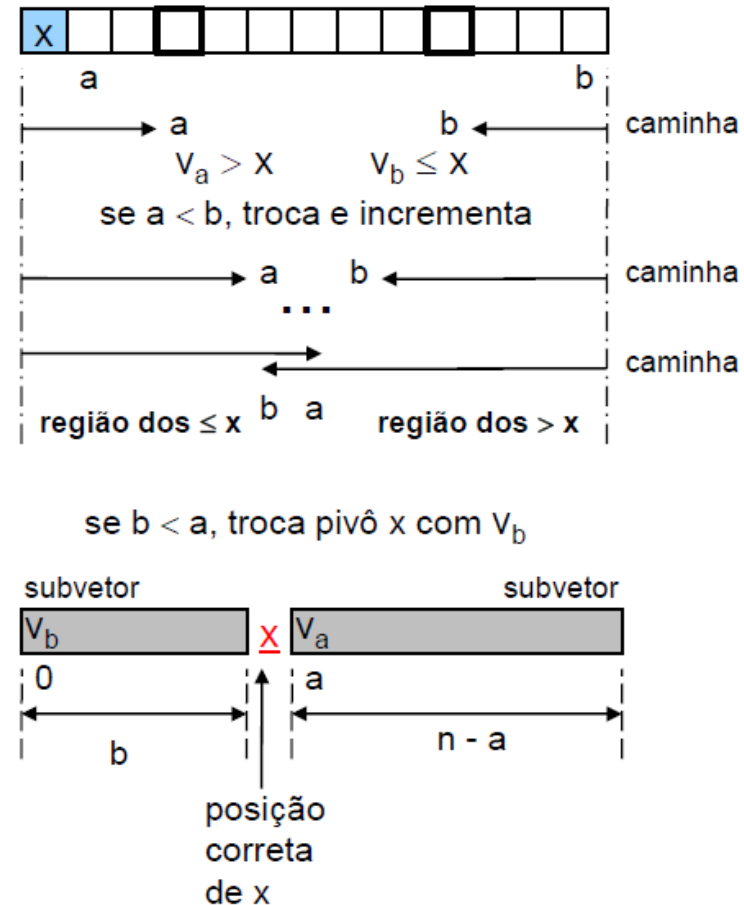
quicksort do subvetor à esquerda de x

quicksort do subvetor à direita de x

Quick Sort

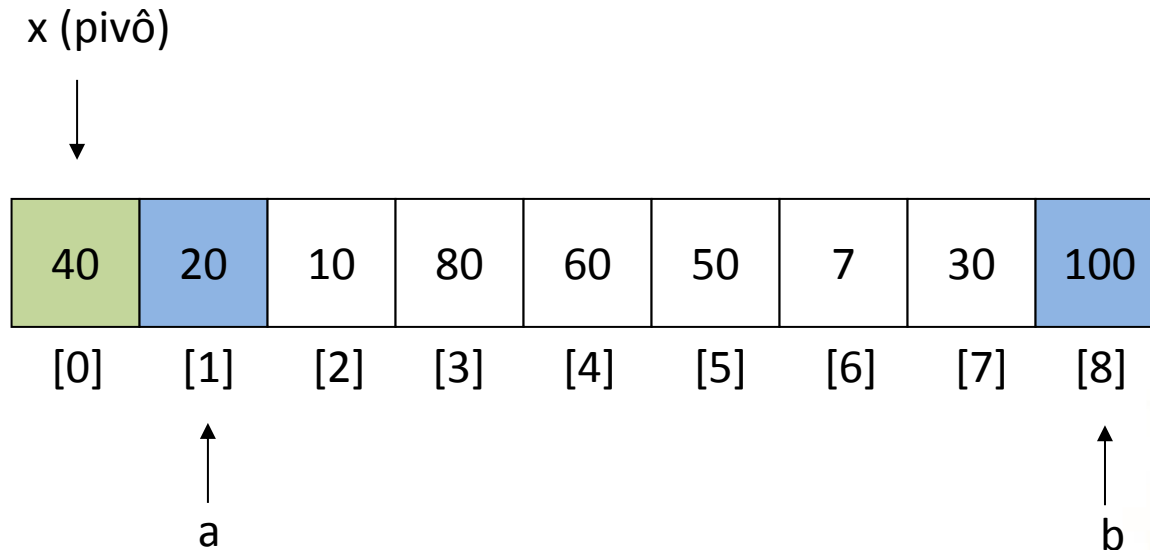
- **Processo de partição:**

1. Caminhe com o índice **a** do início para o final, comparando x com $v[1]$, $v[2]$, ... até encontrar $v[a] > x$
2. Caminhe com o índice **b** do final para o início, comparando x com $v[n-1]$, $v[n-2]$, ... até encontrar $v[b] \leq x$
3. Troque $v[a]$ e $v[b]$
4. Continue para o final a partir de $v[a+1]$ e para o início a partir de $v[b-1]$
5. Termine quando os índices de busca (**a** e **b**) se encontram (**b** < **a**)
 - A posição correta de $x=v[0]$ é a posição **b**, então $v[0]$ e $v[b]$ são trocados

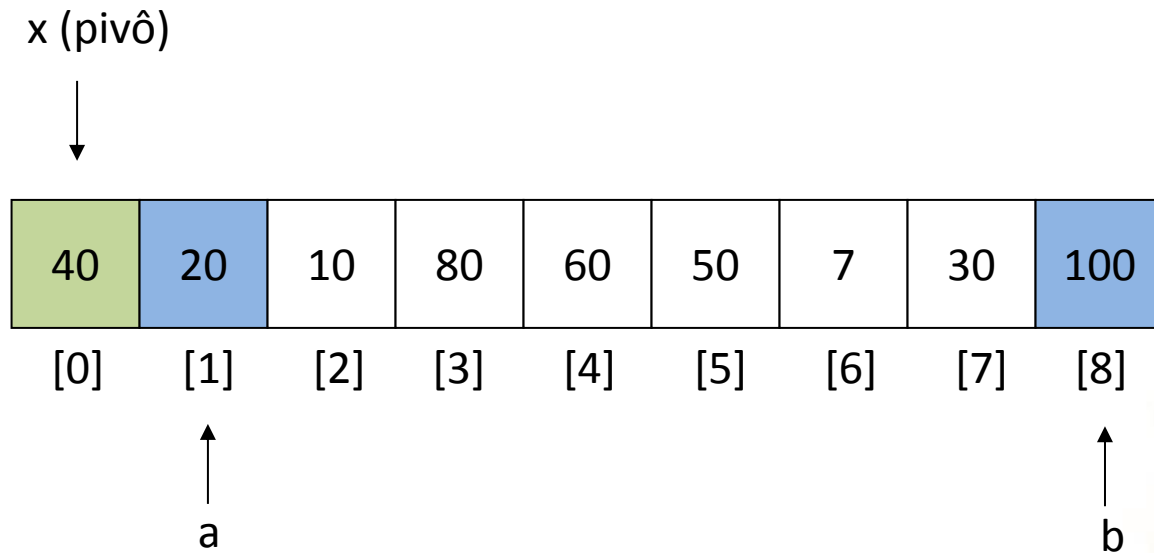


Quick Sort

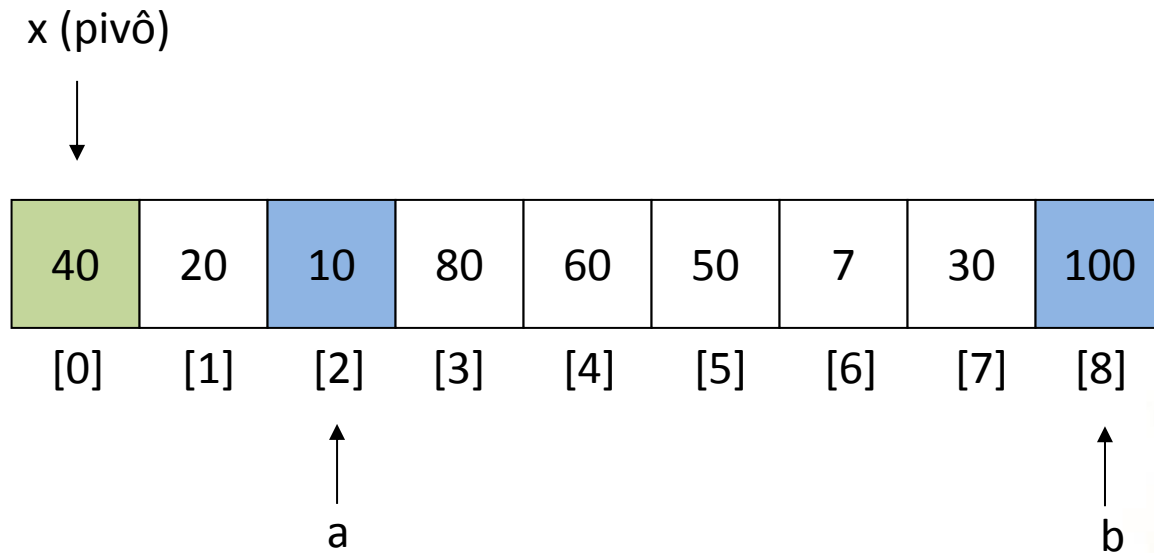
- **Processo de Partição:**
 - Exemplo:



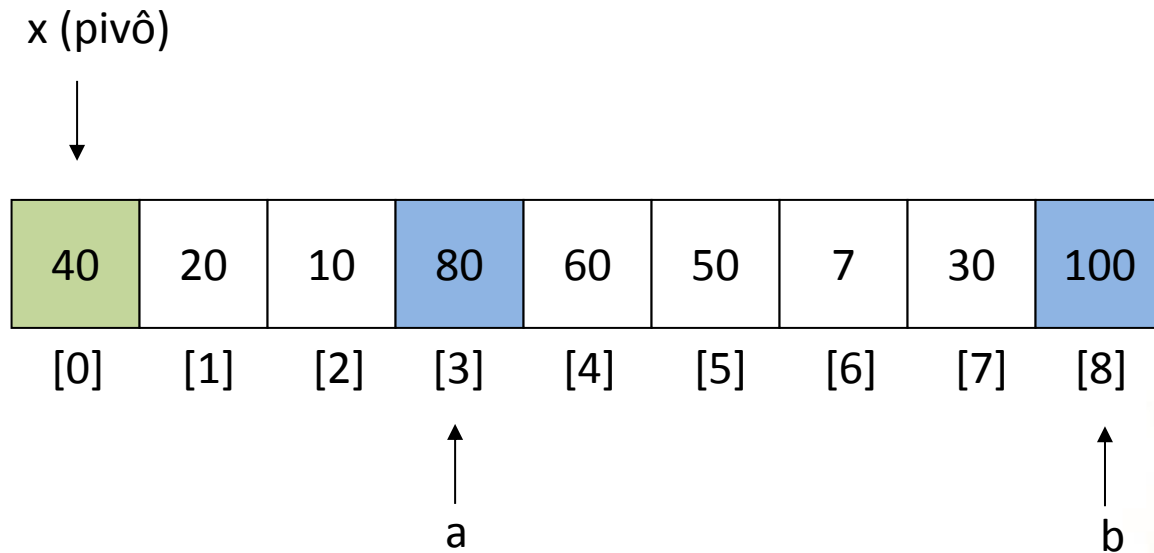
```
do{  
  while (a < n && v[a] <= x)  
    a++;
```



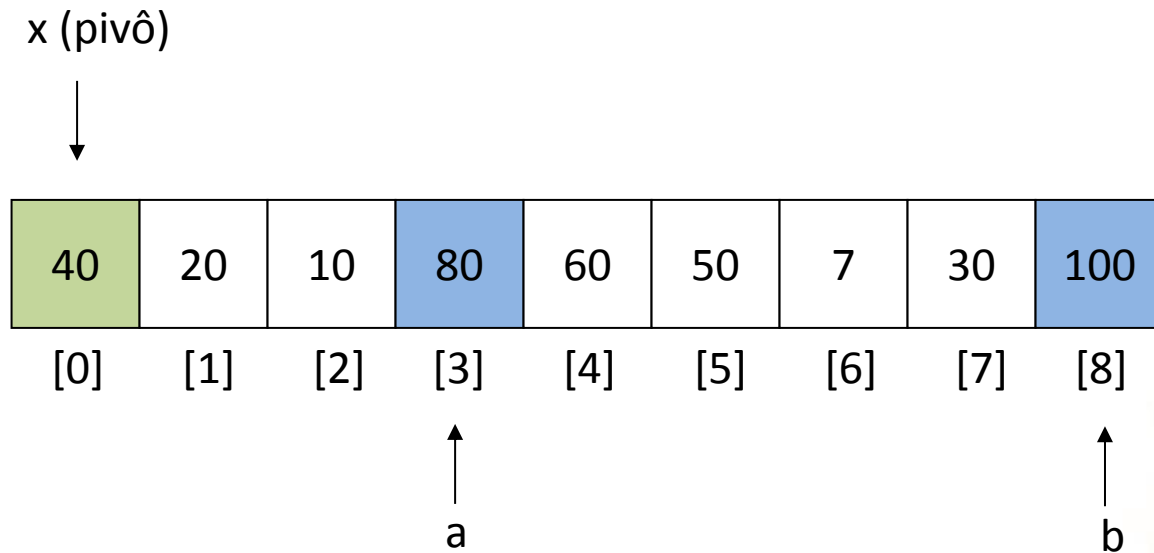
```
do{  
  while (a < n && v[a] <= x)  
    a++;
```



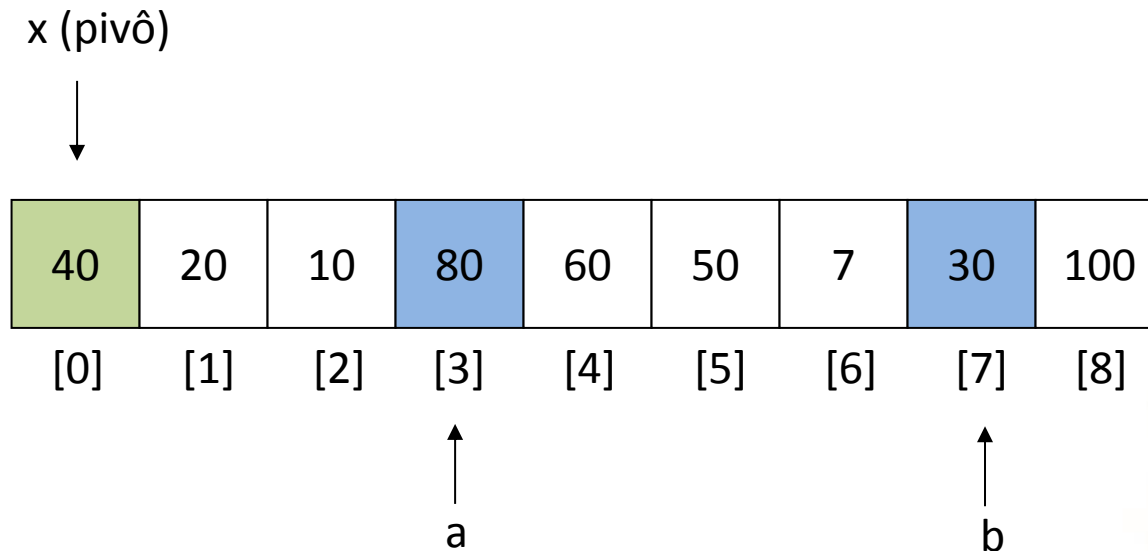
```
do{  
  while (a < n && v[a] <= x)  
    a++;
```



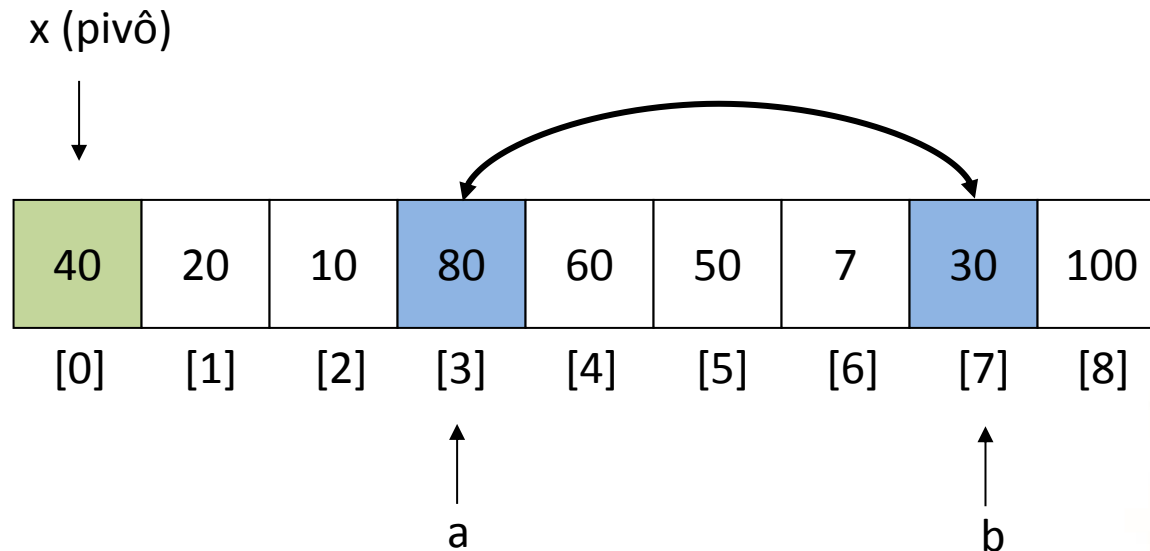

```
do{  
  while (a < n && v[a] <= x)  
    a++;  
  while (v[b] > x)  
    b--;
```



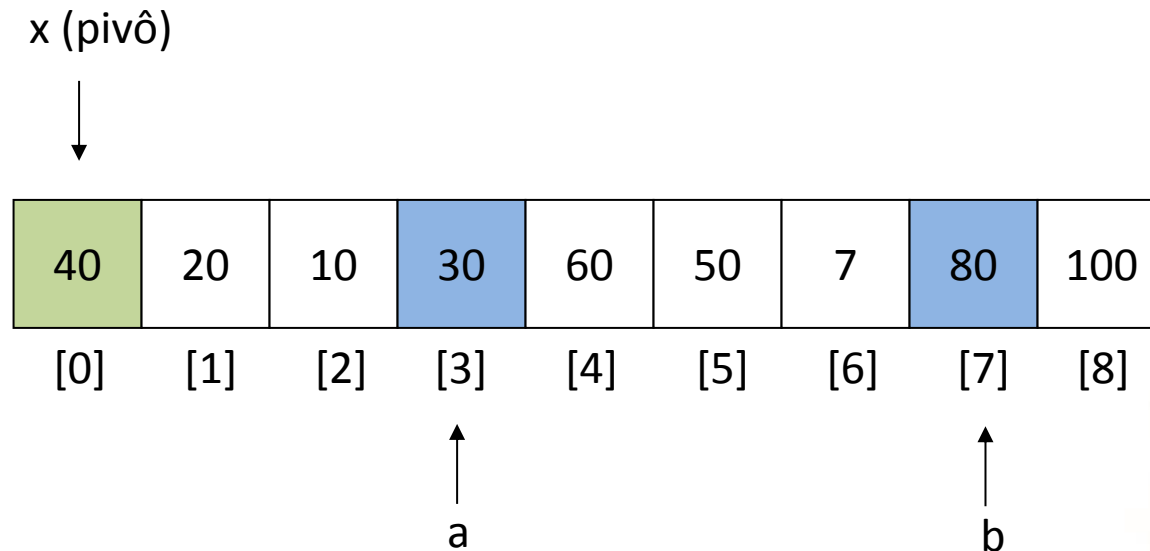
```
do{  
  while (a < n && v[a] <= x)  
    a++;  
  while (v[b] > x)  
    b--;
```



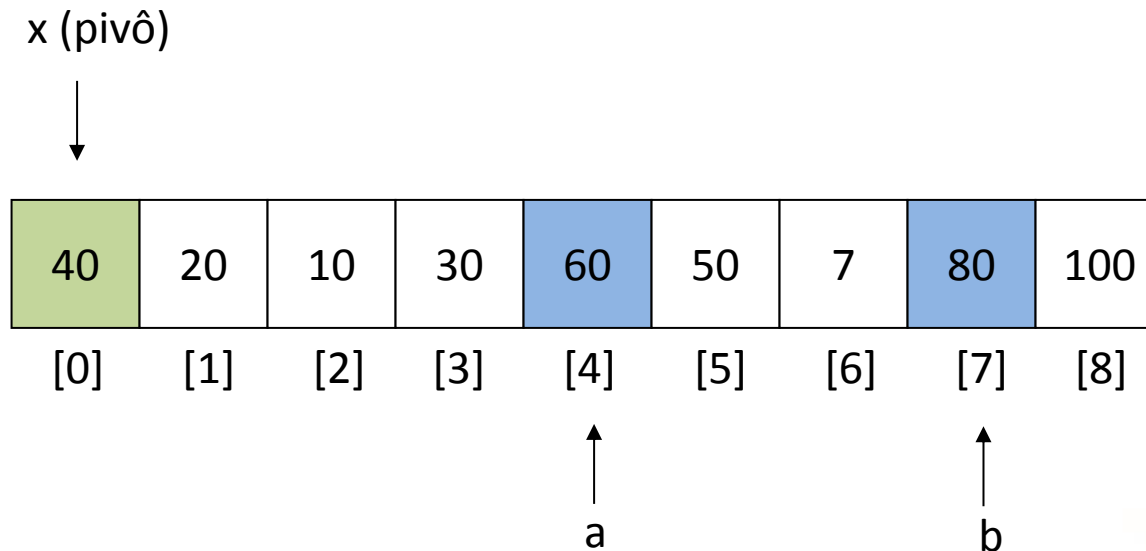
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
}
```



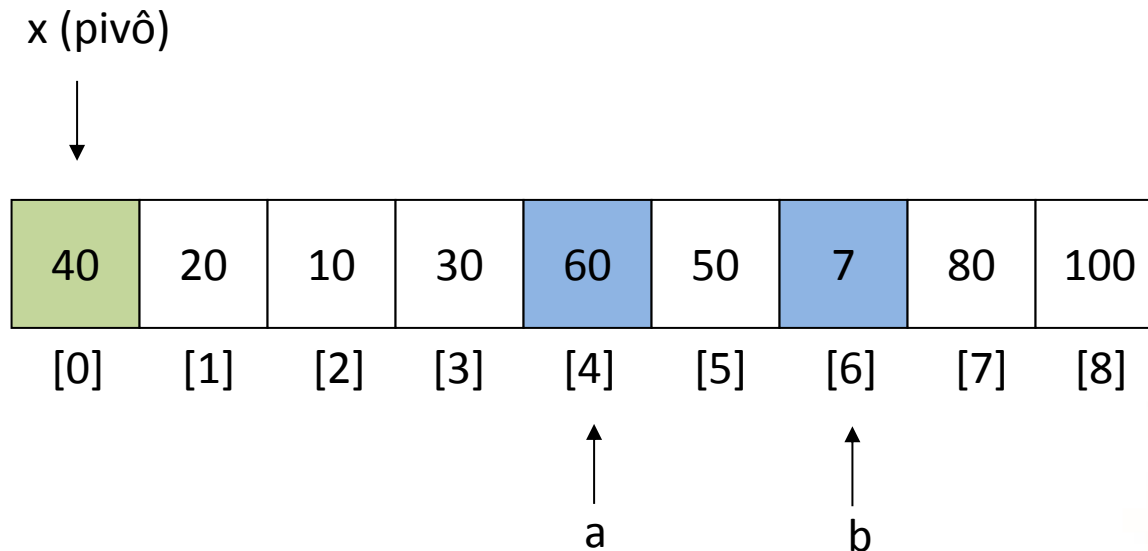
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



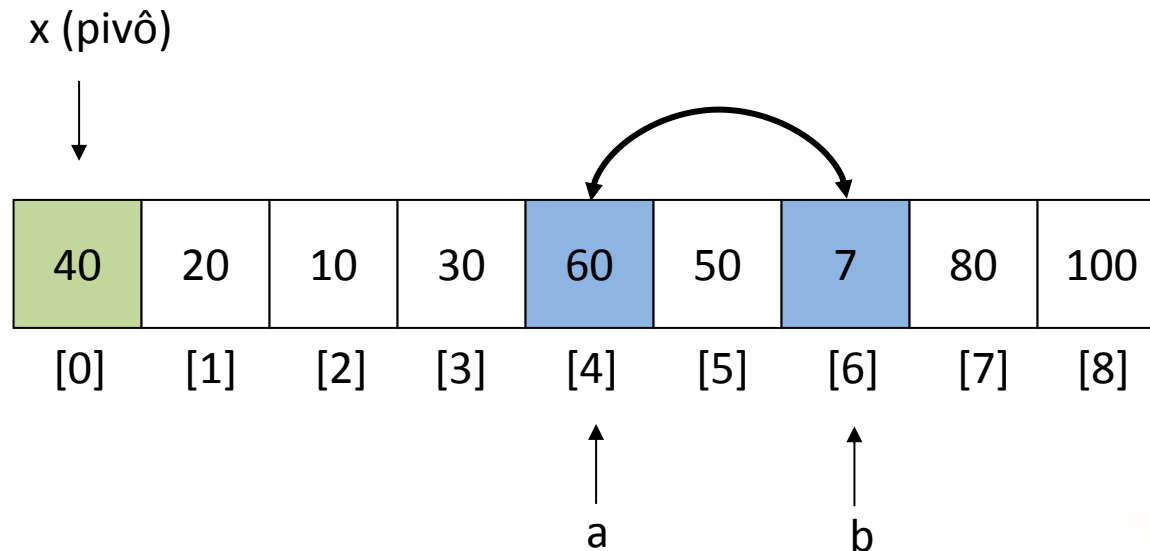
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



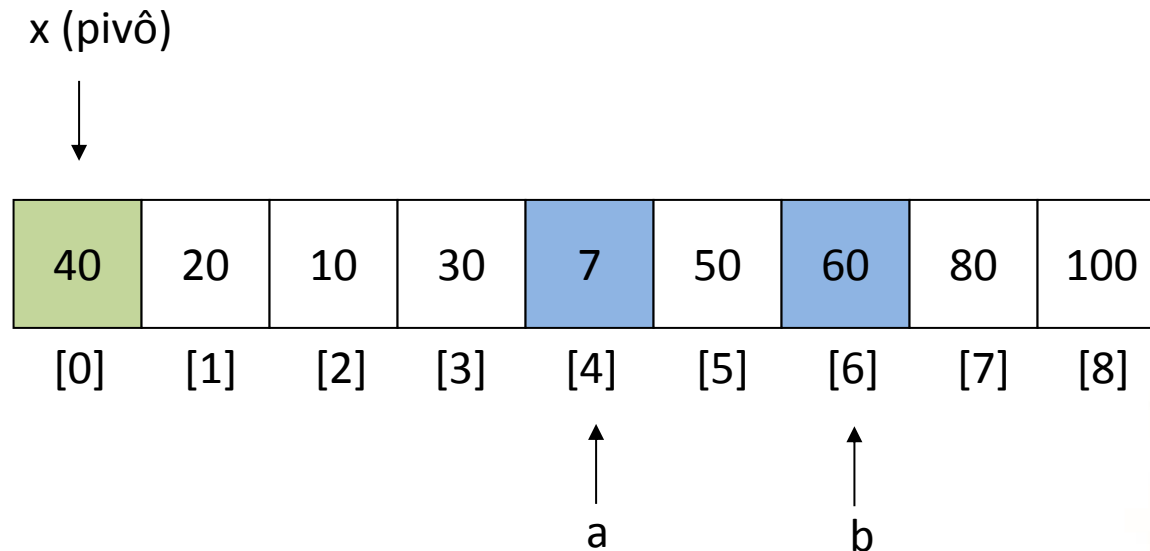
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



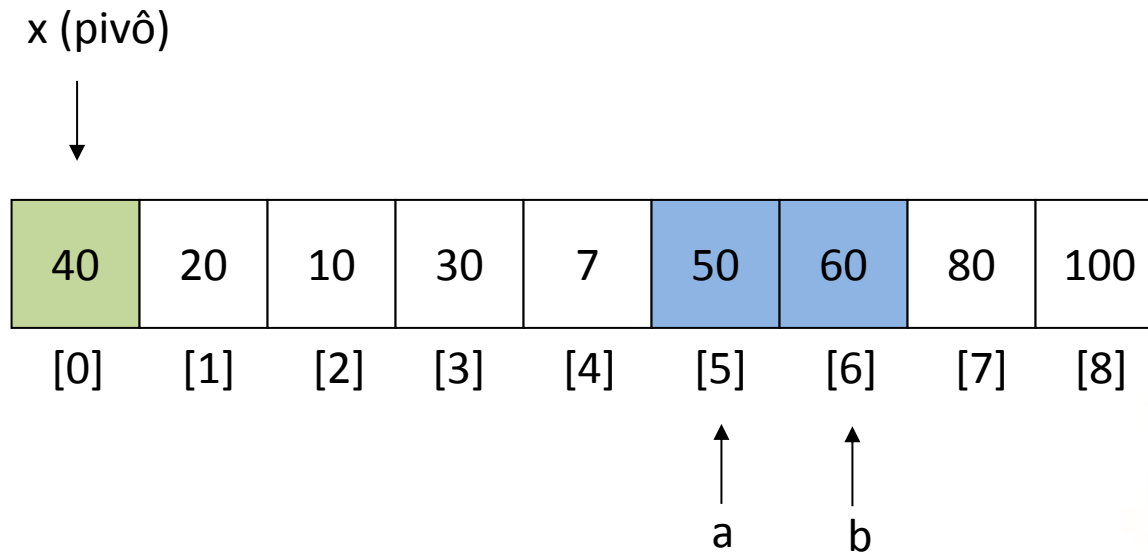
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



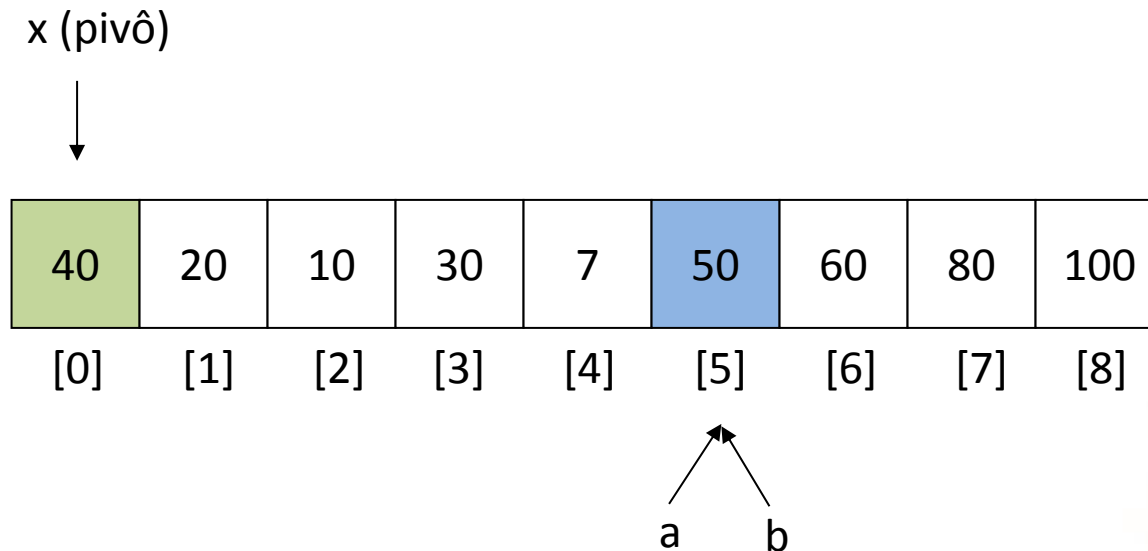
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



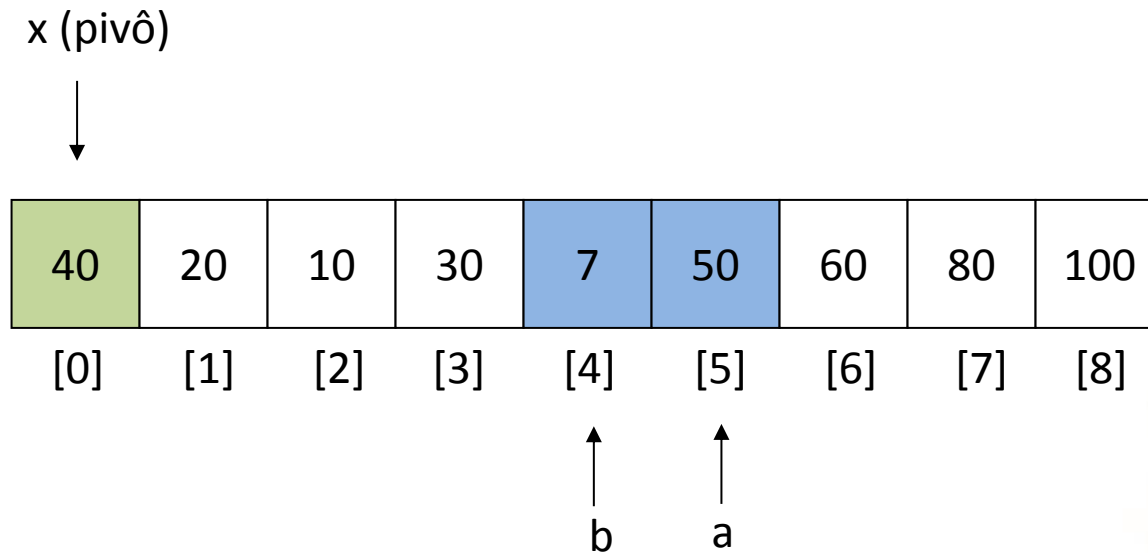

```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



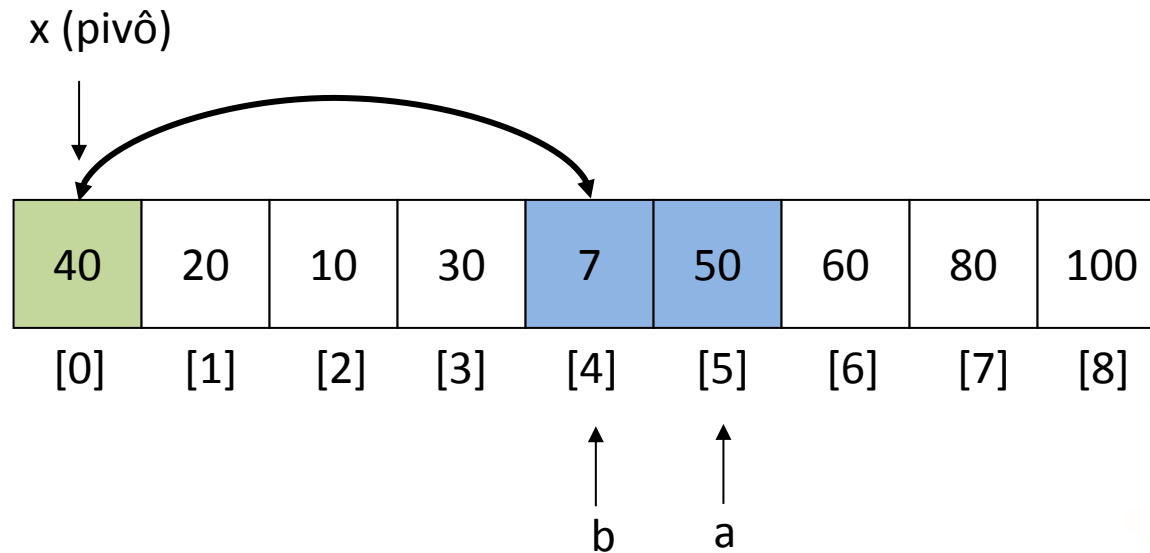
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



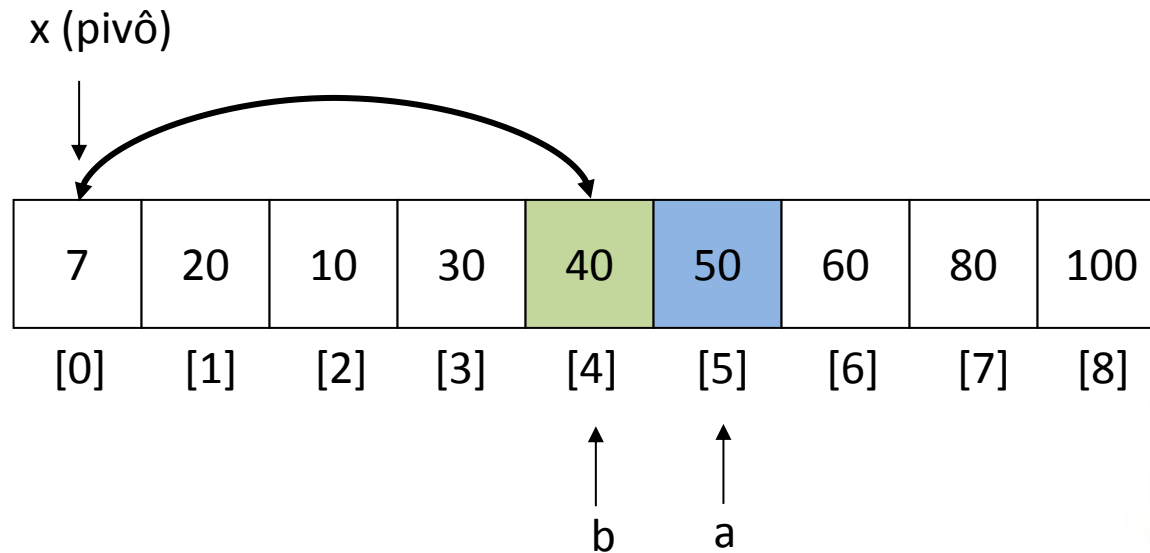
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
troca(&v[x], &v[b]);
```



```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
troca (&v[x], &v[b]);
```



Quick Sort

- Resultado da Partição:

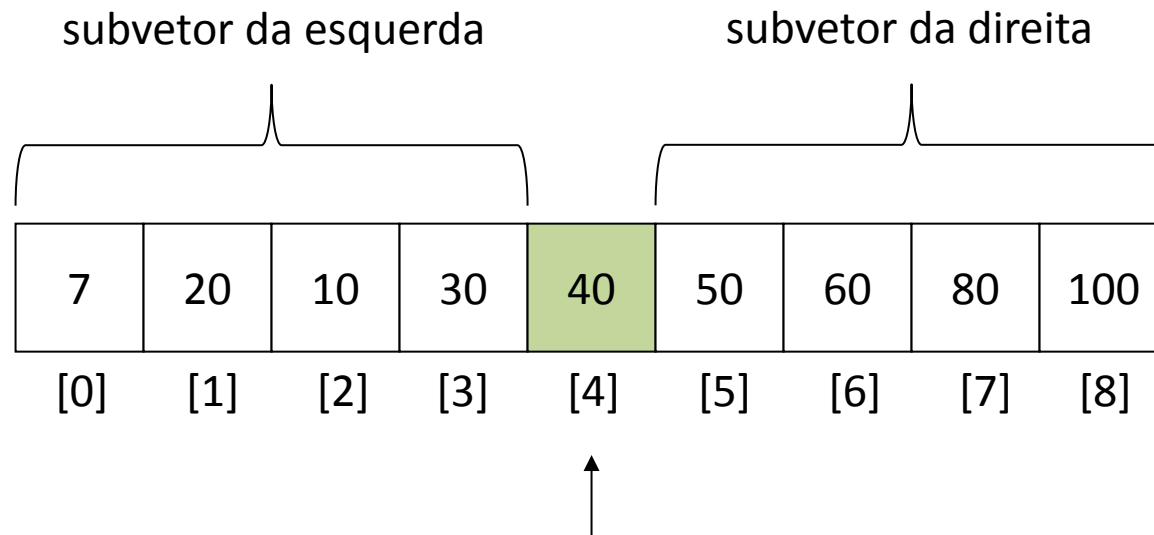
7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]



o pivô ocupa a sua posição
final na ordenação

Quick Sort

- Chamada recursiva aos subvetores:



o pivô ocupa a sua posição
final na ordenação

```
void quicksort(int n, int* v){
    int x, a, b, temp;
    if (n > 1) {
        x = v[0]; a = 1; b = n-1;
        do {
            while (a < n && v[a] <= x)
                a++;
            while (v[b] > x)
                b--;
            if (a < b) {
                temp = v[a];
                v[a] = v[b];
                v[b] = temp;
                a++; b--;
            }
        } while (a <= b);
        v[0] = v[b];
        v[b] = x;
        quicksort(b, v);
        quicksort(n-a, &v[a]);
    }
}
```

Caminha com o índice a

Caminha com o índice b

Faz a troca de a e b

Faz a troca do pivô e b

**Chamada recursiva
para o subvetor da
esquerda e da direita**

Quick Sort

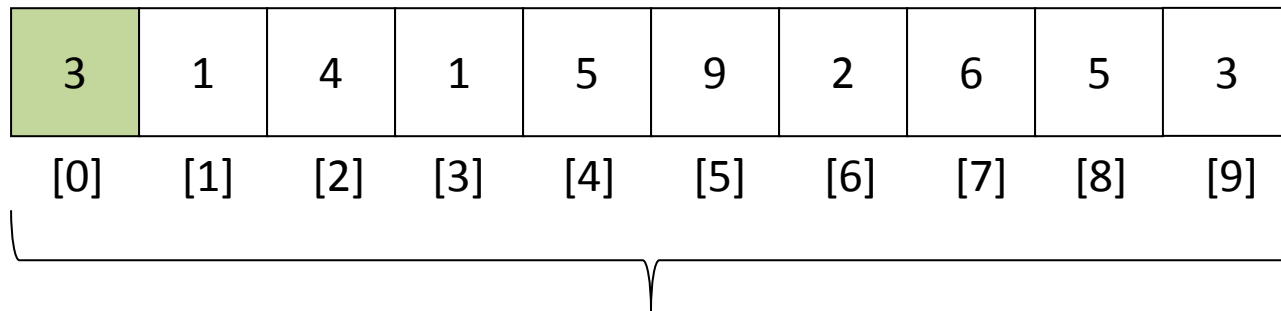
(1) Vetor para ser ordenado:

3	1	4	1	5	9	2	6	5	3
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Quick Sort

(2) Selezione o pivô:

3	1	4	1	5	9	2	6	5	3
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]



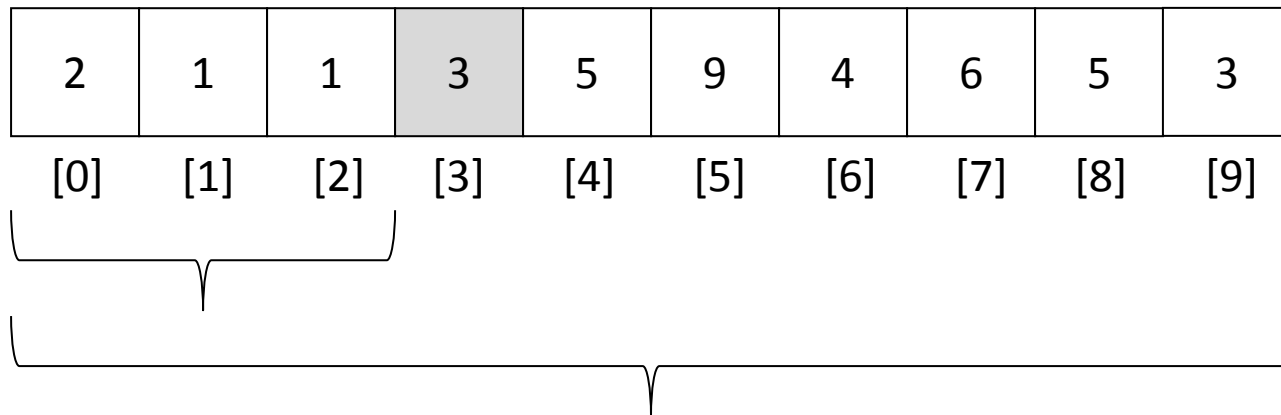
Quick Sort

(3) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:

2	1	1	3	5	9	4	6	5	3
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

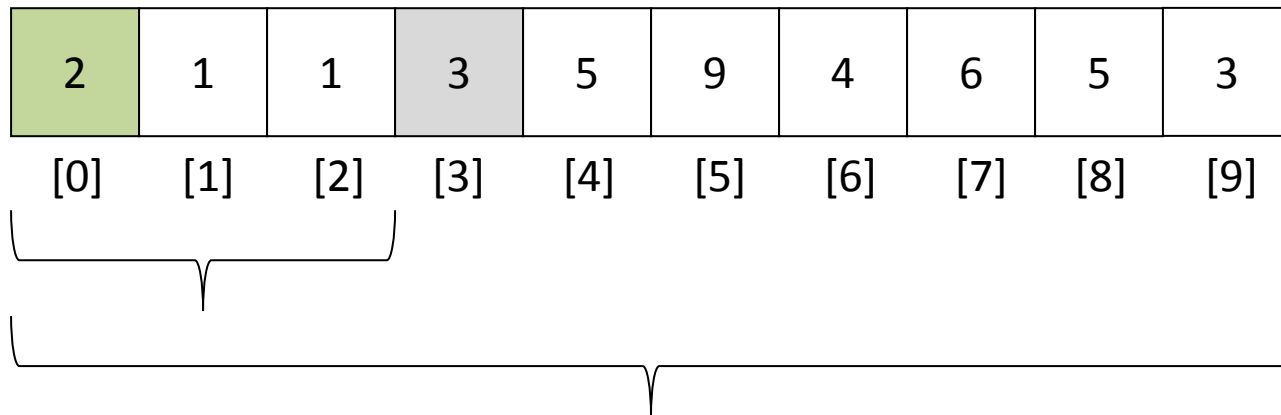
Quick Sort

(4) Chame recursivamente o quick sort para o subvetor da esquerda:



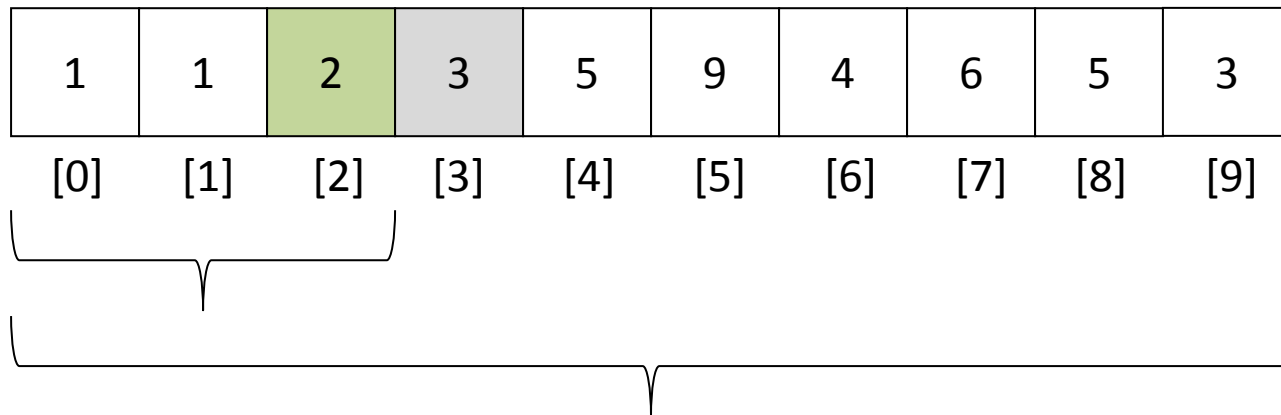
Quick Sort

(5) Selezione o pivô:



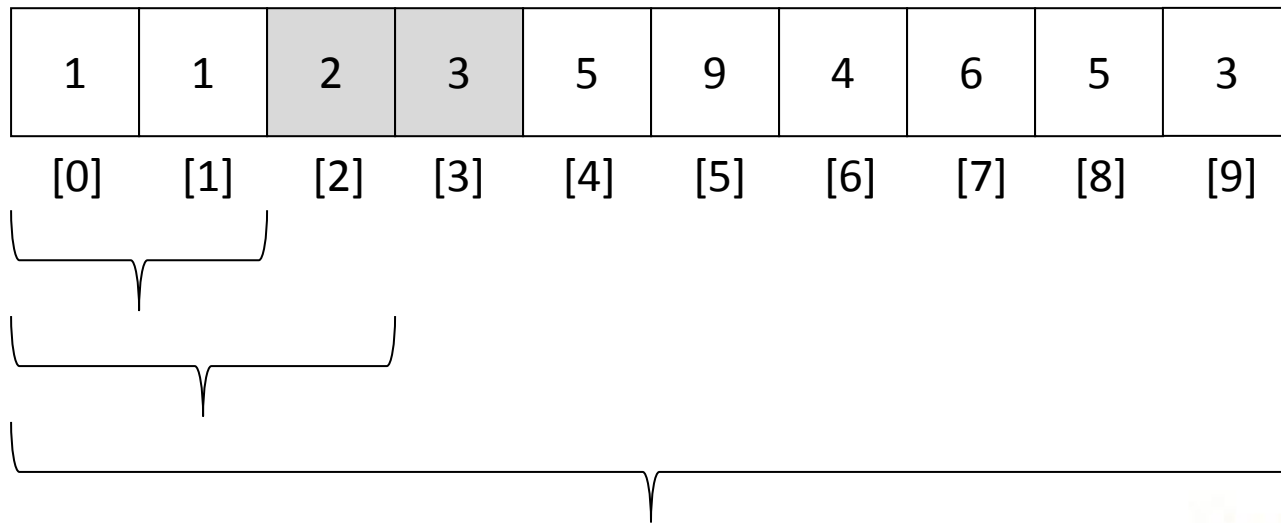
Quick Sort

(6) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



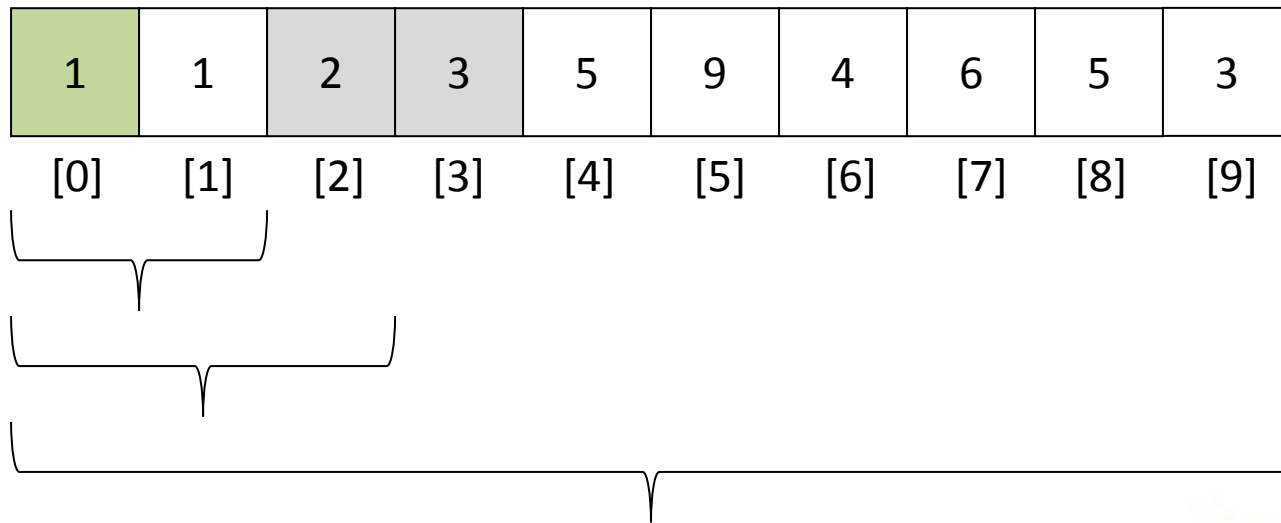
Quick Sort

(7) Chame recursivamente o quick sort para o subvetor da esquerda:



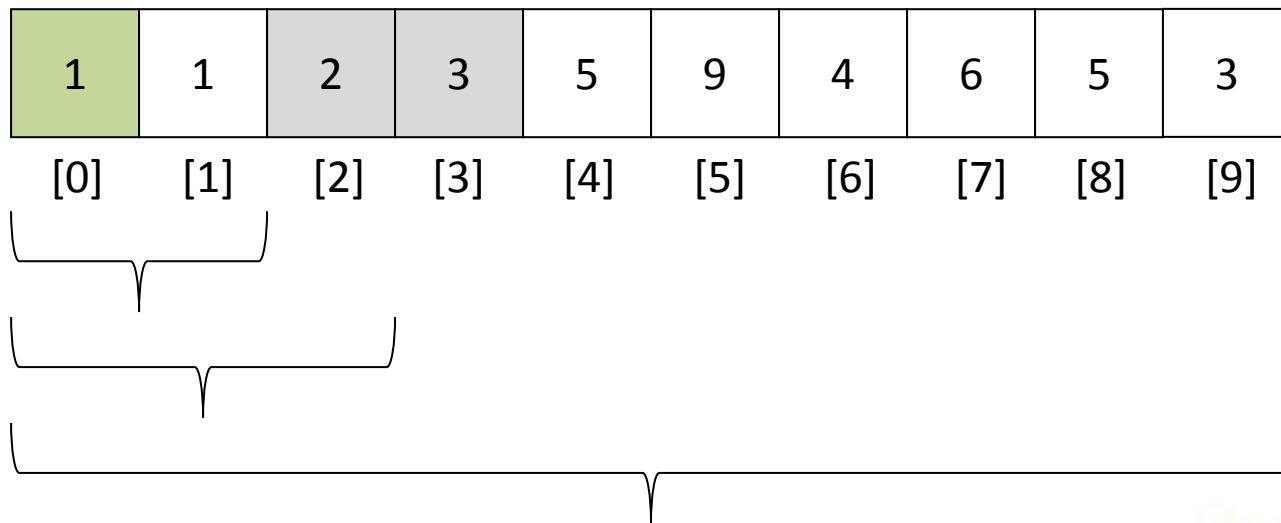
Quick Sort

(8) Selezione o pivô:



Quick Sort

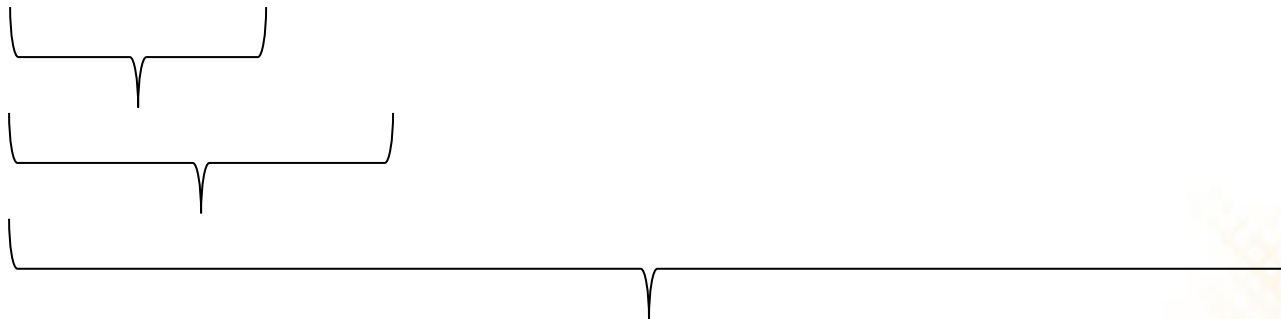
(9) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



Quick Sort

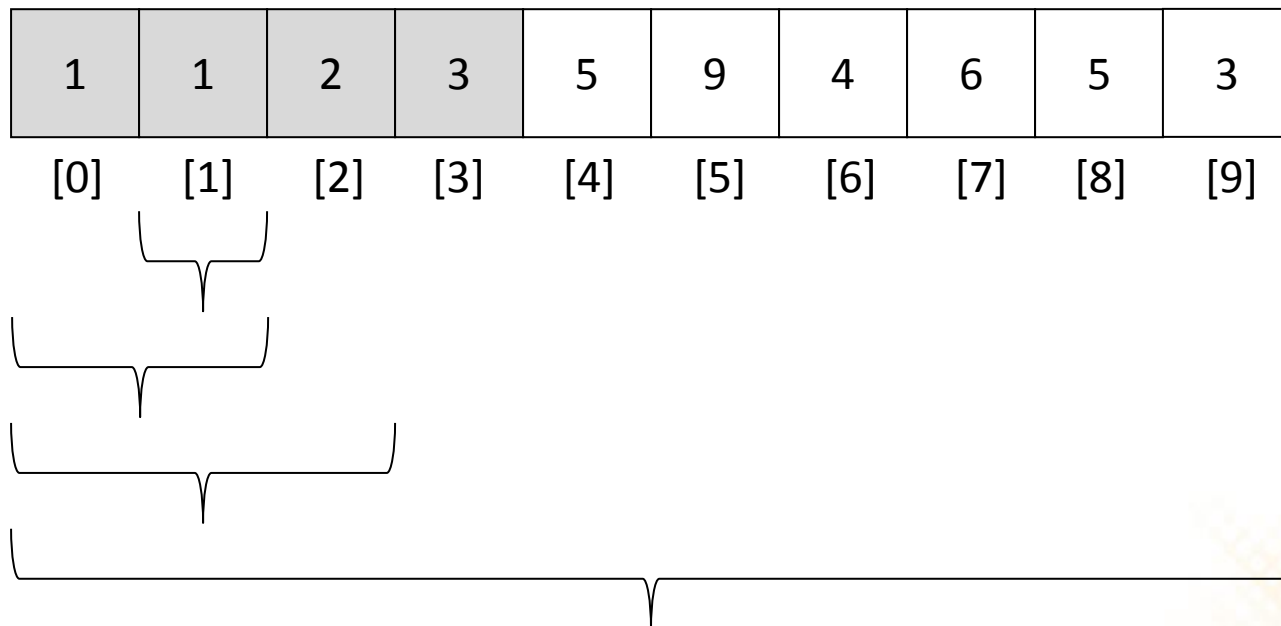
(10) Chame recursivamente o quick sort para o subvetor da esquerda. O tamanho do vetor da esquerda é zero. A recursão retorna.

1	1	2	3	5	9	4	6	5	3
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]



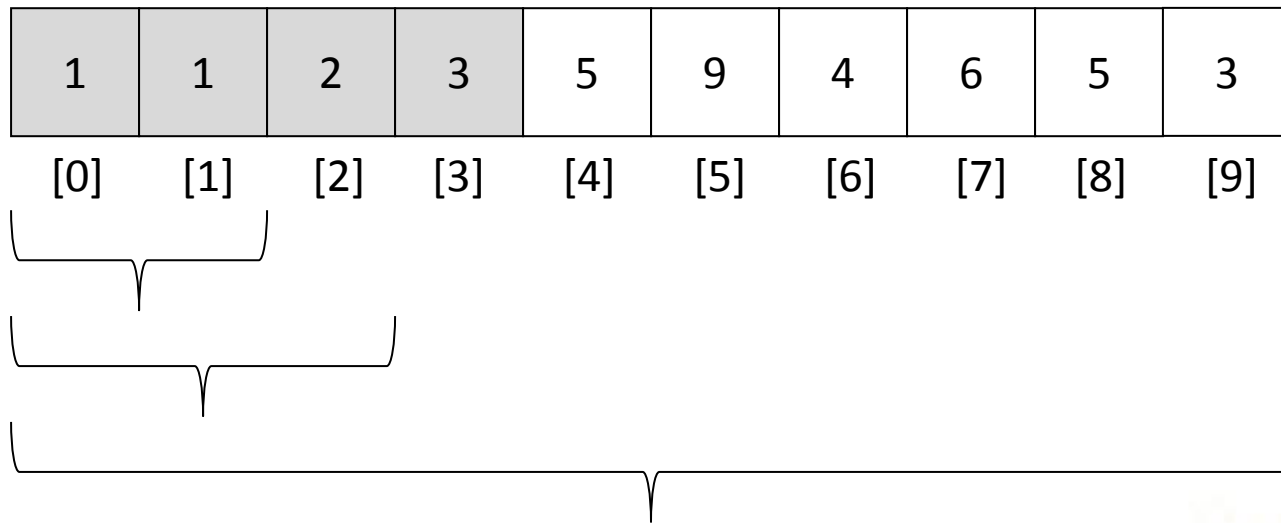
Quick Sort

(11) Chame recursivamente o quick sort para o subvetor da direita. Só existe um elemento, então ele já está ordenado e a recursão retorna.



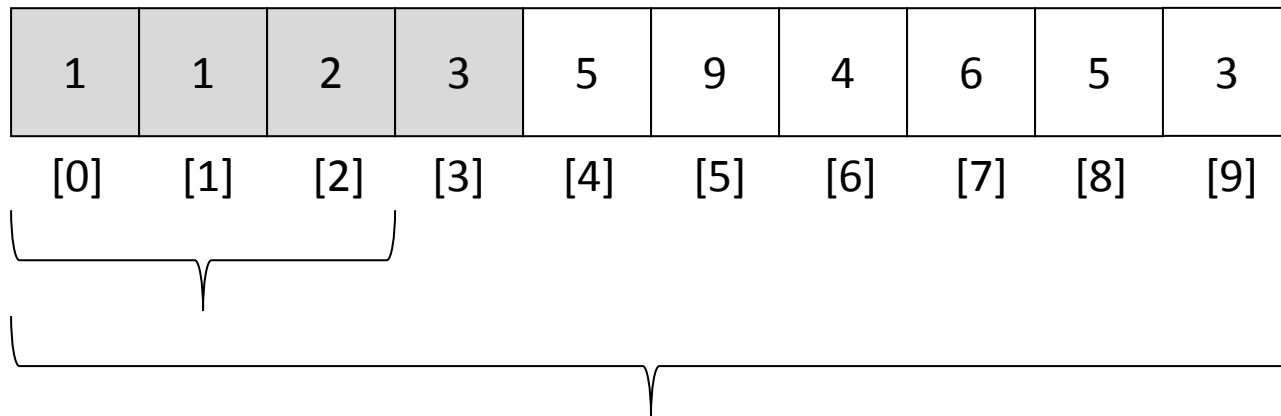
Quick Sort

(12) A recursão retorna. Não tem nada mais para ser feito nesse subvetor:



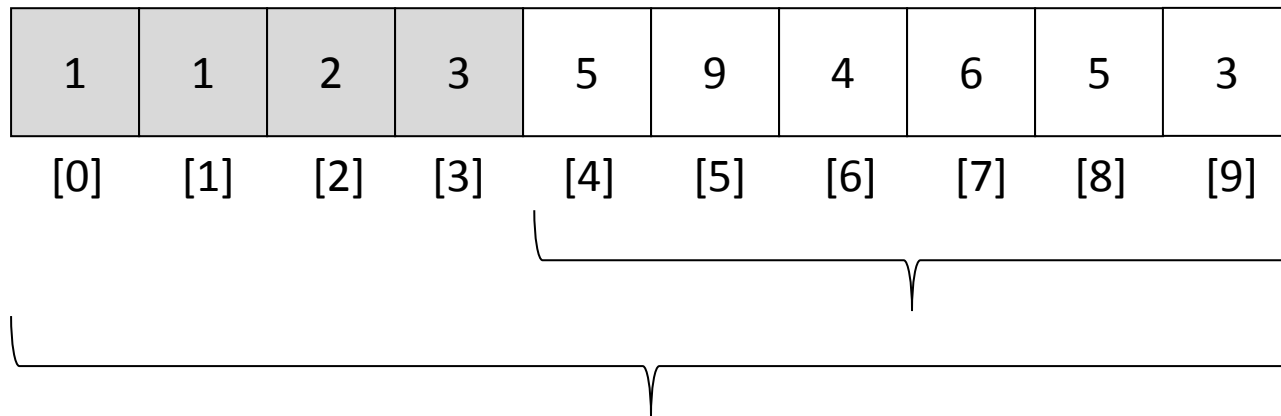
Quick Sort

(13) A recursão retorna. Não tem nada mais para ser feito nesse subvetor:



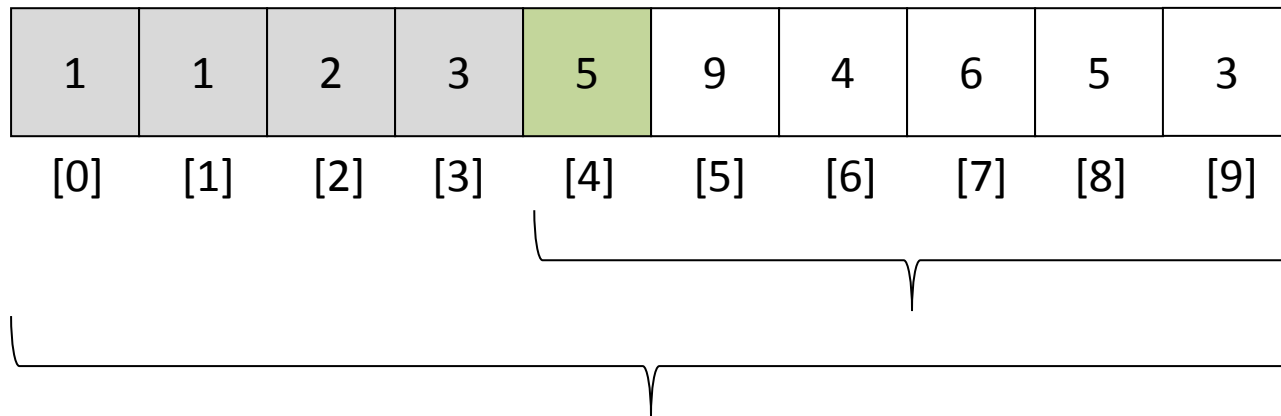
Quick Sort

(14) Chame recursivamente o quick sort para o subvetor da direita:



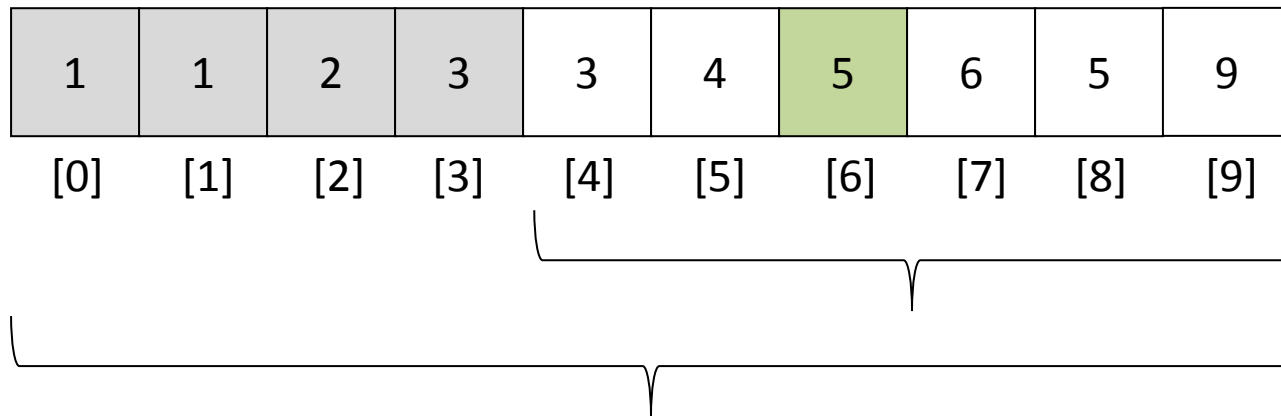
Quick Sort

(15) Selezione o pivô:



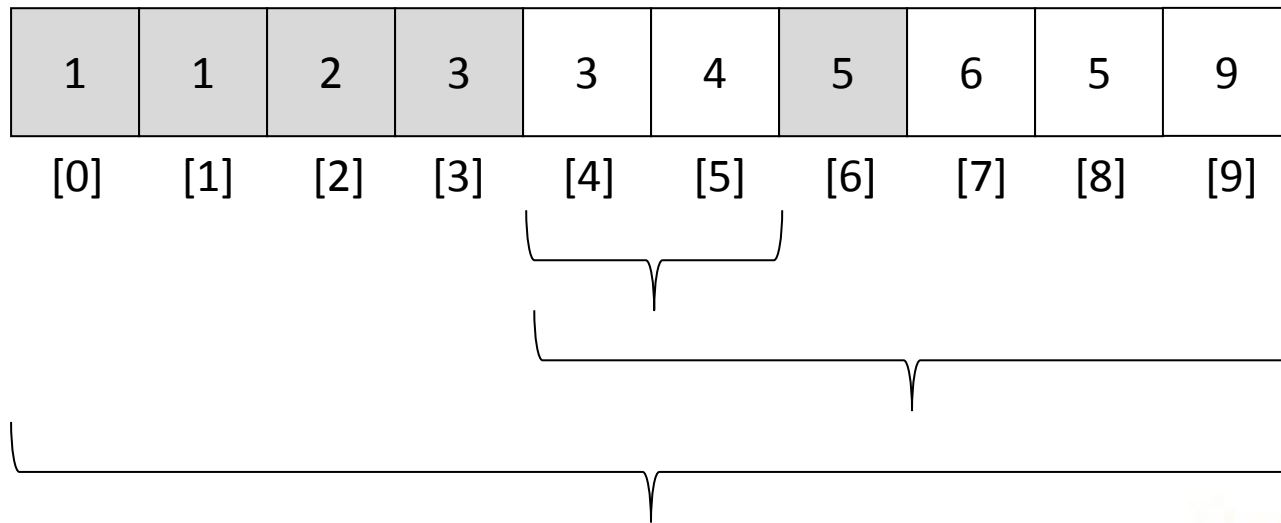
Quick Sort

(16) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



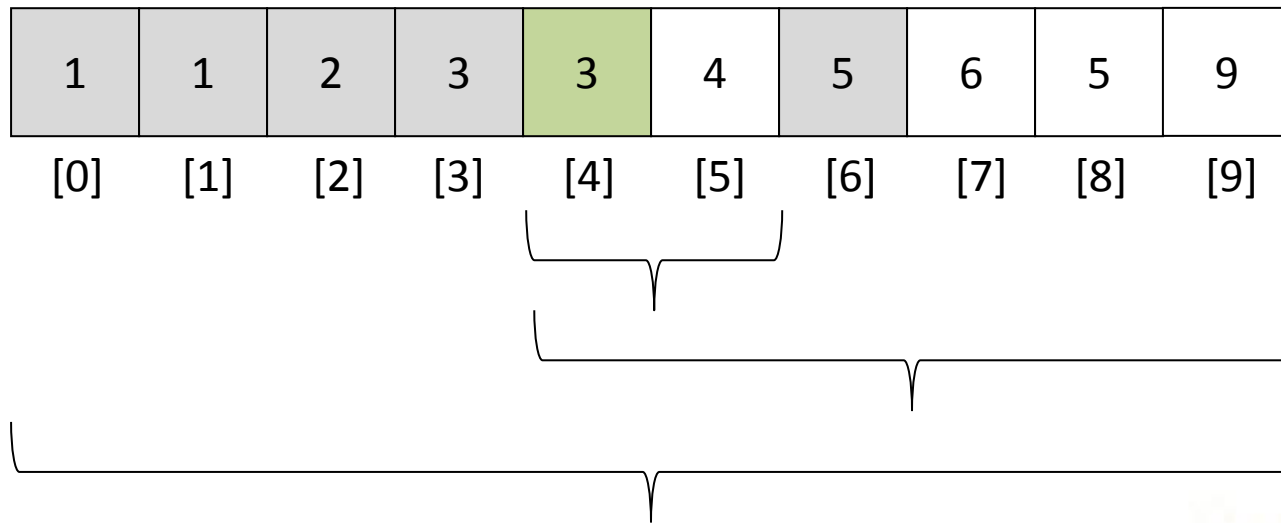
Quick Sort

(17) Chame recursivamente o quick sort para o subvetor da esquerda:



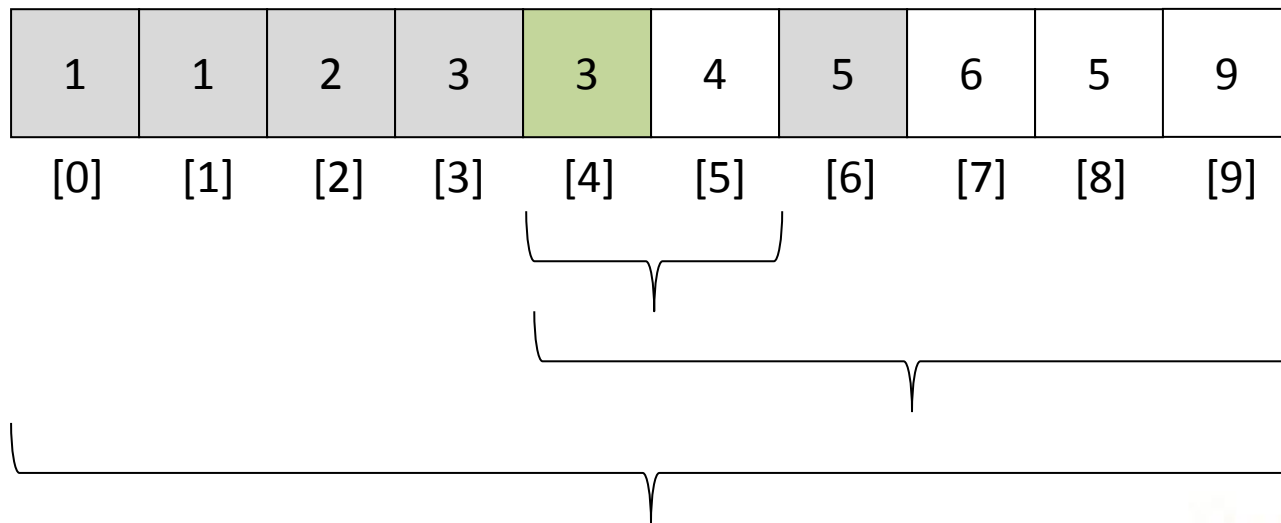
Quick Sort

(18) Selezione o pivô:



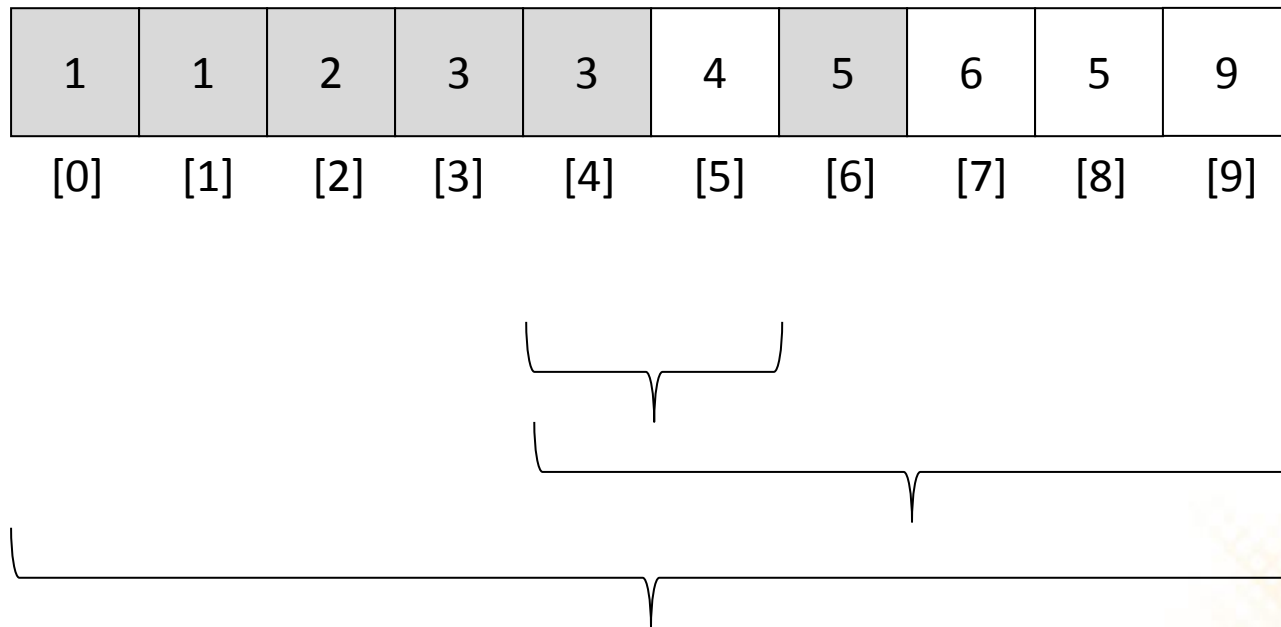
Quick Sort

(19) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



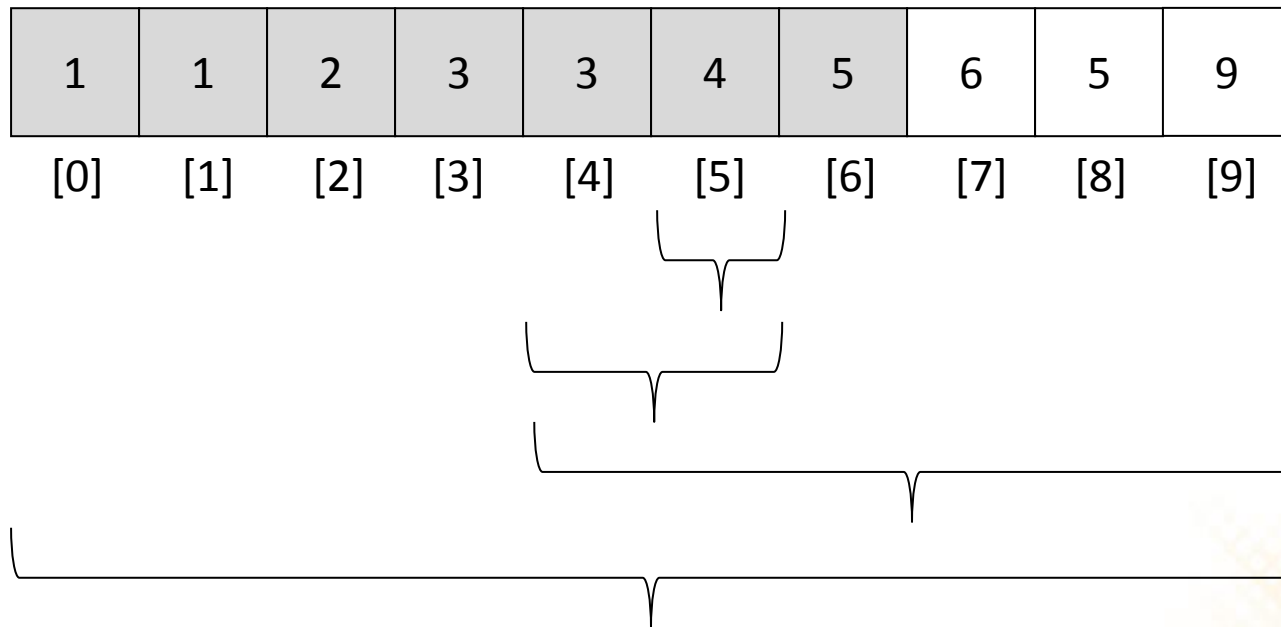
Quick Sort

(20) Chame recursivamente o quick sort para o subvetor da esquerda. O tamanho do vetor da esquerda é zero. A recursão retorna.



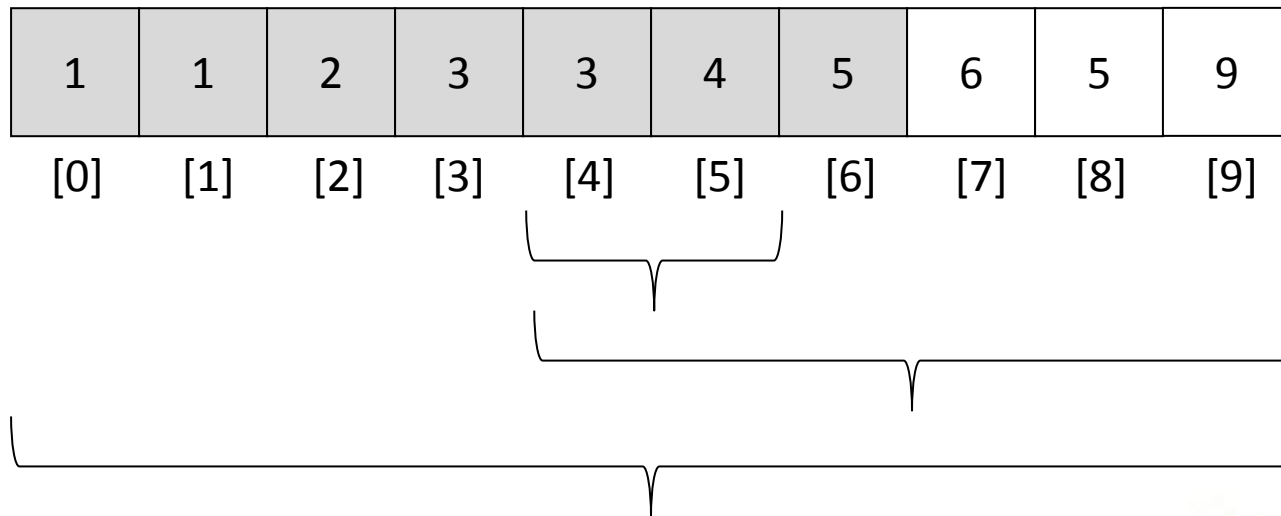
Quick Sort

(21) Chame recursivamente o quick sort para o subvetor da direita. Só existe um elemento, então ele já está ordenado e a recursão retorna.



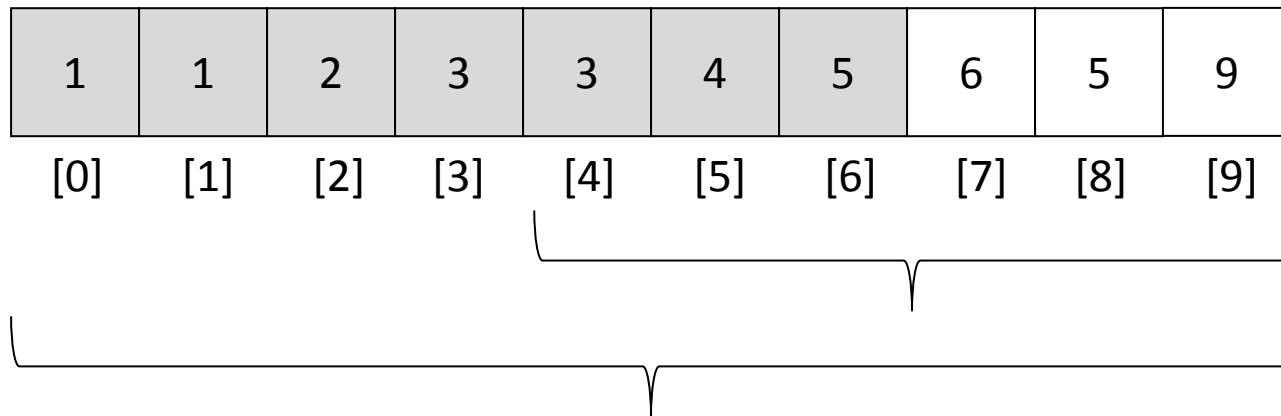
Quick Sort

(22) A recursão retorna. Não tem nada mais para ser feito nesse subvetor.



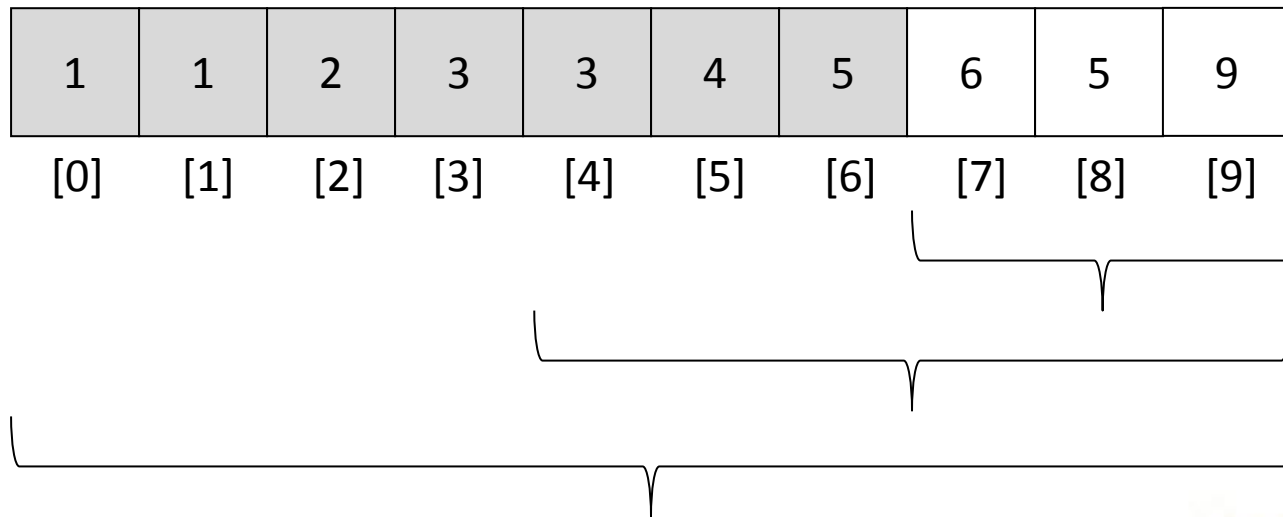
Quick Sort

(23) A recursão retorna. Não tem nada mais para ser feito nesse subvetor.



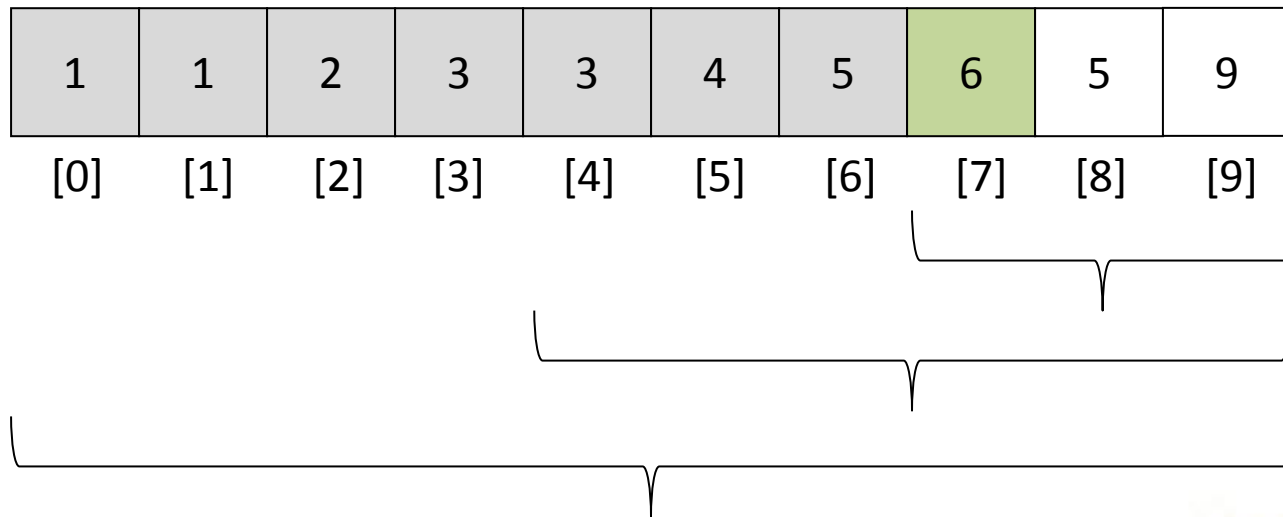
Quick Sort

(24) Chame recursivamente o quick sort para o subvetor da direita:



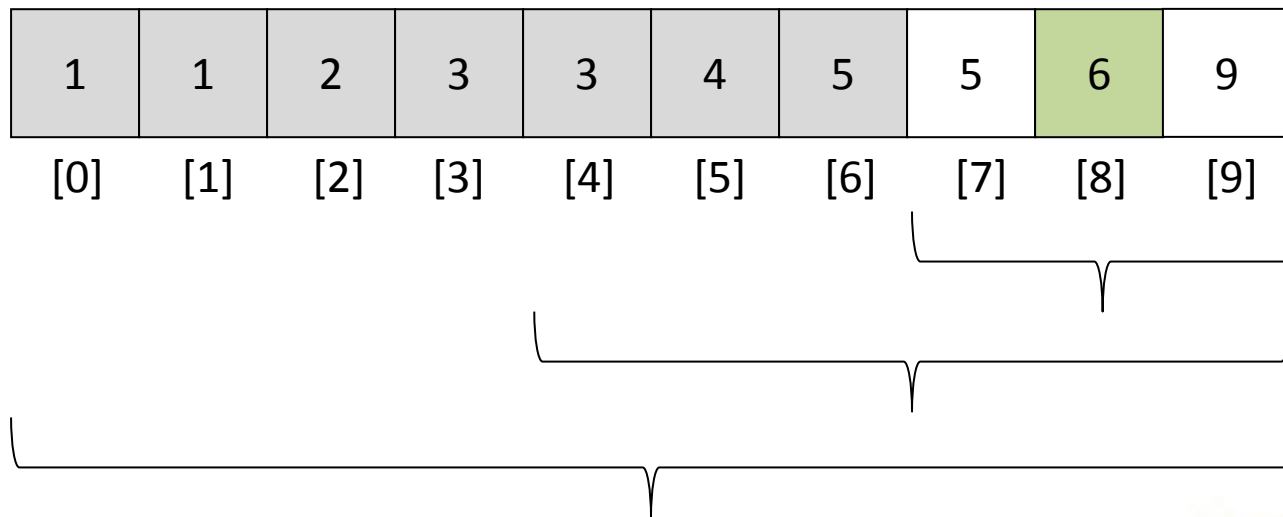
Quick Sort

(25) Selezione o pivô:



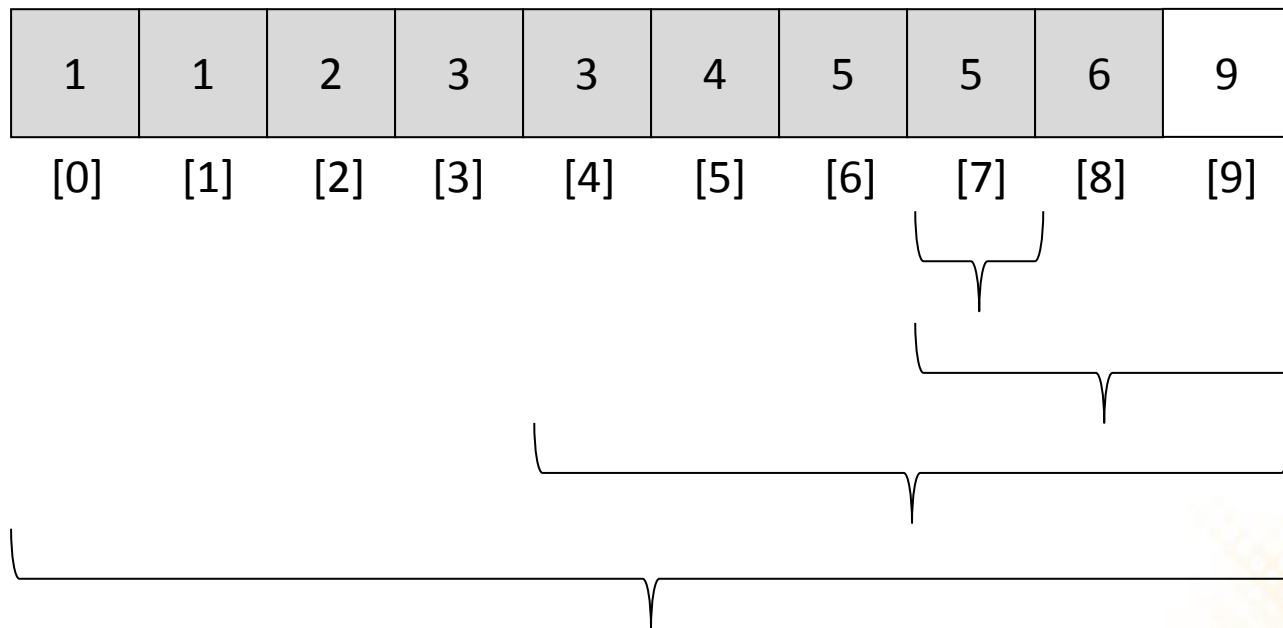
Quick Sort

(26) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



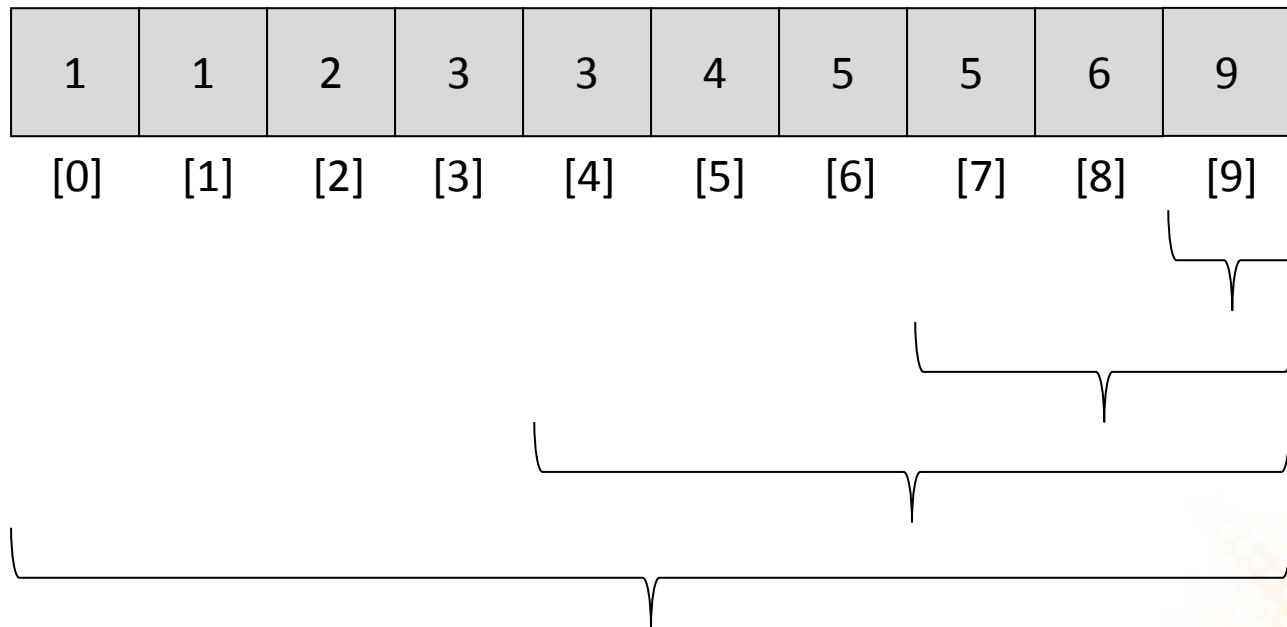
Quick Sort

(27) Chame recursivamente o quick sort para o subvetor da esquerda. Só existe um elemento, então ele já está ordenado e a recursão retorna:



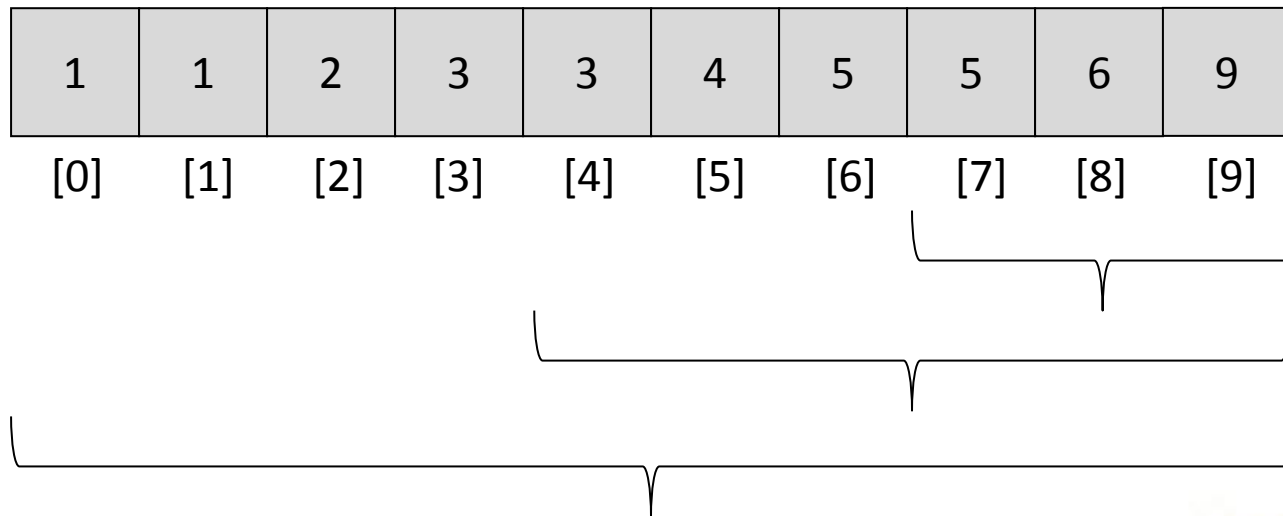
Quick Sort

(28) Chame recursivamente o quick sort para o subvetor da direita. Só existe um elemento, então ele já está ordenado e a recursão retorna:



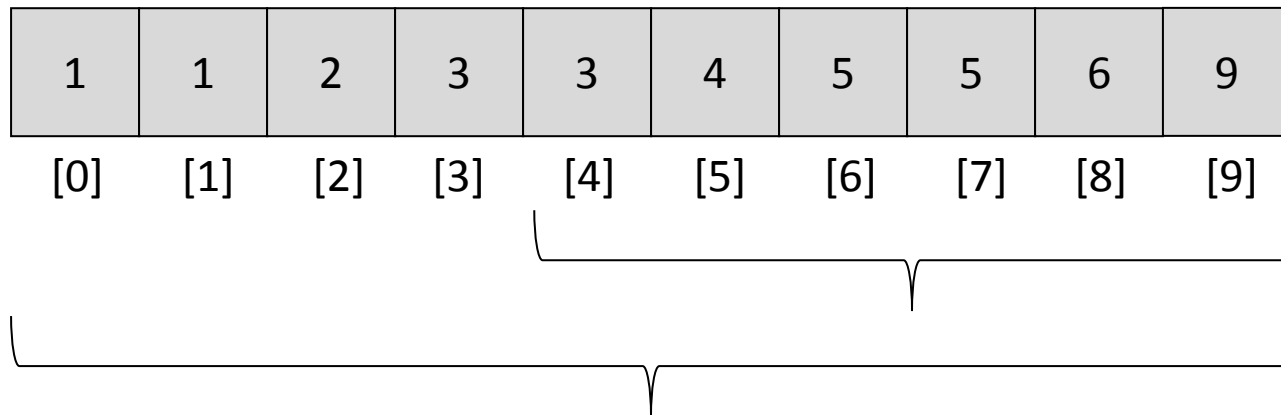
Quick Sort

(29) A recursão retorna. Não tem nada mais para ser feito nesse subvetor.



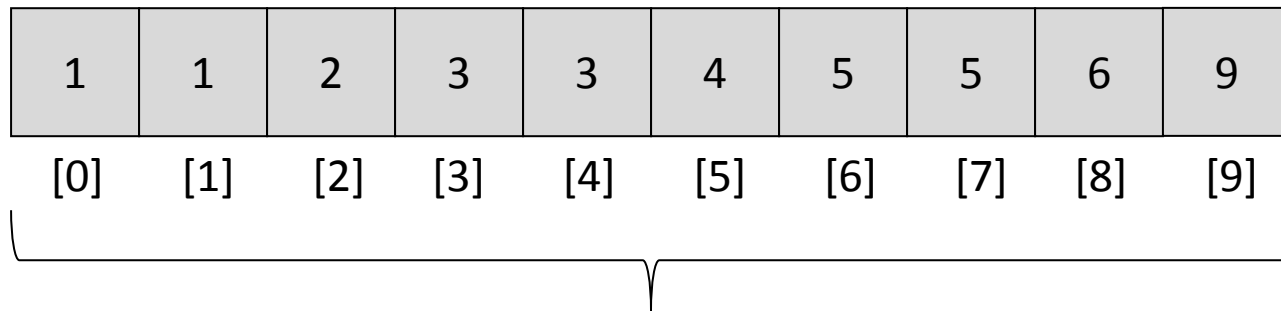
Quick Sort

(30) A recursão retorna. Não tem nada mais para ser feito nesse subvetor.



Quick Sort

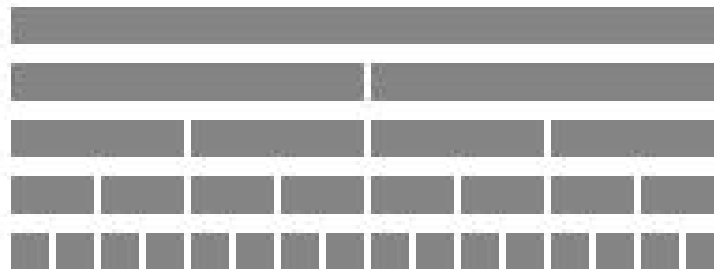
(31) A recursão retorna para a primeira chamada do quick sort com o vetor completamente ordenado.



Quick Sort - Complexidade

- **Melhor caso:**

- Pivô representa o valor mediano do conjunto dos elementos do vetor;
- Após mover o pivô para sua posição, restarão dois sub-vetores para serem ordenados, ambos com o número de elementos reduzido à metade, em relação ao vetor original;

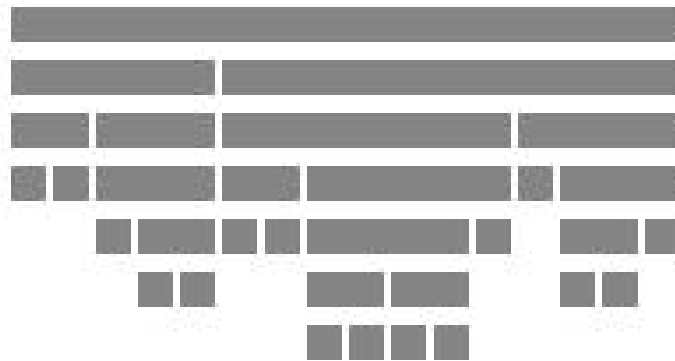


- Complexidade: $T(n) = 2T(n/2) + n = n \log n - n + 1 = \mathbf{O(n \log n)}$

Quick Sort - Complexidade

- **Caso médio:**

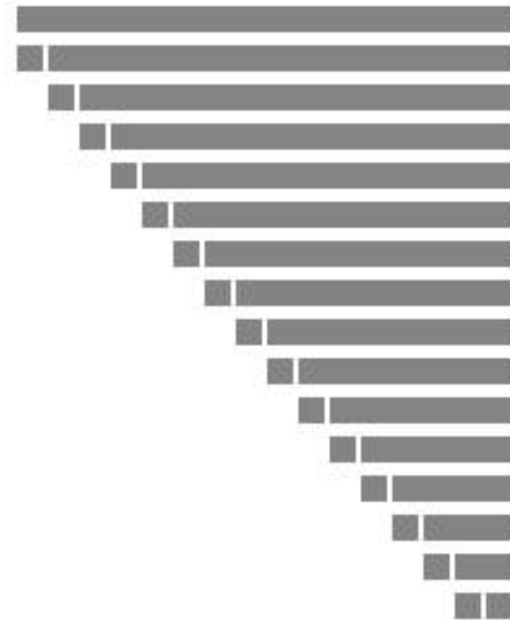
- De acordo com Sedgwick e Flajolet (1996, p. 17), o número de comparações é aproximadamente: $T(n) \approx 1.386 n \log n - 0.846$



- Complexidade: $T(n) = O(n \log n)$

Quick Sort - Complexidade

- **Pior caso:**
 - Pivô é o maior elemento e algoritmo recai em ordenação bolha;
 - Complexidade: $T(n) = O(n^2)$



Quick Sort

- Melhorias:
 - Eliminação da recursão;
 - Melhor seleção do pivô: mediana de 3 partições evita pior caso em conjuntos de dados ordenados;
 - Utilizar Insertion Sort conjuntos de dados pequenos;
 - Combinadas, essas melhorias fornecem um ganho de performance de aproximadamente 20-25%.
- O Quick Sort é considerado o melhor método para ordenação interna de grandes conjuntos de dados ($n \geq 10.000$).

Quick Sort

- Complexidade $O(n \log n)$
- Vantagens:
 - Complexidade $O(n \log n)$
- Desvantagens:
 - Tradicionalmente baseia-se em chamadas recursivas, mas é possível implementá-lo sem utilizar recursão;
 - Não é estável;

Exercícios

Lista de Exercícios 03 – Algoritmos de Ordenação

<http://www.inf.puc-rio.br/~elima/paa/>



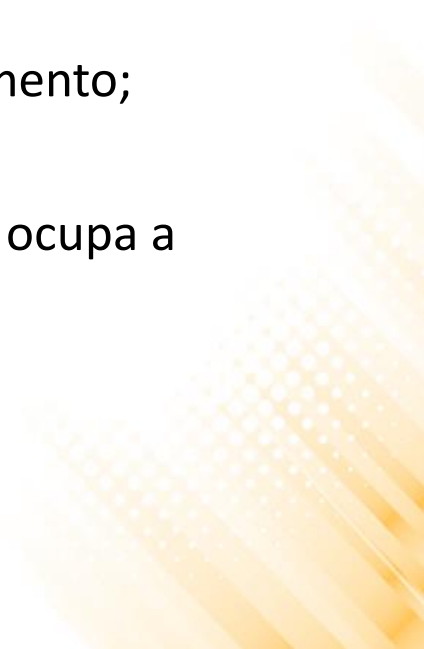
Métodos de Ordenação de Complexidade Linear

- O **limite assintótico mínimo** para algoritmos de ordenação baseados em comparações é **$O(n \log n)$** , mas isso não quer dizer que não seja possível existir um algoritmo de ordenação melhor!
- Existem algoritmos que **não são baseados em comparações**.
- Porém, eles exigem algum outro tipo de conhecimento sobre os dados a serem ordenados, portanto **não são tão genéricos** como os algoritmos clássicos de ordenação por comparação.

Métodos de Ordenação de Complexidade Linear

- Algoritmos de ordenação de complexidade $O(n)$:
 - **Counting Sort:** Os elementos a serem ordenados são números inteiros "pequenos", isto é, inteiros x onde $x \in O(n)$;
 - **Radix Sort:** Os elementos a serem ordenados são números inteiros de comprimento máximo constante, isto é, independente de n ;
 - **Bucket Sort:** Os elementos são números uniformemente distribuídos em um determinado intervalo (Exemplo: $[0..1)$).

Counting Sort

- Suponha que um vetor A a ser ordenado contenha n números inteiros, todos menores ou iguais a k ($k \in O(n)$).
 - Ideia básica:
 - Determinar, para cada elemento x da entrada A , o número de elementos maiores que x ;
 - Com esta informação, determinar a posição de cada elemento;
 - **Exemplo:** se 17 elementos forem menores que x então x ocupa a posição 18 na saída.
- 

Counting Sort

- O algoritmo Counting Sort ordena estes n números em tempo $O(n + k)$ (equivalente a $O(n)$).
- O algoritmo usa dois vetores auxiliares:
 - C , de tamanho k , que guarda em $C[i]$ o número de ocorrências de elementos i em A ;
 - B , de tamanho n , onde se constrói o vetor ordenado;

Counting Sort

```
function counting_sort(int A[], n, k)
begin
  for i ← 0 to k do
    C[i] ← 0
  for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1
  for i ← 1 to k do
    C[i] ← C[i] + C[i - 1]
  for j ← n down to 1 do
  begin
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
  end
end
end
```

$O(n + k)$

Counting Sort

A

4	0	2	2	4	3
---	---	---	---	---	---

```
for i ← 0 to k do  
  C[i] ← 0
```

C

0	0	0	0	0
---	---	---	---	---

```
for j ← 1 to n do  
  C[A[j]] ← C[A[j]]+1
```

C

1	0	2	1	2
---	---	---	---	---

```
for i ← 1 to k do  
  C[i] ← C[i] + C[i-1]
```

C

1	1	3	4	6
---	---	---	---	---

Counting Sort

A

4	0	2	2	4	3
---	---	---	---	---	---

```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```

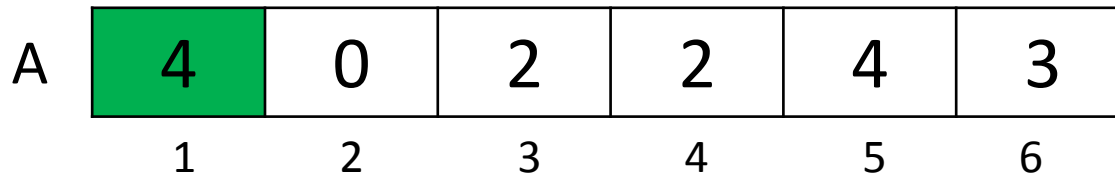
C

1	1	3	4	6
---	---	---	---	---

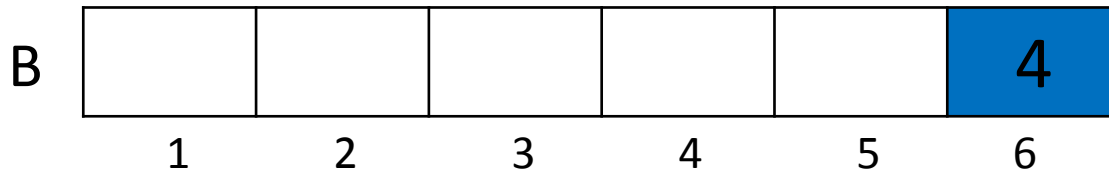
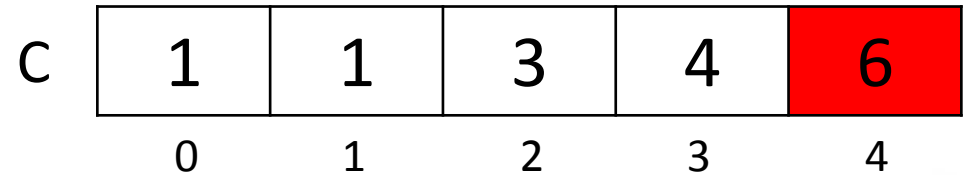
B

0	0	0	0	0
---	---	---	---	---

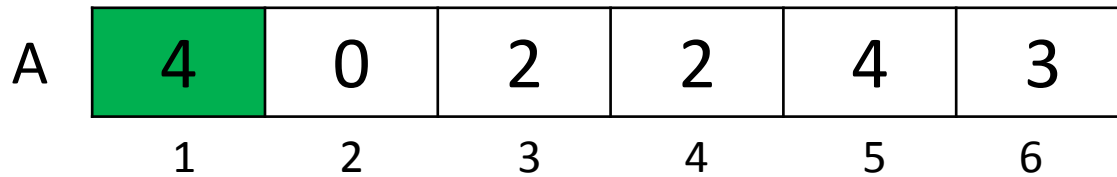
Counting Sort



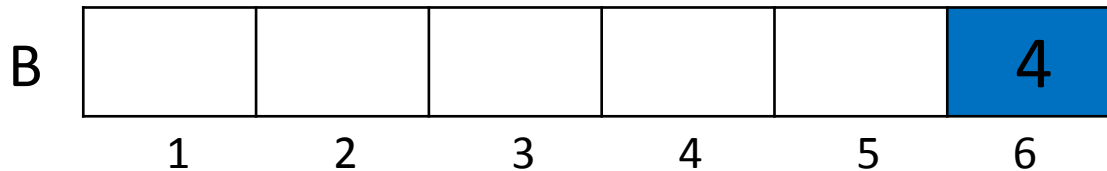
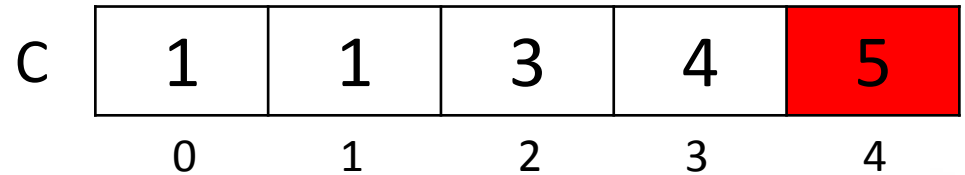
```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



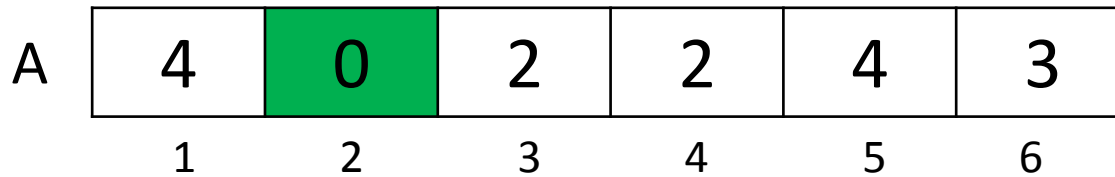
Counting Sort



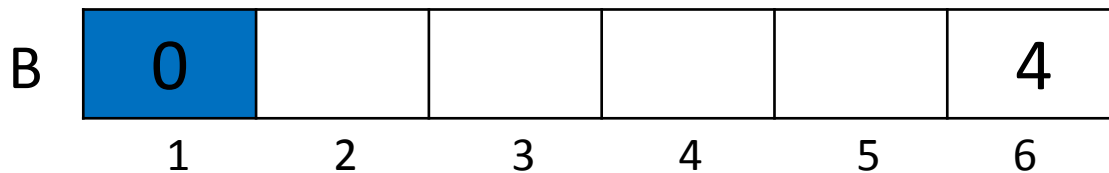
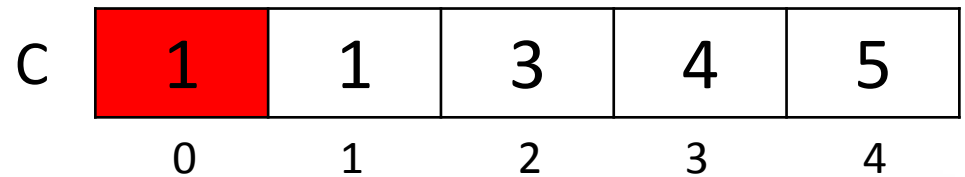
```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



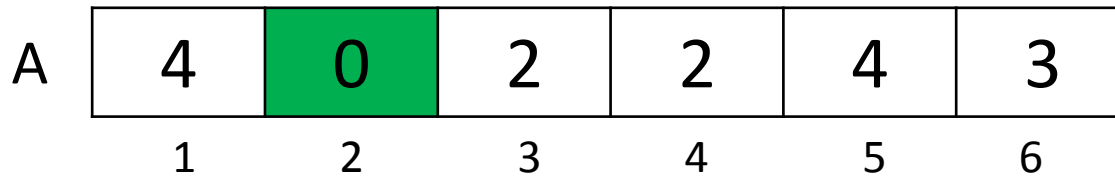
Counting Sort



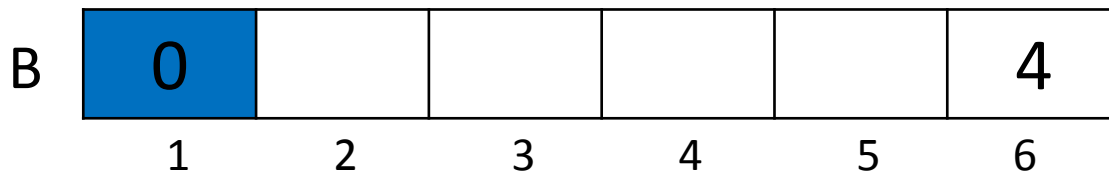
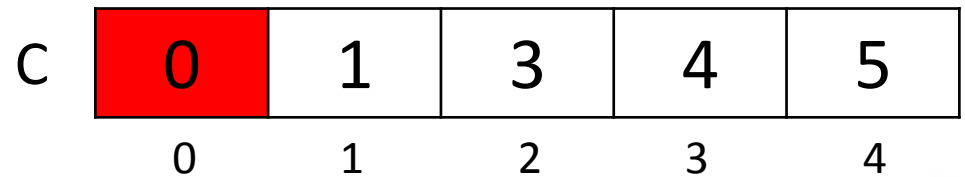
```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort



```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort

A

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

```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```

C

0	1	3	4	5
0	1	2	3	4

B

0		2			4
1	2	3	4	5	6

Counting Sort

A

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

```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```

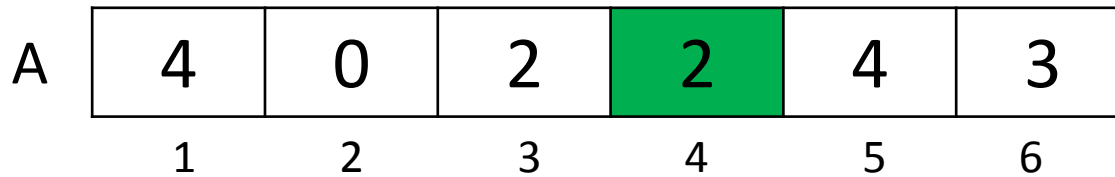
C

0	1	2	4	5
0	1	2	3	4

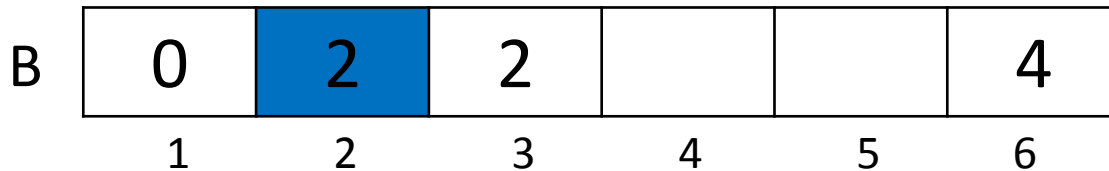
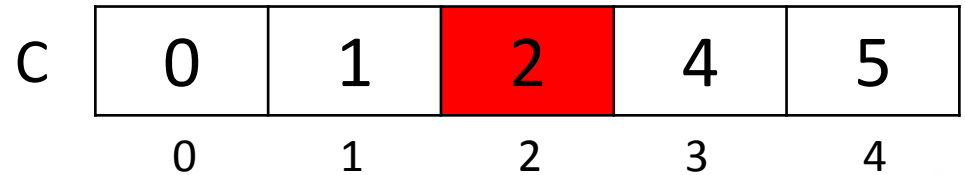
B

0		2			4
1	2	3	4	5	6

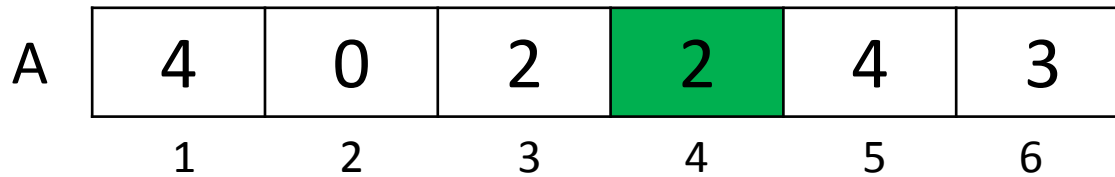
Counting Sort



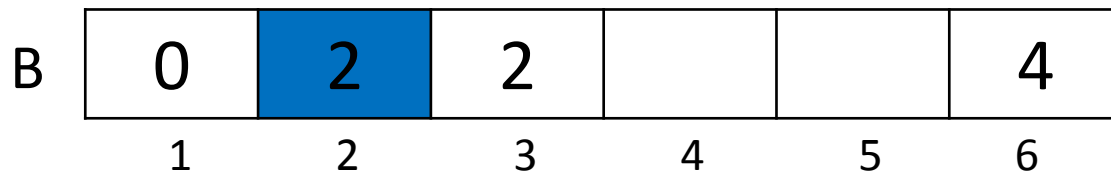
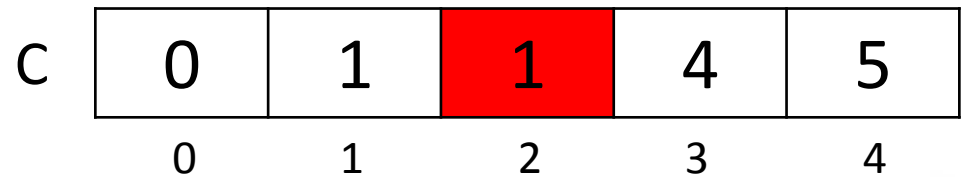
```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort



```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort

A

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

```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```

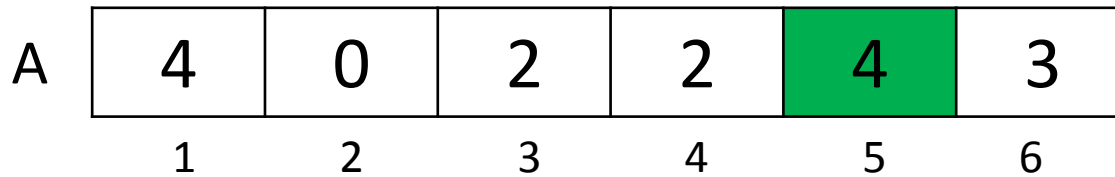
C

0	1	1	4	5
0	1	2	3	4

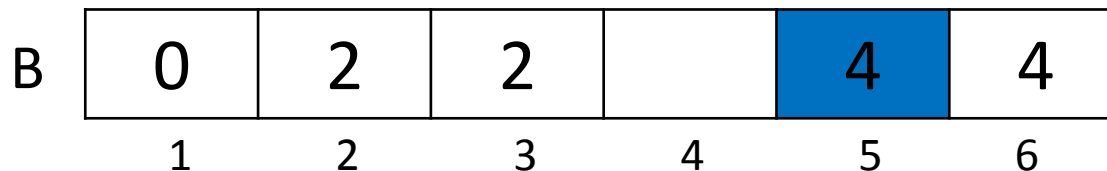
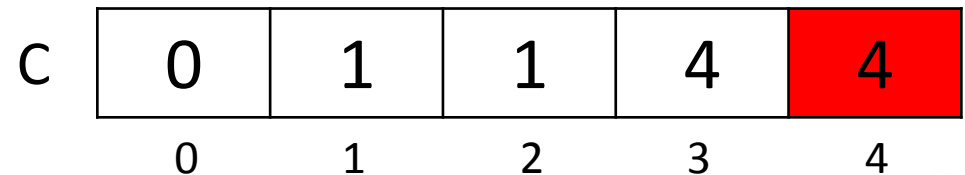
B

0	2	2		4	4
1	2	3	4	5	6

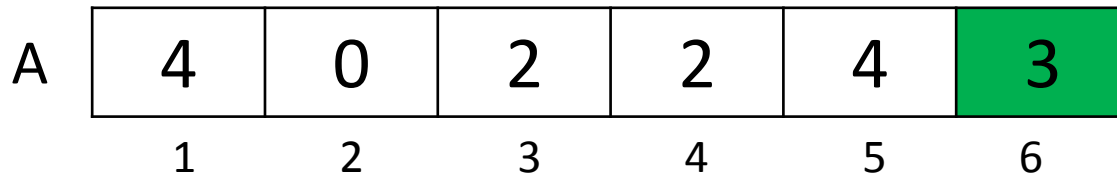
Counting Sort



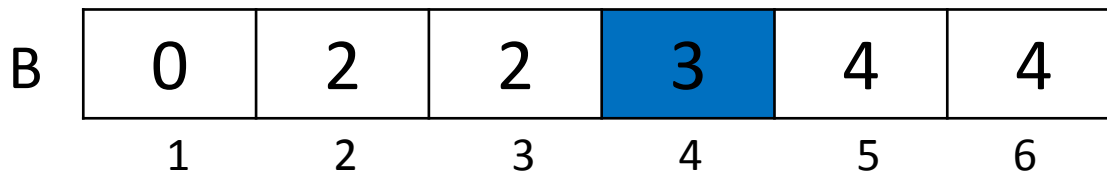
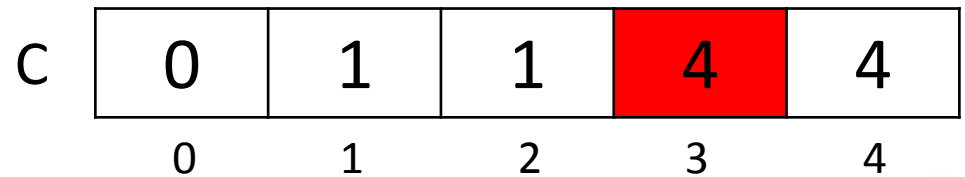
```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort



```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```



Counting Sort

A

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

```
for j ← n down to 1 do
begin
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] - 1
end
```

C

0	1	1	3	4
0	1	2	3	4

B

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

Counting Sort – Complexidade

- O algoritmo não faz comparações entre elementos de A .
- Sua complexidade deve ser medida com base nas outras operações (aritméticas, atribuições, etc.)
- Claramente, o número de tais operações é uma função em $O(n + k)$, já que temos dois loops simples com n iterações e dois com k iterações.
- Assim, quando $k \in O(n)$, este algoritmo tem complexidade $O(n)$.

Radix Sort

- Pressupõe que as chaves de entrada possuem limite no valor e no tamanho (quantidade de dígitos d);
- Ordena em função dos dígitos (um de cada vez) a partir do mais significativo ou a partir do menos significativo;
 - É essencial utilizar um segundo algoritmo estável para realizar a ordenação de cada dígito.

```
function radix_sort(int A[], n)
begin
  for i ← 1 to n do
    ordene os elementos de A pelo i-ésimo dígito usando
    um metodo estável
end
```

Radix Sort

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5



7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9



7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7



3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9

Radix Sort – Complexidade

- A complexidade do Radix Sort depende da complexidade do algoritmo estável usado para ordenar cada dígito dos elementos.
- Se essa complexidade for $O(n)$, obtemos uma complexidade total de $\Theta(dn)$ para o Radix Sort.
 - Como supomos d constante, a complexidade reduz-se para $O(n)$.
- Se o algoritmo estável for, por exemplo, o Counting Sort, obtemos a complexidade $O(n + k)$.
 - Supondo $k \in O(n)$, resulta numa complexidade **$O(n)$** .

Bucket Sort

- Assume que a entrada consiste em elementos distribuídos de forma **uniforme** sobre um determinado intervalo (exemplo: $[0..1)$);
- A ideia do Bucket Sort é dividir o intervalo em n subintervalos de mesmo tamanho (**balde**), e então distribuir os n números nos baldes;
- Uma vez que as entradas são uniformemente distribuídas, não se espera que muitos números **caiam em cada balde**;
- Para produzir a saída ordenada, basta ordenar os números em cada balde, e depois **examinar os baldes em ordem**, listando seus elementos;

Bucket Sort

```
function bucket_sort(int A[], n, k)
begin
    crie o vetor de baldes (listas) B de tamanho k

    for i ← 1 to n do
        insira A[i] no seu balde B[msbits(A[i])] na ordem

    for i ← 1 to k do
        concatene os baldes B[i]
end
```

Bucket Sort

B

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

Bucket Sort

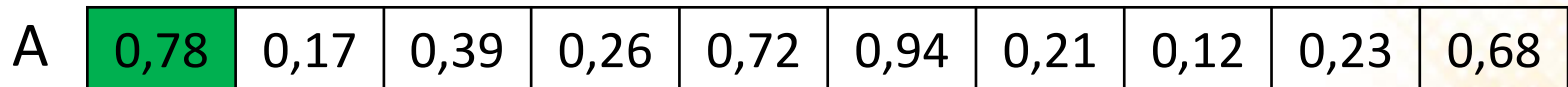
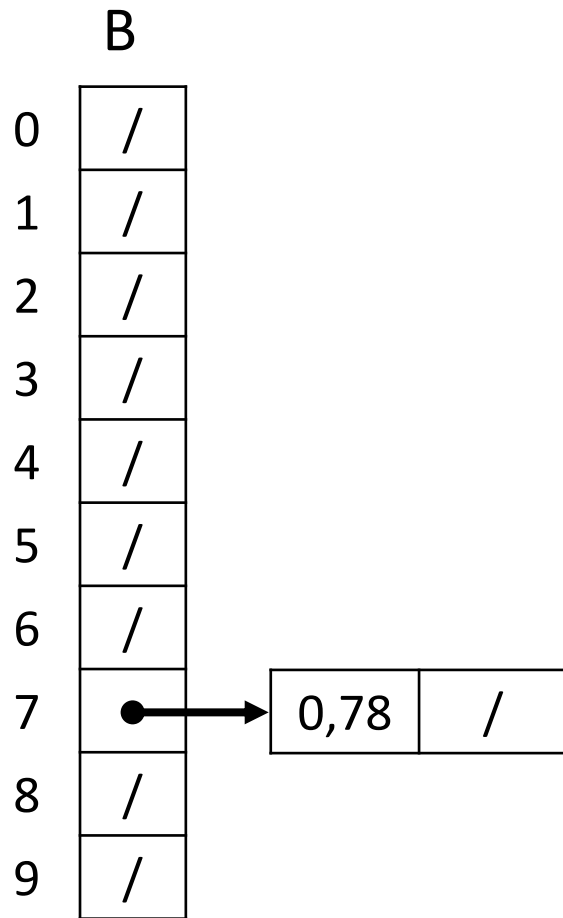
B

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

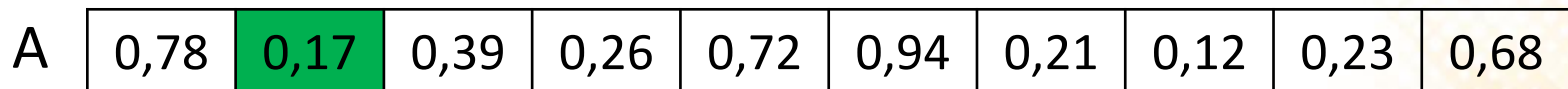
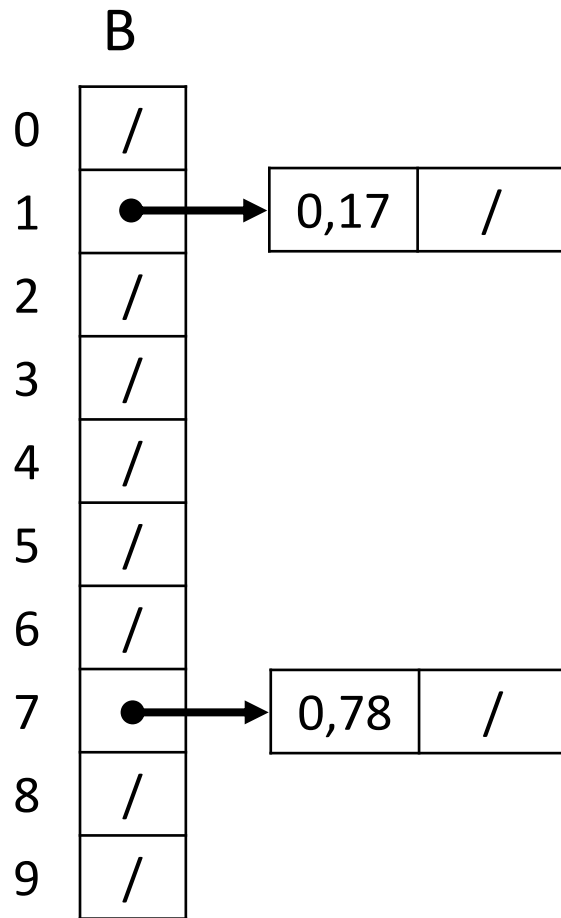
A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

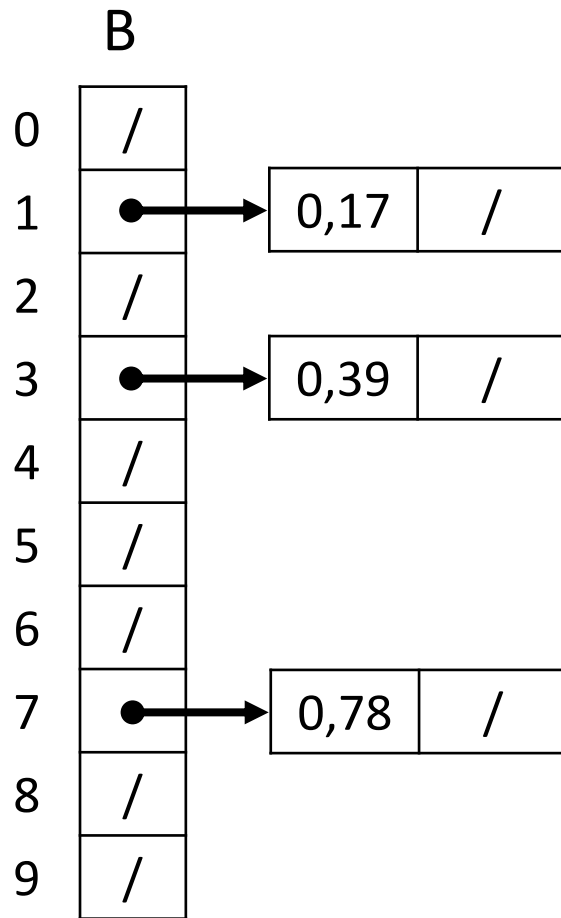
Bucket Sort



Bucket Sort



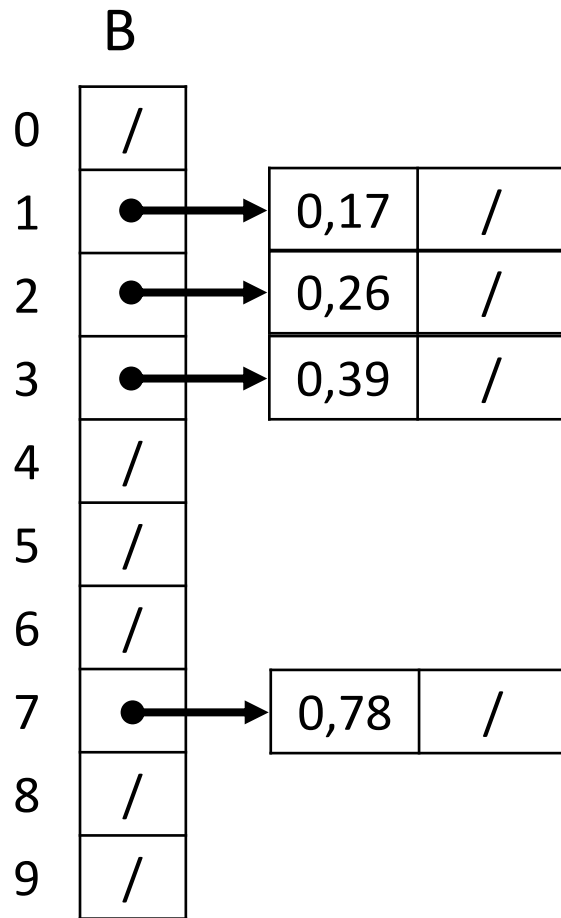
Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

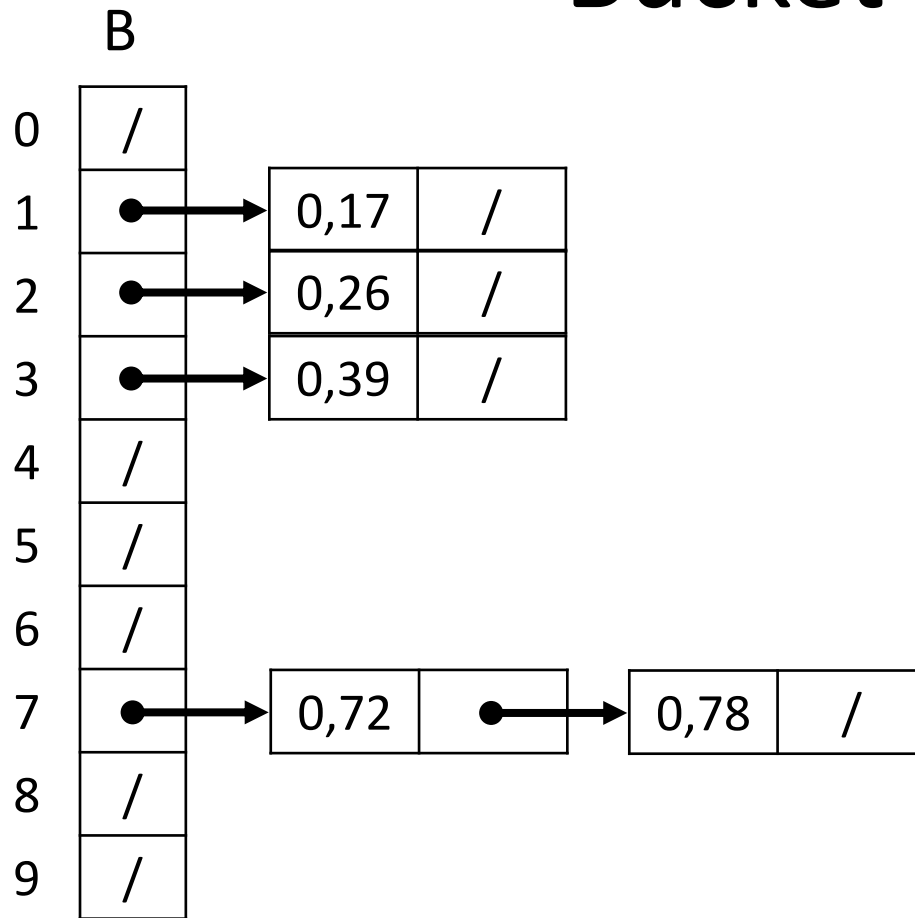
Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

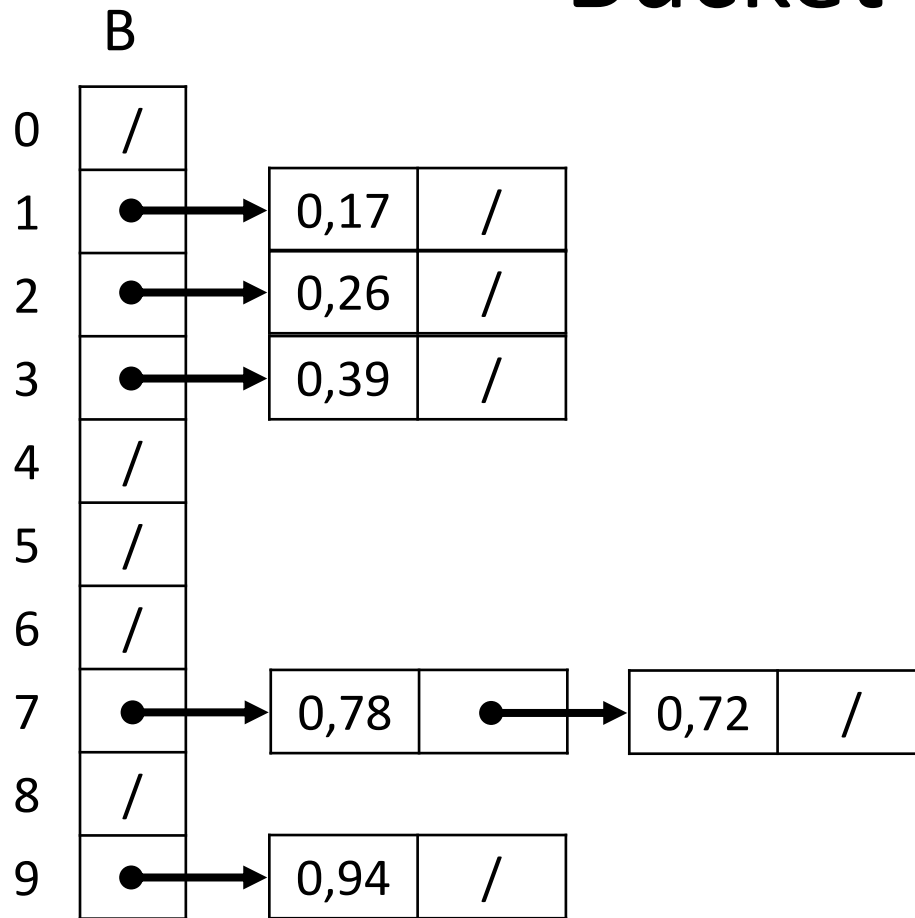
Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

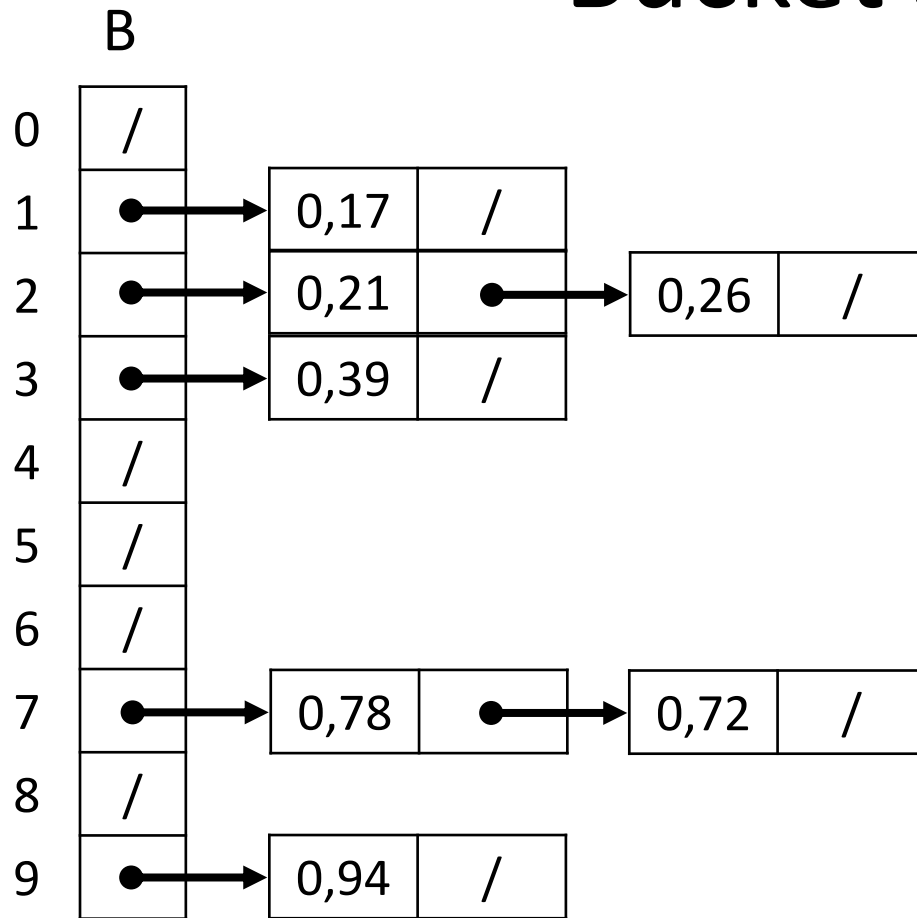
Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

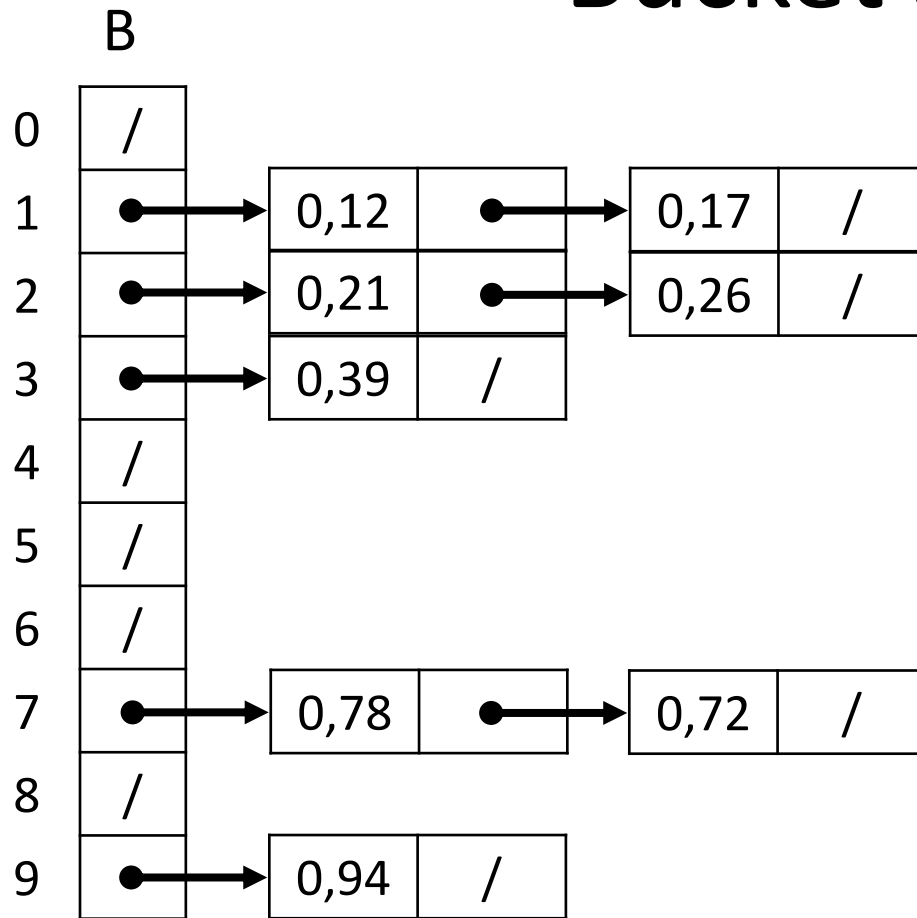
Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

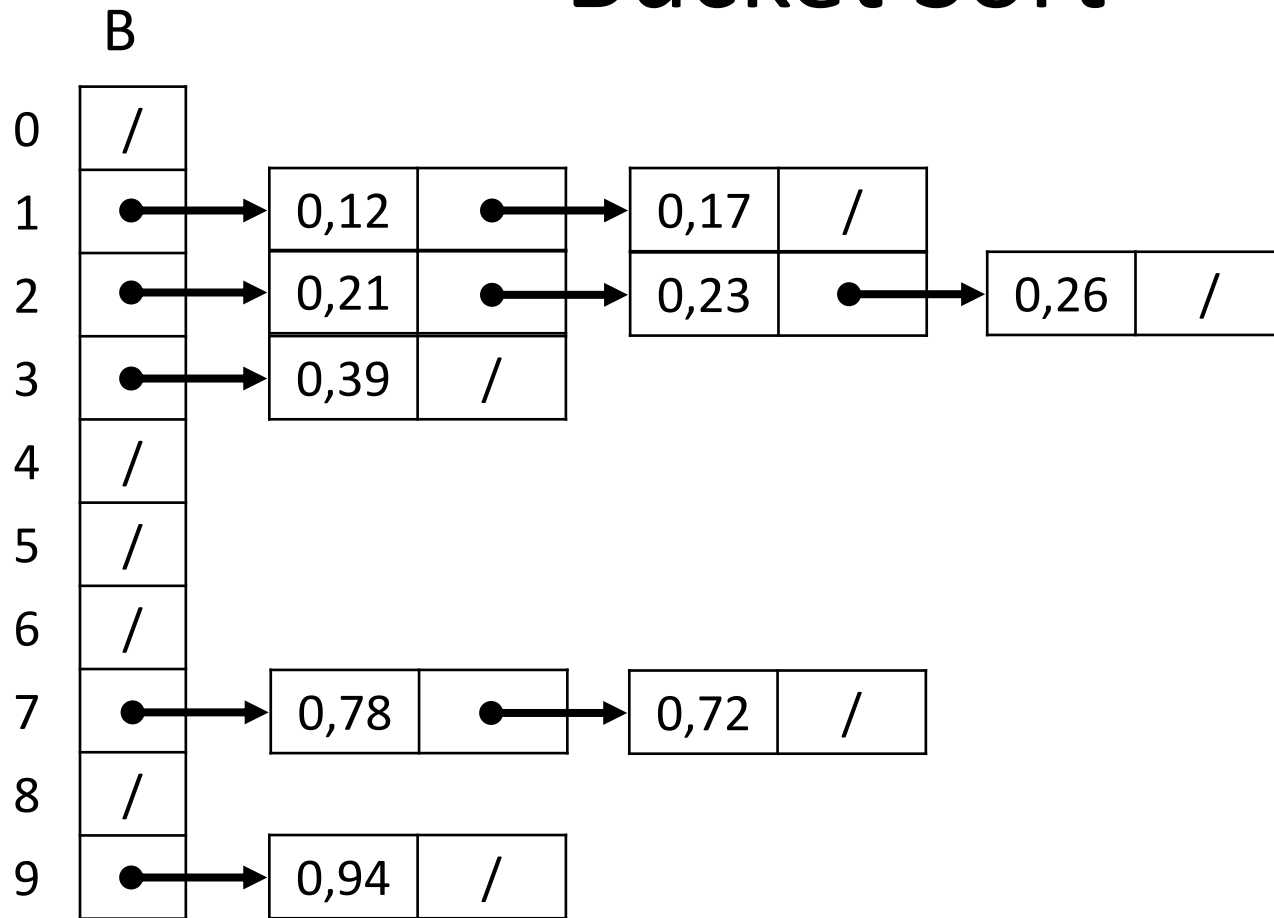
Bucket Sort



A

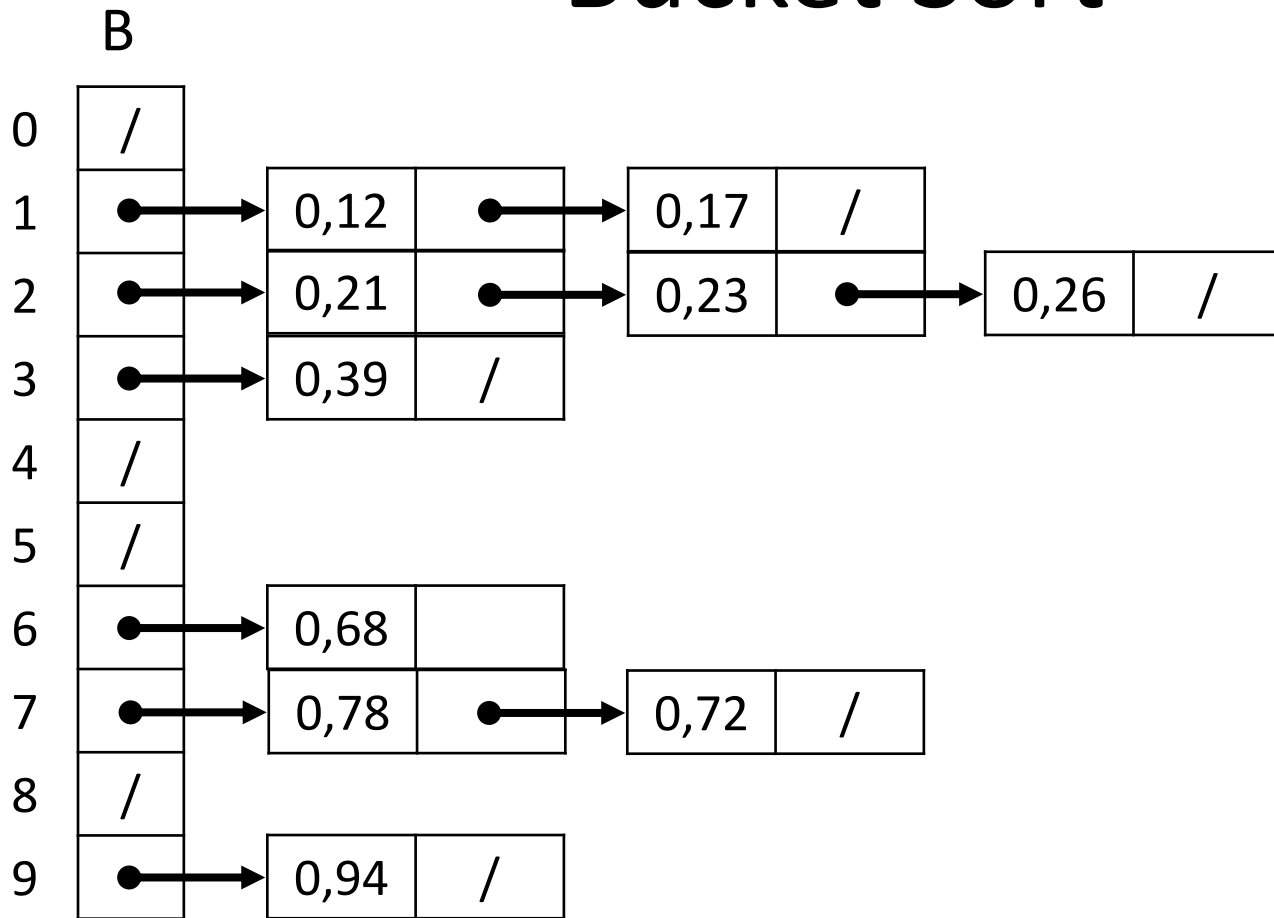
0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

Bucket Sort



A 0,78 0,17 0,39 0,26 0,72 0,94 0,21 0,12 0,23 0,68


Bucket Sort



A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68
------	------	------	------	------	------	------	------	------	------

Bucket Sort – Complexidade

- Se o vetor estiver uniformemente distribuído:
 - Caso médio: $O(n + k)$
 - Pior Caso: $O(n^2)$
 - Bom quando o número de chaves é pequeno e há em média poucos elementos por balde.
- 

Exercícios

Lista de Exercícios 04 – Algoritmos de Ordenação de Complexidade Linear

<http://www.inf.puc-rio.br/~elima/paa/>

