Pontifícia Universidade Católica do Rio de Janeiro

INF 1007 – Programação II

Aula 09 – Ordenação de Vetores

Edirlei Soares de Lima

<elima@inf.puc-rio.br>

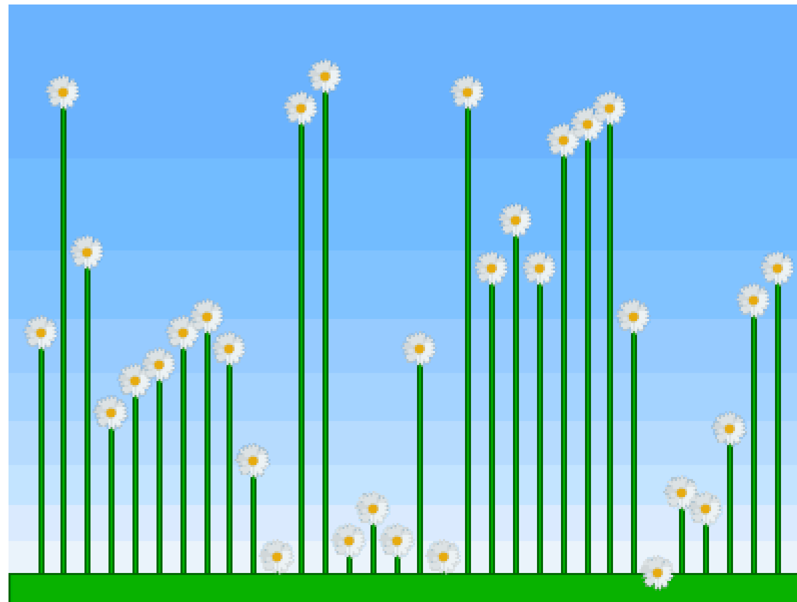
visionlab
visualization laboratory

Ordenação de Vetores

- **Problema:**
  - **Entrada:**
    - vetor com os elementos a serem ordenados;
  - **Saída:**
    - mesmo vetor com elementos na ordem especificada;
- **Algoritmos básicos de ordenação:**
  - Ordenação Bolha (Bubble Sort);
  - Ordenação rápida (Quick Sort);

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

12 / 6	6 / 12	22 / 14	14 / 22 / 8	22 / 8 / 17	17 / 22

6	12	14 / 8	8 / 14	17	22

Bubble Sort

6	12	14	8	17	22

6	12	8	14	17	22

6	12 / 8	8 / 12	14	17	22

6	8	12	14	17	22

Bubble Sort - Exemplo

25 48 37 12 57 86 33 92

25 48 37 12 57 86 33 92 — 25x48
25 **48 37** 12 57 86 33 92 — 48x37 troca
25 37 **48 12** 57 86 33 92 — 48x12 troca
25 37 12 **48 57** 86 33 92 — 48x57
25 37 12 48 **57 86** 33 92 — 57x86
25 37 12 48 57 **86 33** 92 — 86x33 troca
25 37 12 48 57 33 **86 92** — 86x92
25 37 12 48 57 33 86 **92** — **Final do Passo 1**

O maior elemento, 92, já está na sua posição final!

Bubble Sort - Exemplo

25 37 12 48 57 33 86 **92** — **Final do Passo 1**

25 37 12 48 57 33 86 **92** — 25x37

25 **37 12** 48 57 33 86 **92** — 37x12 troca

25 12 **37 48** 57 33 86 **92** — 37x48

25 12 37 **48 57** 33 86 **92** — 48x57

25 12 37 48 **57 33** 86 **92** — 57x33 troca

25 12 37 48 33 **57 86** **92** — 57x86

25 12 37 48 33 57 **86 92** — **Final do Passo 2**

O segundo maior elemento, 86, já está na sua posição final!

Bubble Sort - Exemplo

```
25 12 37 48 33 57 86 92        Final do Passo 2

25 12 37 48 33 57 86 92        25x12  troca
12 25 37 48 33 57 86 92        25x37
12 25 37 48 33 57 86 92        37x48
12 25 37 48 33 57 86 92        48x33  troca
12 25 37 33 48 57 86 92        48x57
12 25 37 33 48 57 86 92        Final do Passo 3
```

Bubble Sort - Exemplo

```
25 12 37 48 33 57 86 92        Final do Passo 3

12 25 37 33 48 57 86 92        12x25
12 25 37 33 48 57 86 92        25x37
12 25 37 33 48 57 86 92        37x33 troca
12 25 33 37 48 57 86 92        37x48
12 25 33 37 48 57 86 92        Final do Passo 4

12 25 33 37 48 57 86 92        12x25
12 25 33 37 48 57 86 92        25x33
12 25 33 37 48 57 86 92        33x37
12 25 33 37 48 57 86 92        Final do Passo 5
```

Bubble Sort - Exemplo

25 12 33 **37 48 57 86 92**	**Final do Passo 5**
12 25 33 37 48 57 86 92	12x25
12 **25 33** 37 48 57 86 92	25x33
12 25 **33 37 48 57 86 92**	**Final do Passo 6**
12 25 33 37 48 57 86 92	12x25
12 **25 33 37 48 57 86 92**	**Final do Passo 7**
12 25 33 37 48 57 86 92	**Fim da Ordenação**

Bubble Sort - Implementação Iterativa (I)

```
void bolha(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 bolha(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

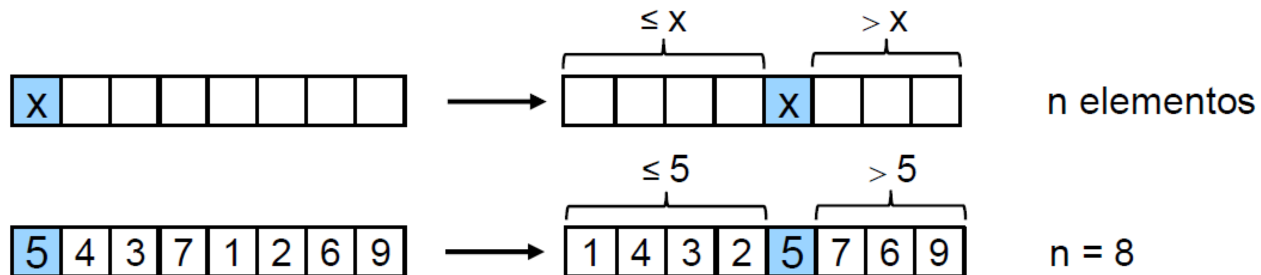
- Tempo total gasto pelo algoritmo:
  - T(n) = (n-1) + (n-2) + ... + 2 + 1
  - T(n) = n(n-1)/2 = $\frac{n^2 - n}{2}$
  - Algoritmo de ordem quadrática: $O(n^2)$

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], … v[i-1] são menores que x;

       todos os elementos v[i+1], …, v[n-1] são maiores que x;



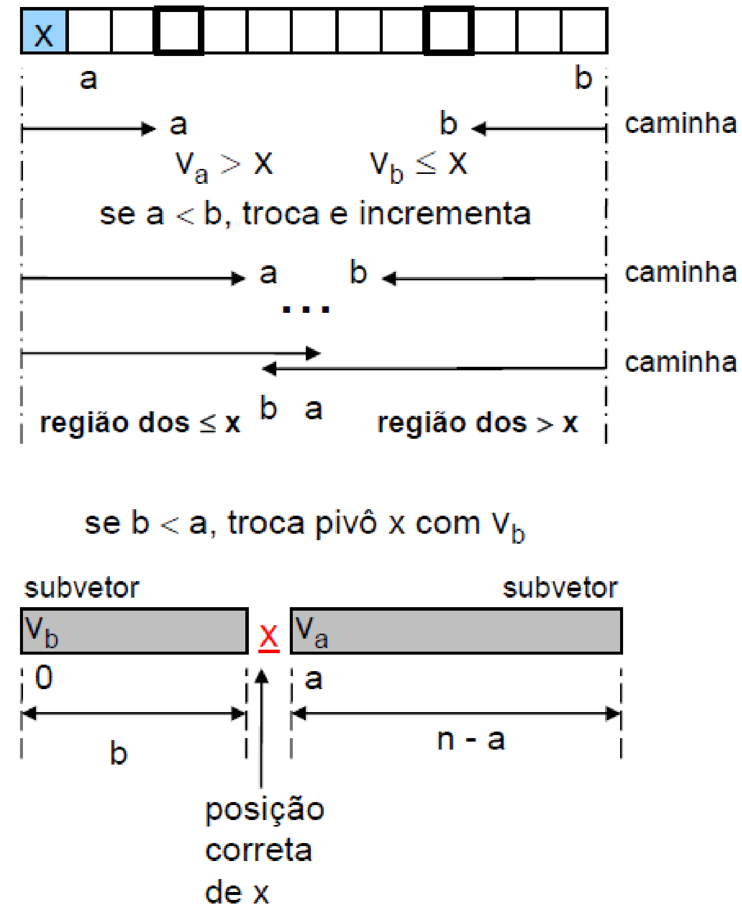
  3. Chame **recursivamente** o algoritmo para ordenar os subvetores v[0], … v[i-1] e v[i+1], …, v[n-1] (vetor da esquerda e vetor da direita)
     - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento

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
```

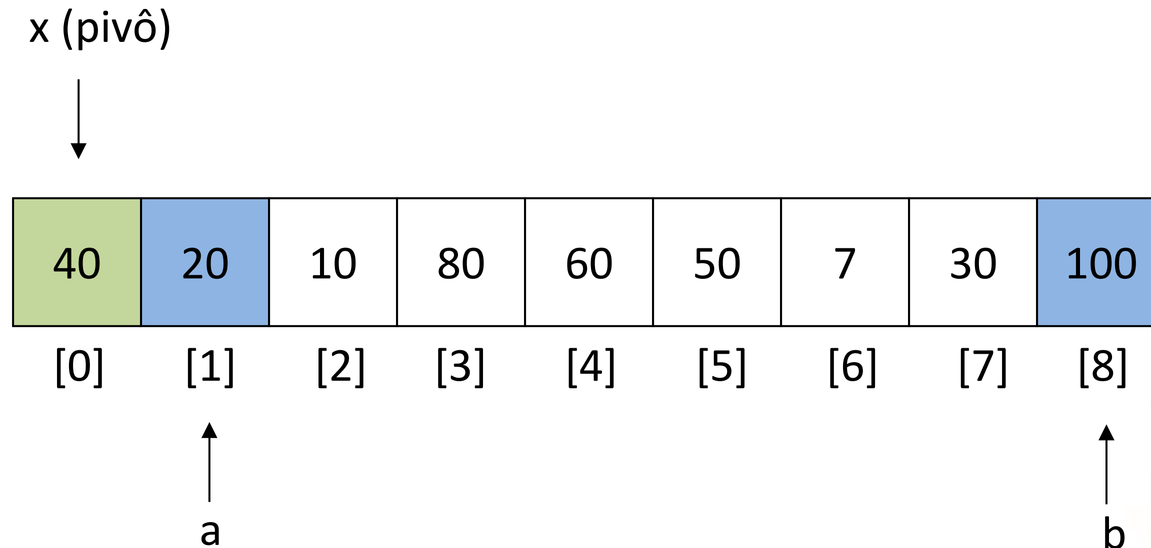
Quick Sort

- **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**] <= 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

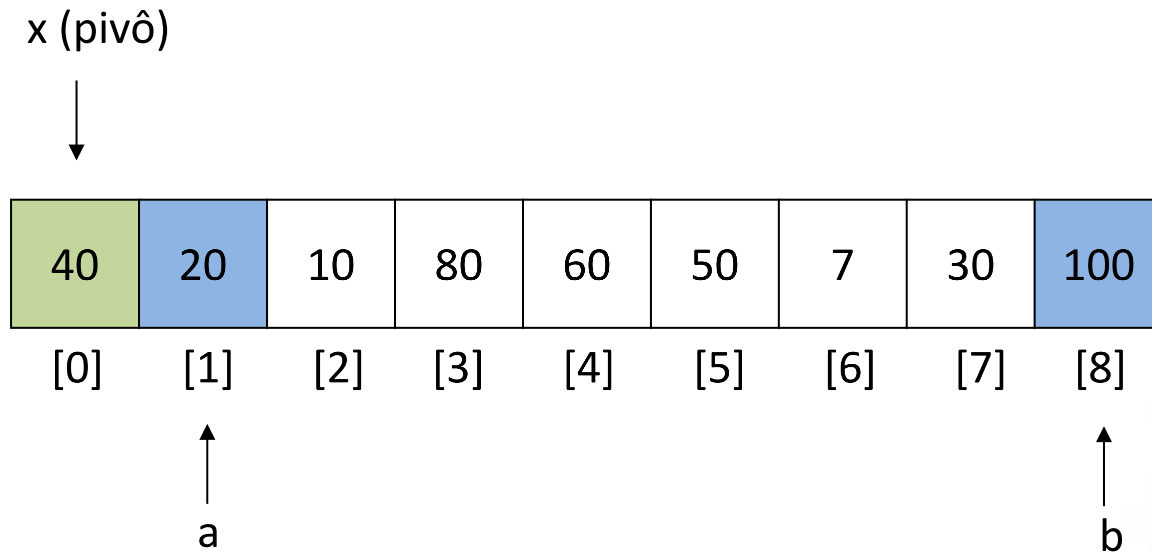


Quick Sort

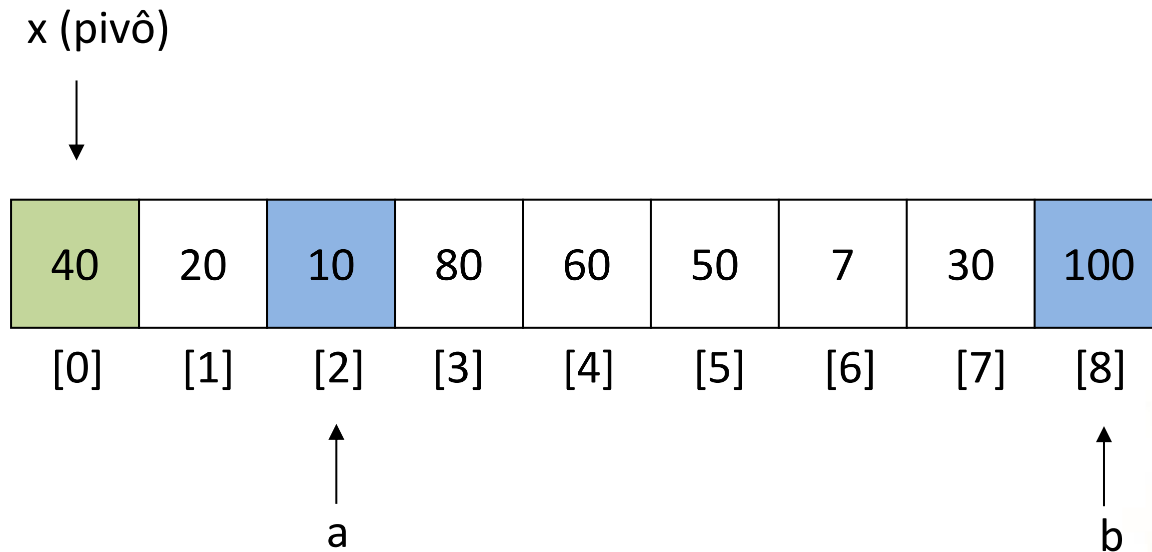
- **Processo de Partição:**
  – Exemplo:



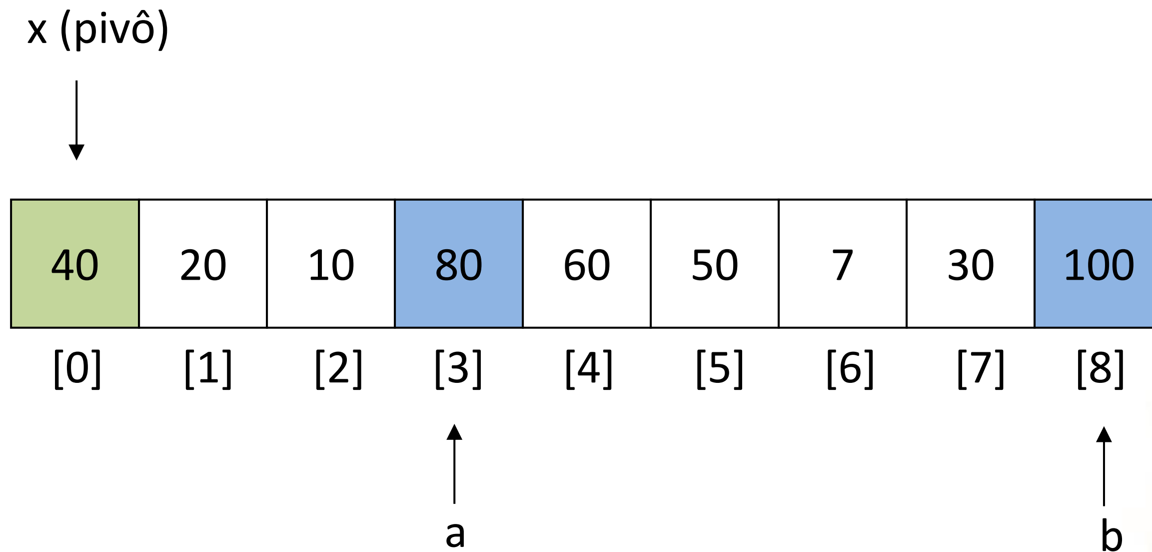
```
do{
  while (a < n && v[a] <= x)
    a++;
```



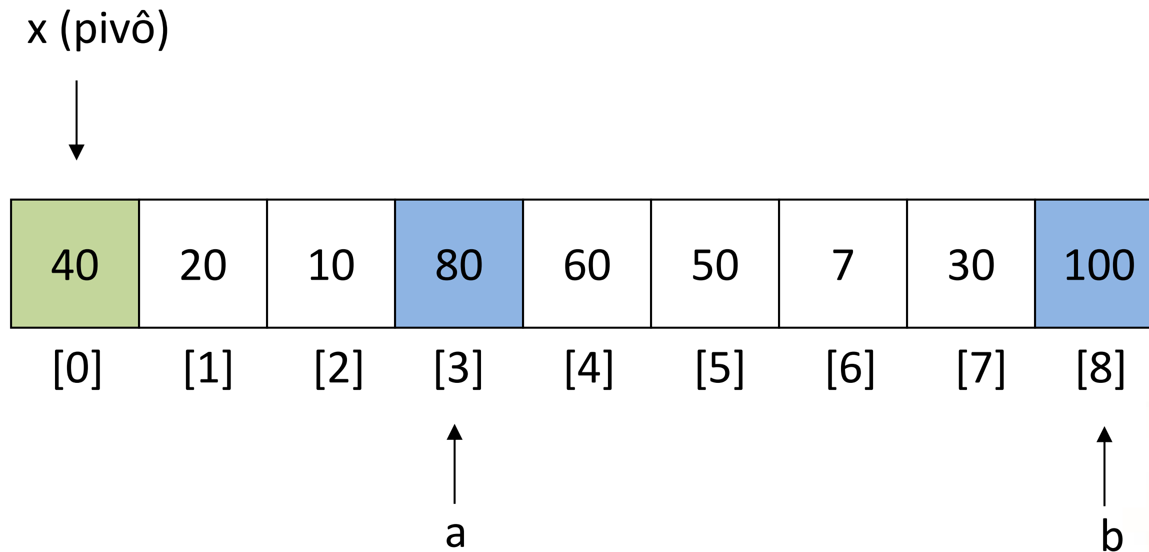
```
do{
  while (a < n && v[a] <= x)
    a++;
```



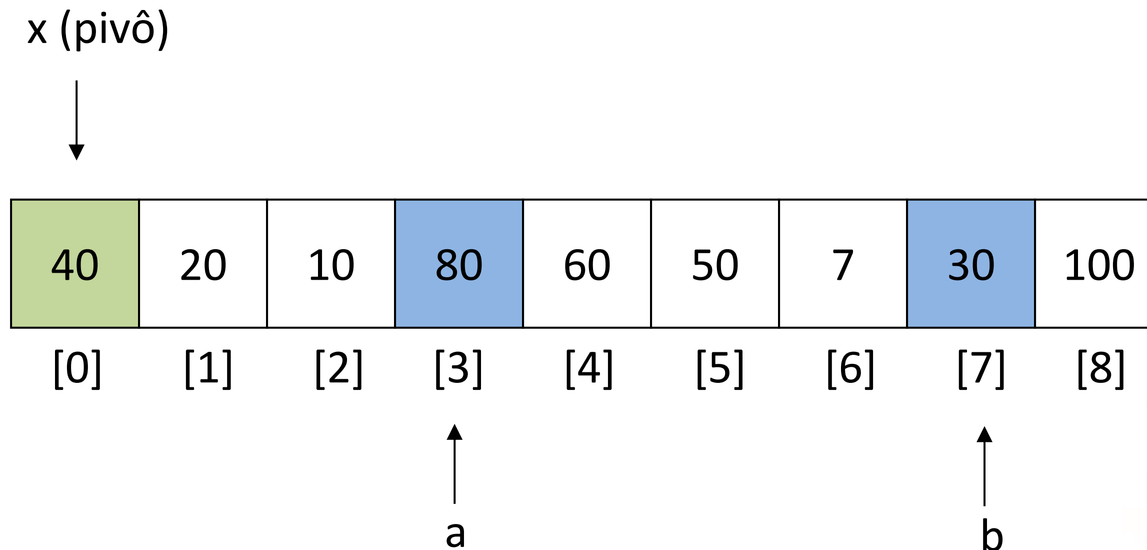
```
do{
  while (a < n && v[a] <= x)
    a++;
```



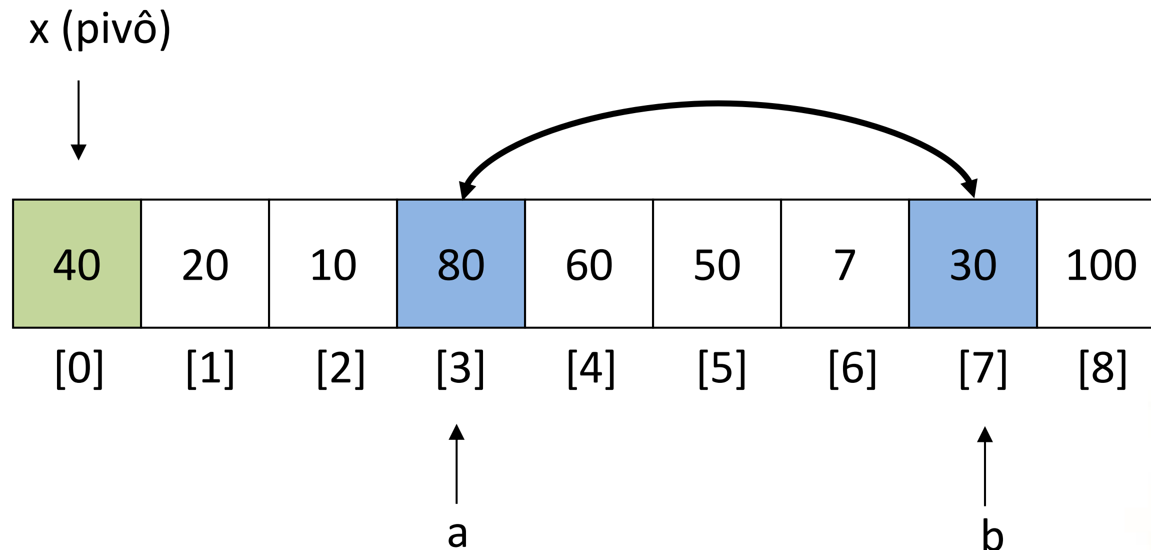
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
```



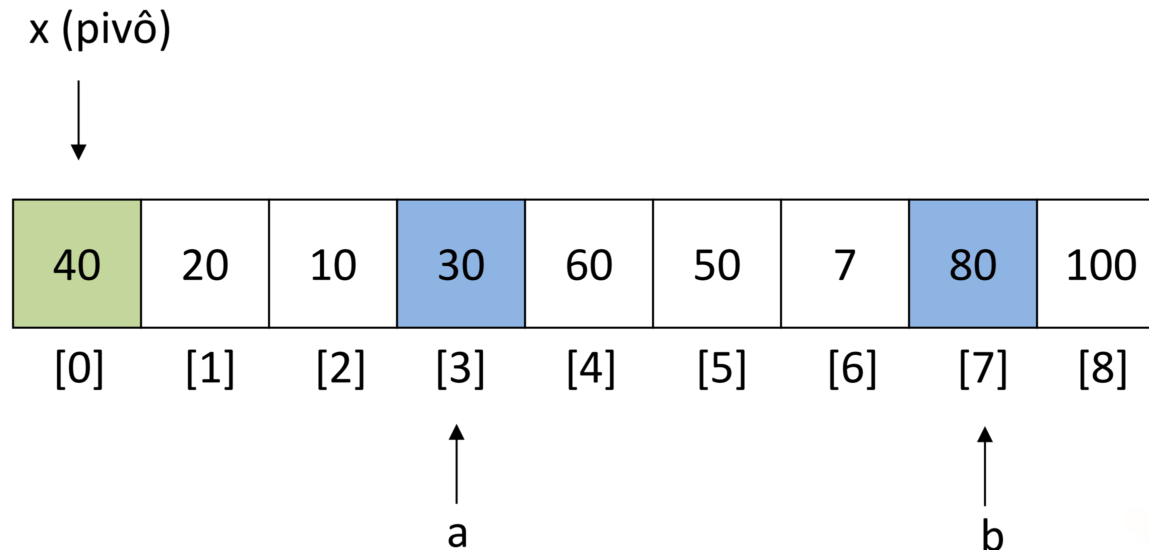
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
```



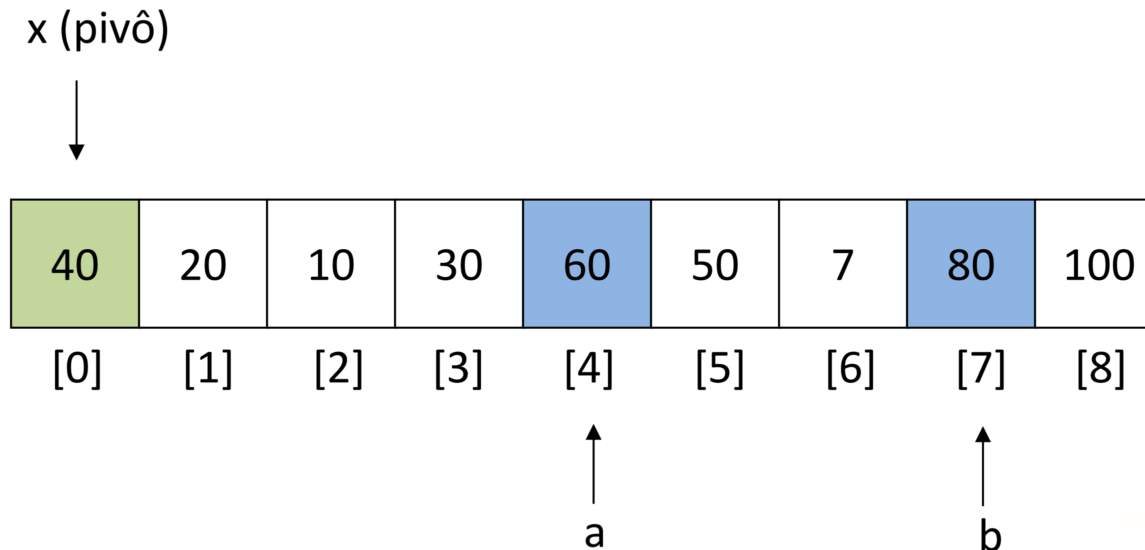
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
```



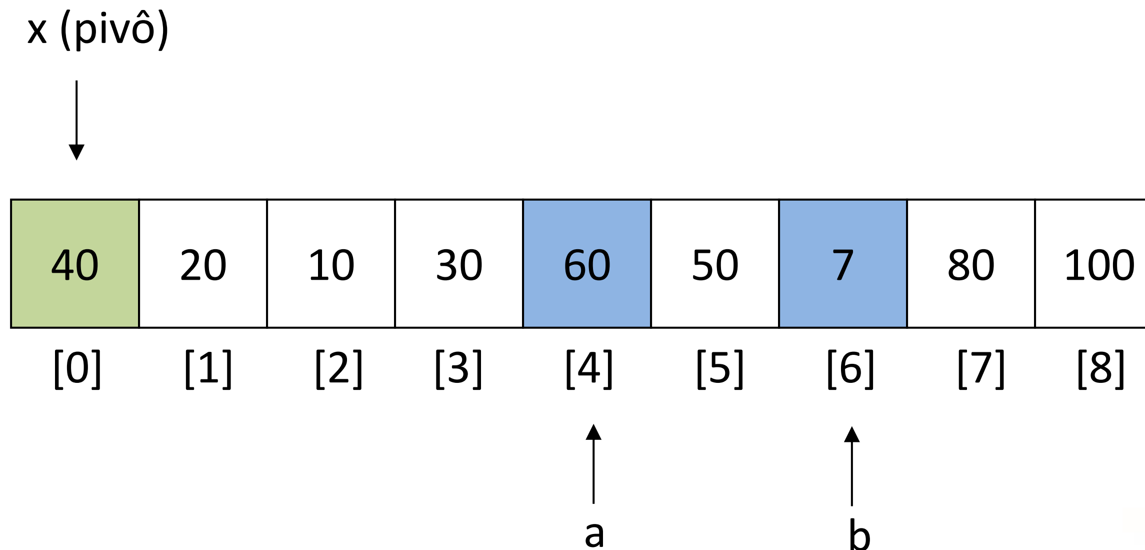
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



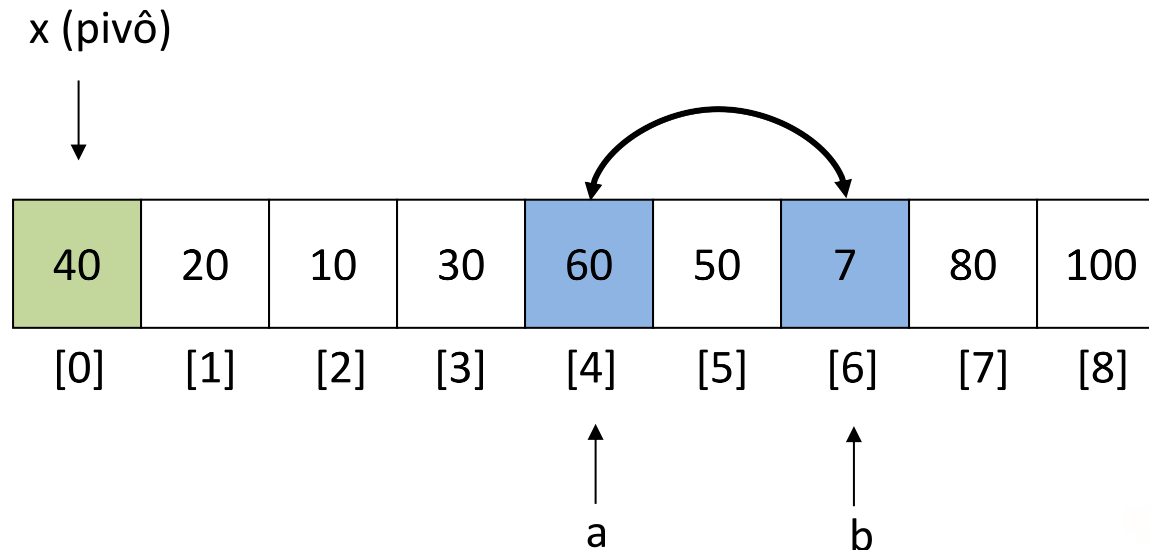
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



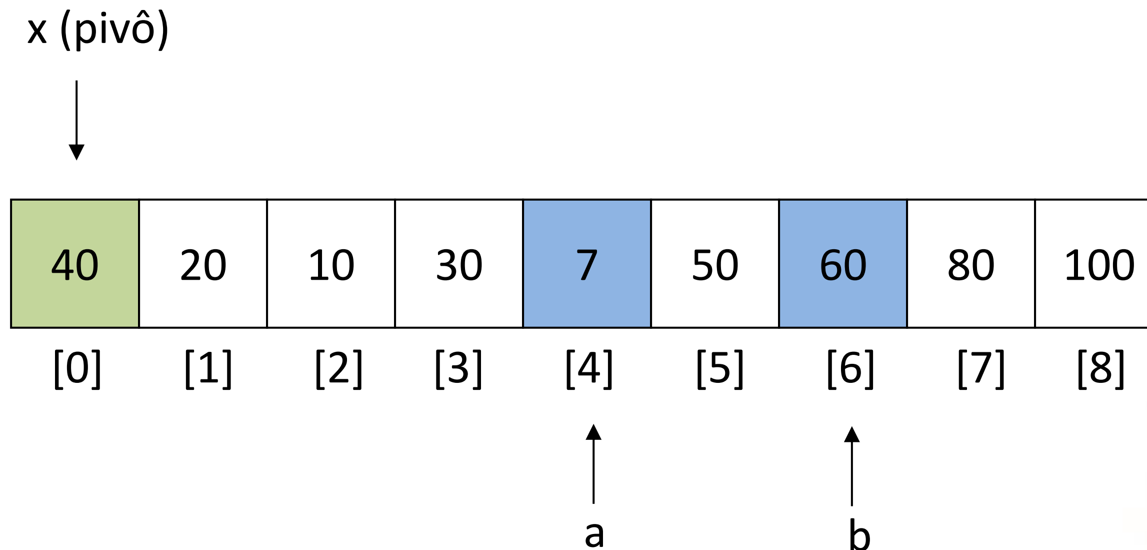
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



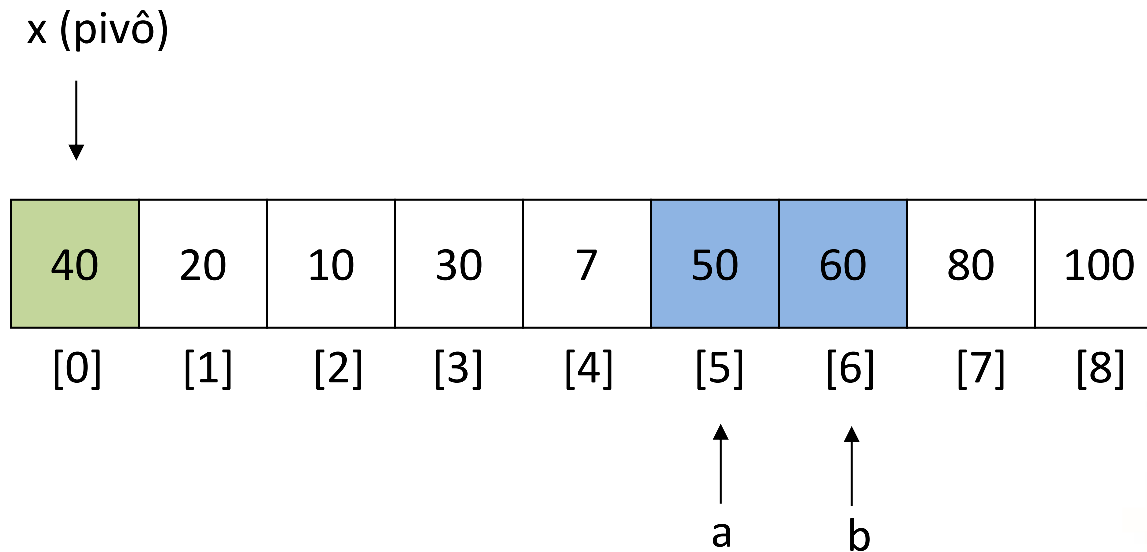
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



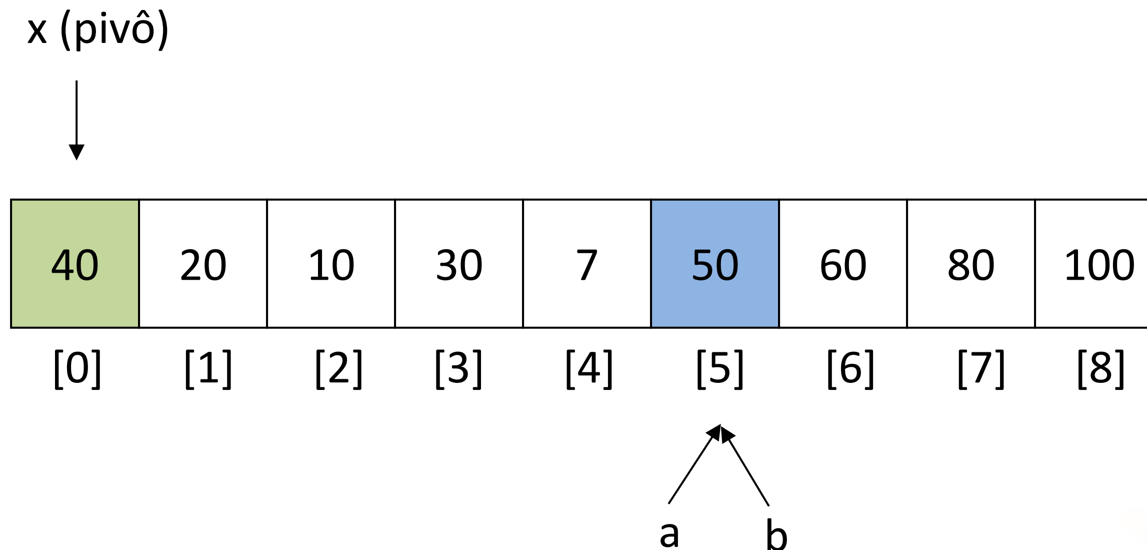
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



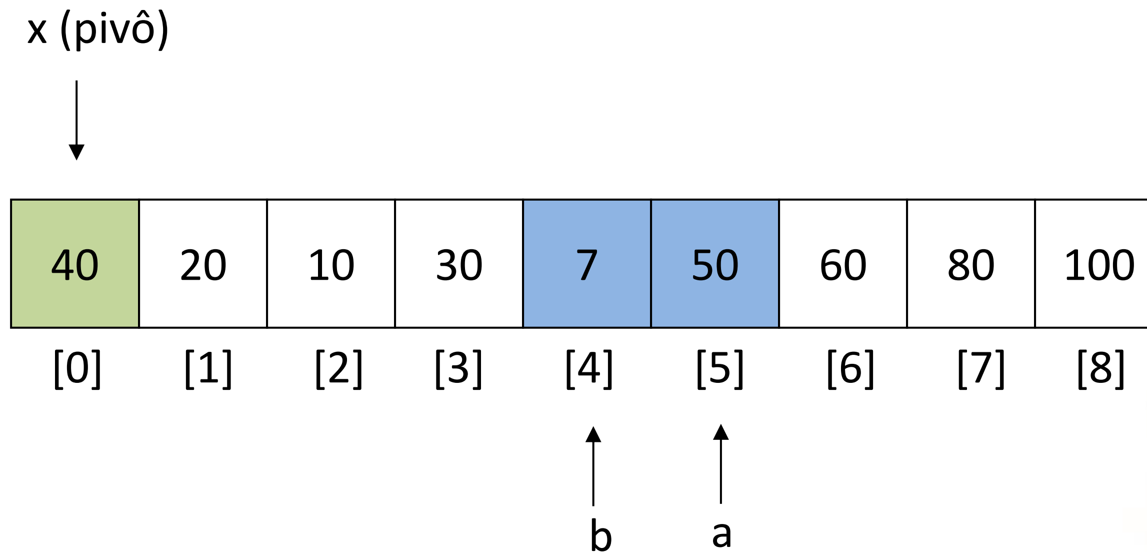
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



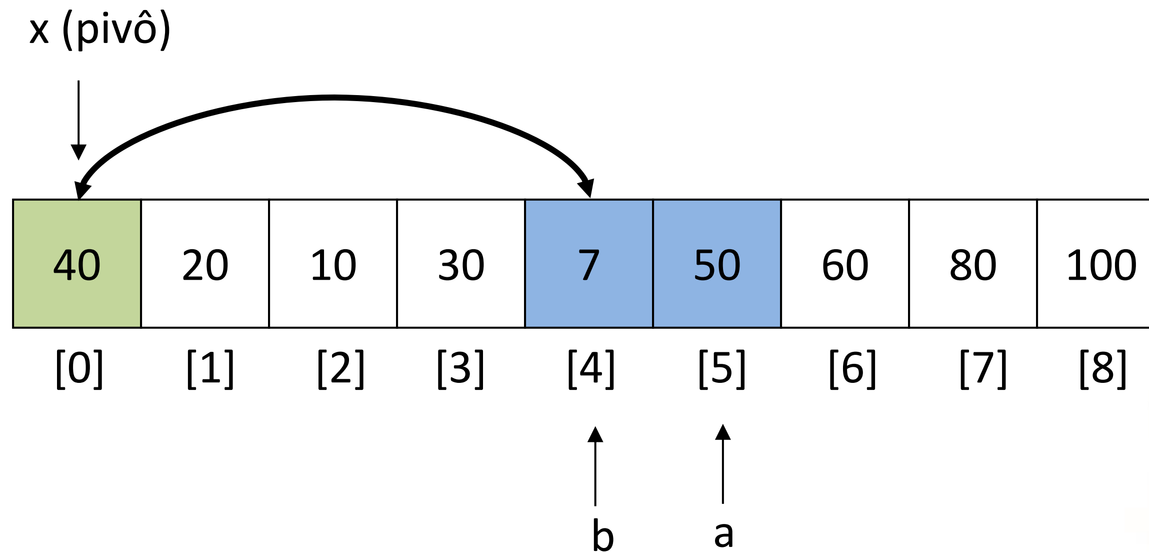
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



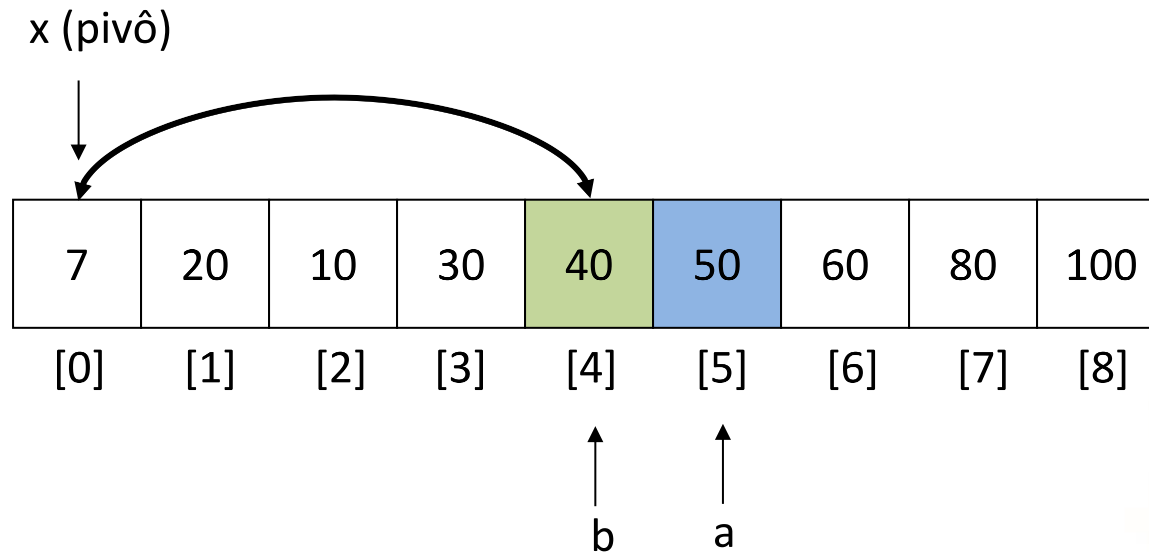
```
do{
  while (a < n && v[a] <= x)
    a++;
  while (v[b] > x)
    b--;
  if (a < b)
    troca(&v[a], &v[b]);
} while (a <= b);
```



```
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

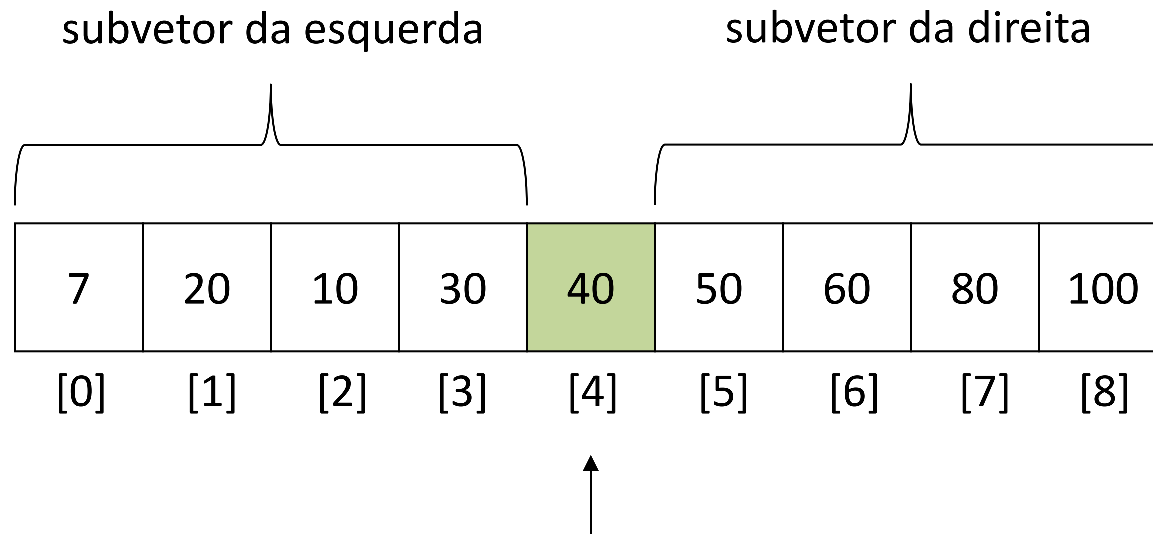
- Resultado da Partição:

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:



```
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

```c
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 && compInt(v[a],x) == 0)
        a++;
      while(compInt(v[b],x) == 1)
        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]);
  }
}
```

```c
int compInt(int a, int b)
{
  if (a > b)
    return 1;
  else
    return 0;
}
```

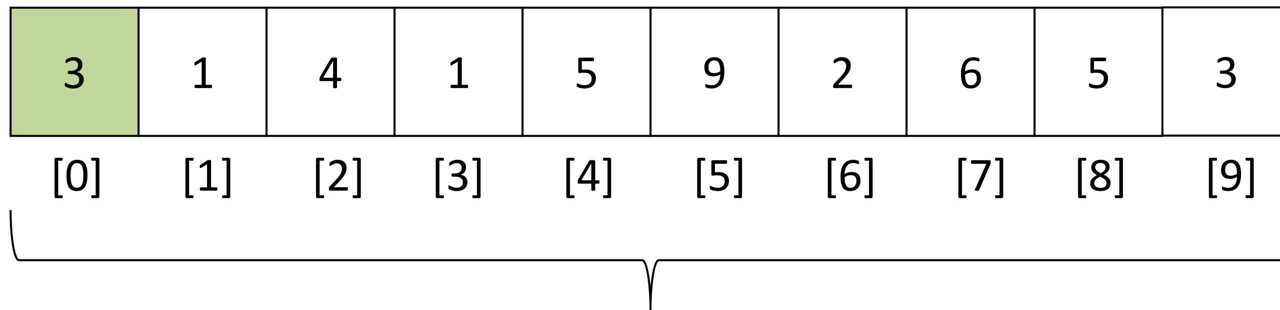
Quick Sort

(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) Selecione o pivô:



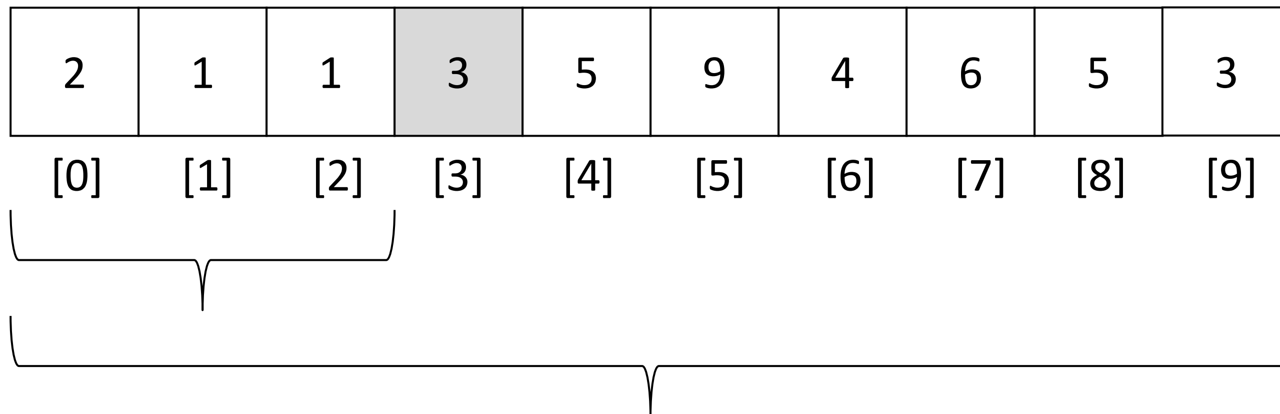
Quick Sort

(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]

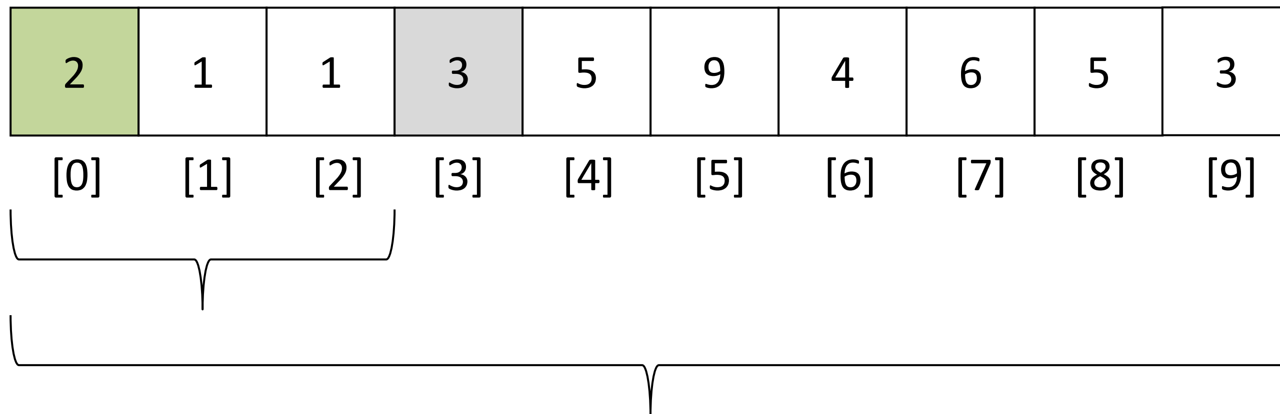
Quick Sort

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



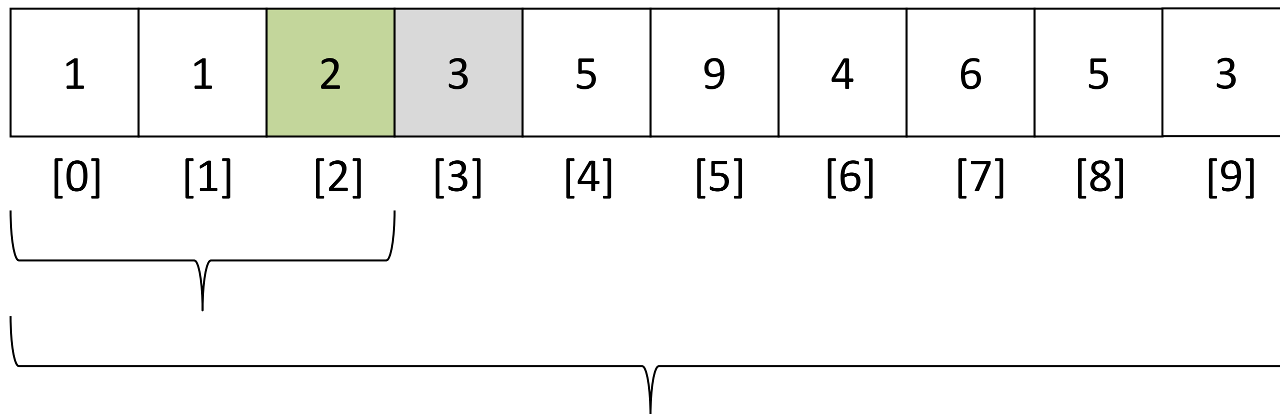
Quick Sort

(5) Selecione o pivô:



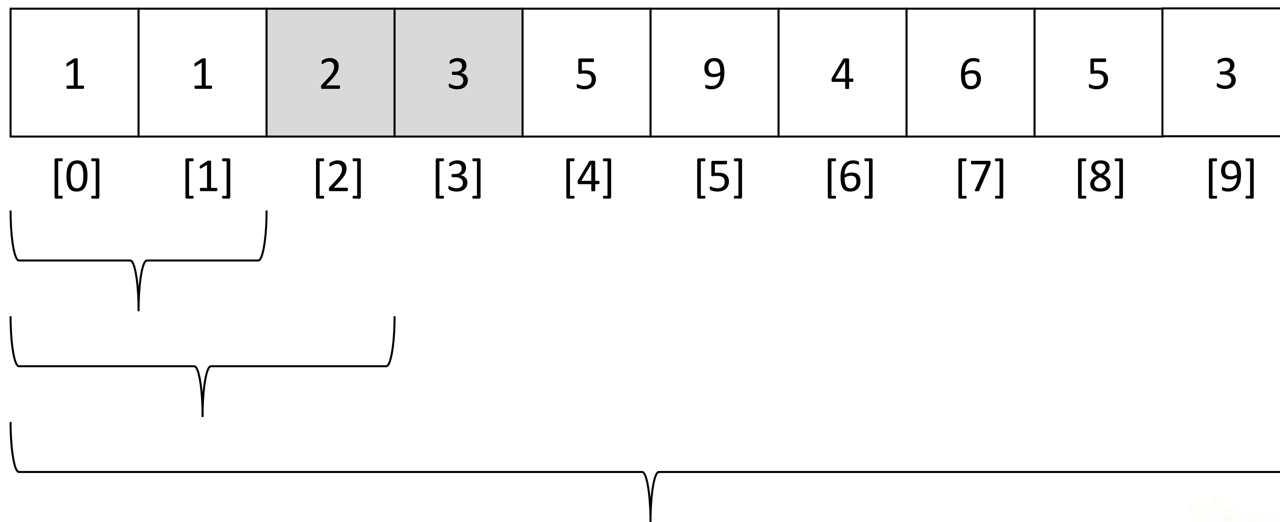
Quick Sort

(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:



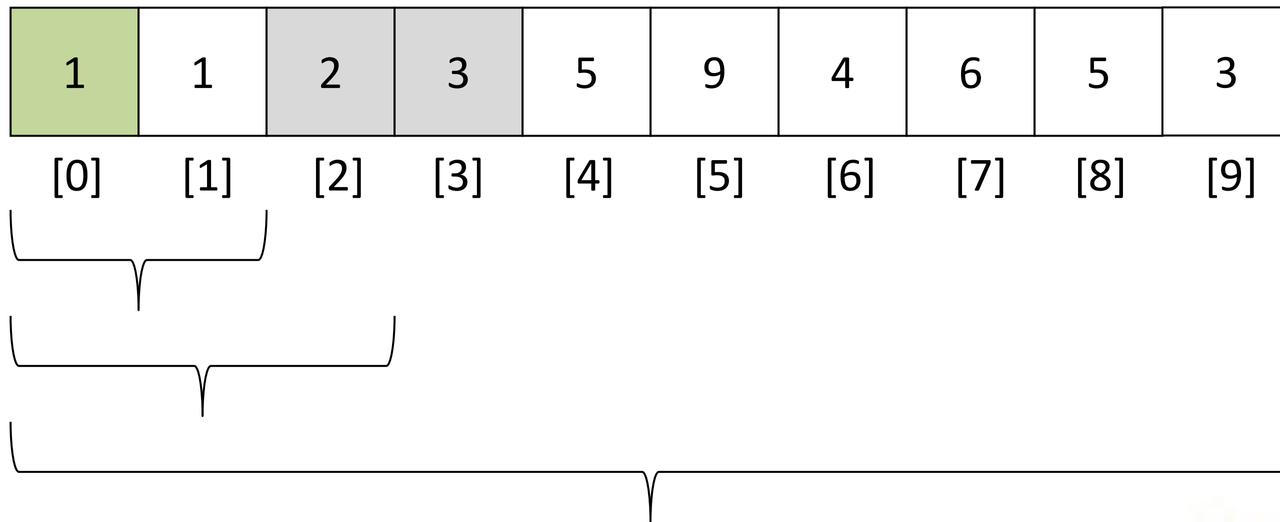
Quick Sort

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



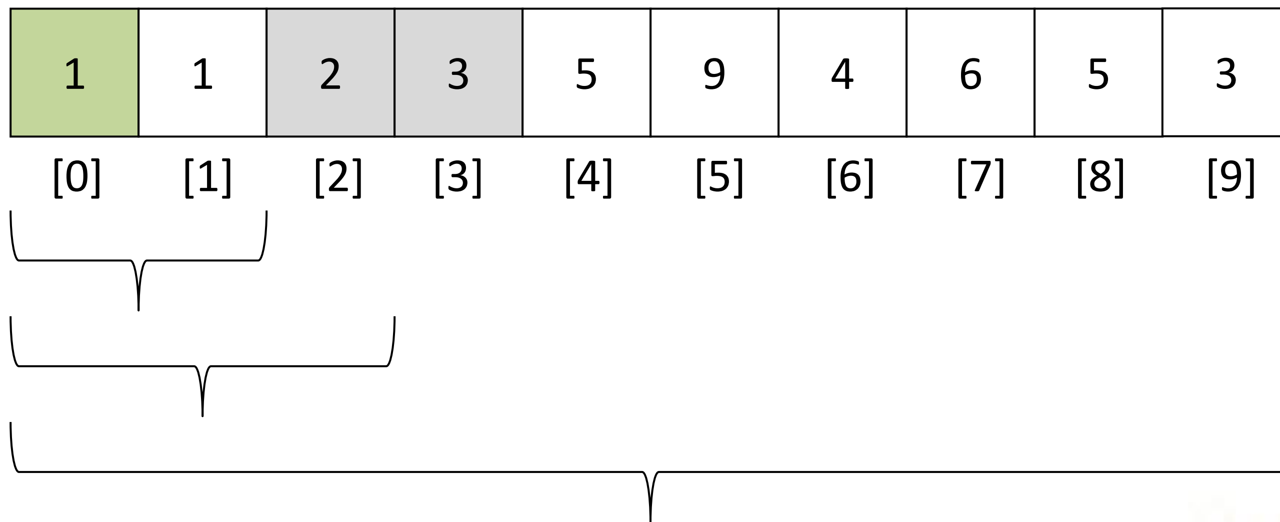
Quick Sort

(8) Selecione o pivô:



Quick Sort

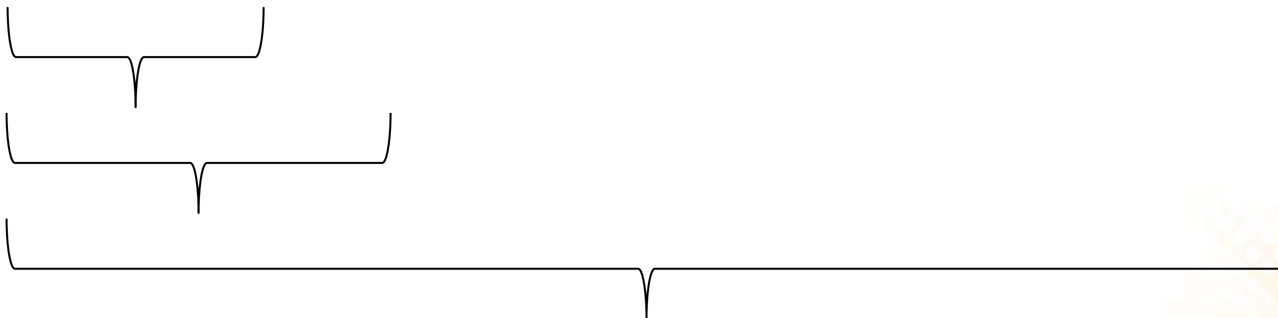
(9) Particione o vetor. Todos os elementos maiores que o pivô ficaram na direita e todos os elementos menores na esquerda. O pivô já está ordenado:



Quick Sort

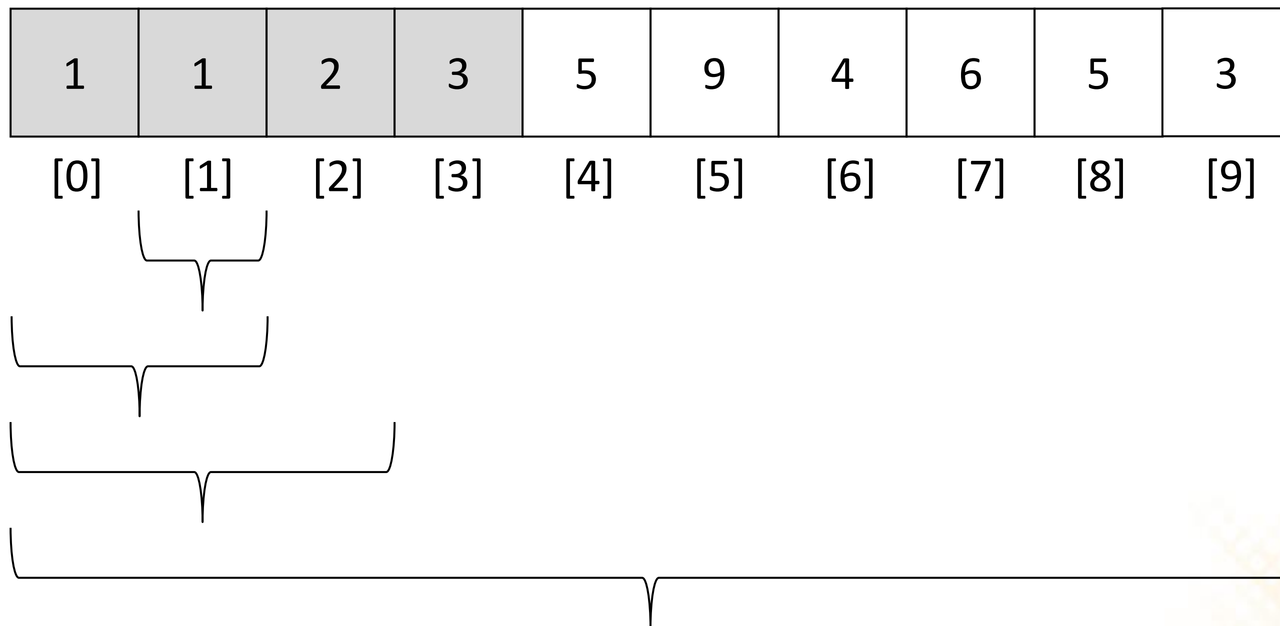
(10) Chame recursivamente o quick sort para o subvetor da esquerda. O tamanho do vetor da esquerda é zero. A recursão retorna.

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



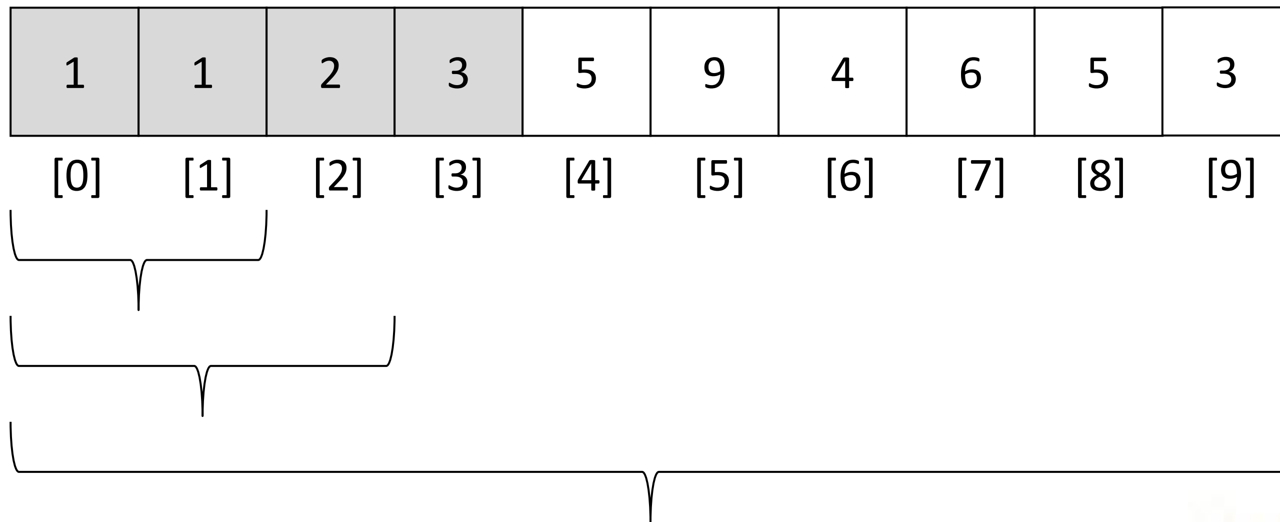
Quick Sort

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



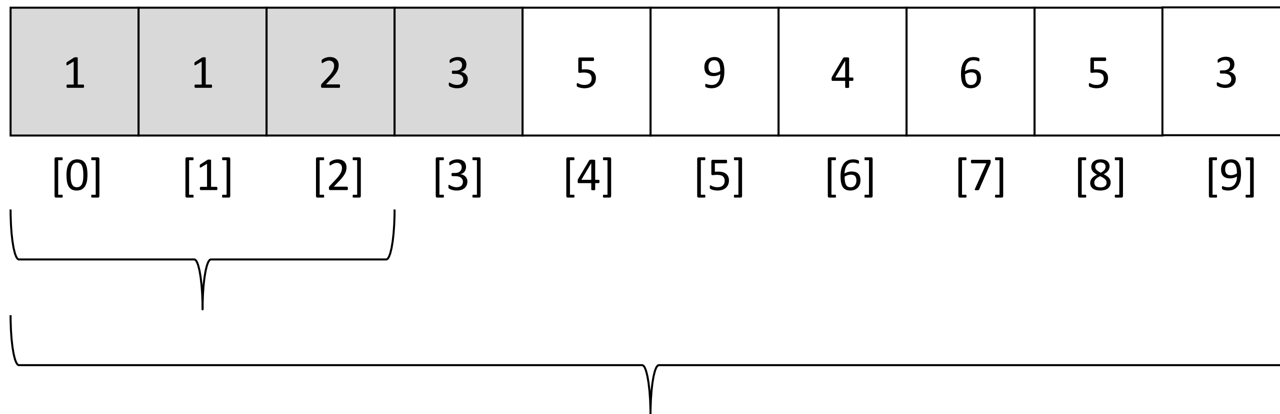
Quick Sort

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



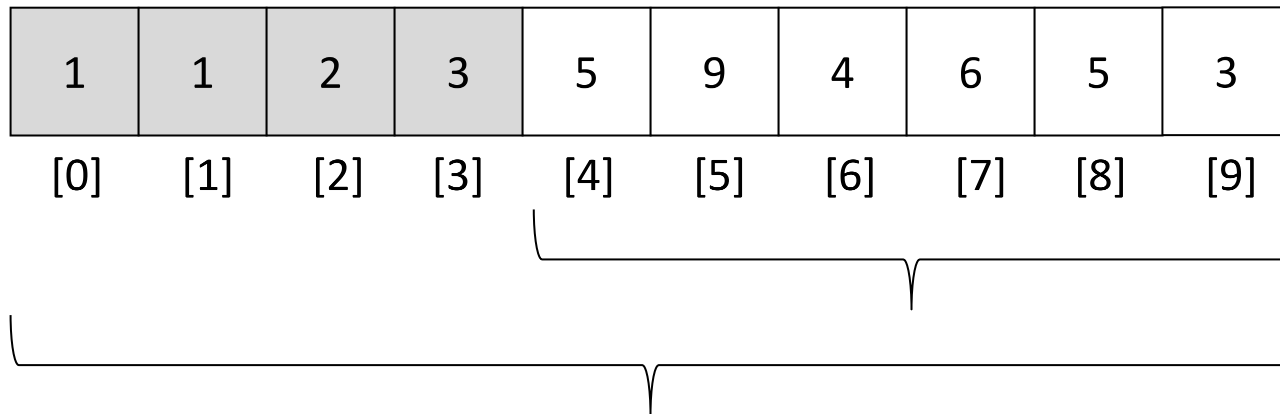
Quick Sort

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



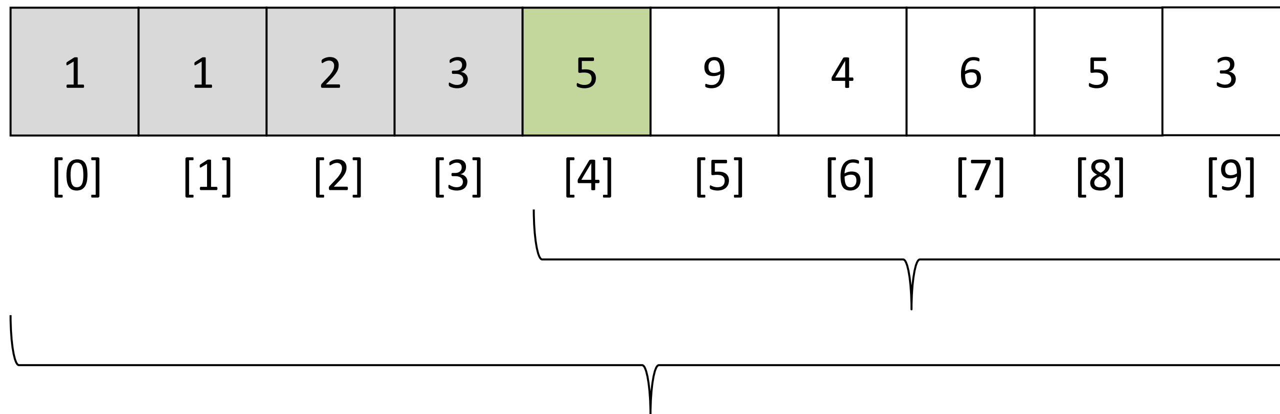
Quick Sort

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



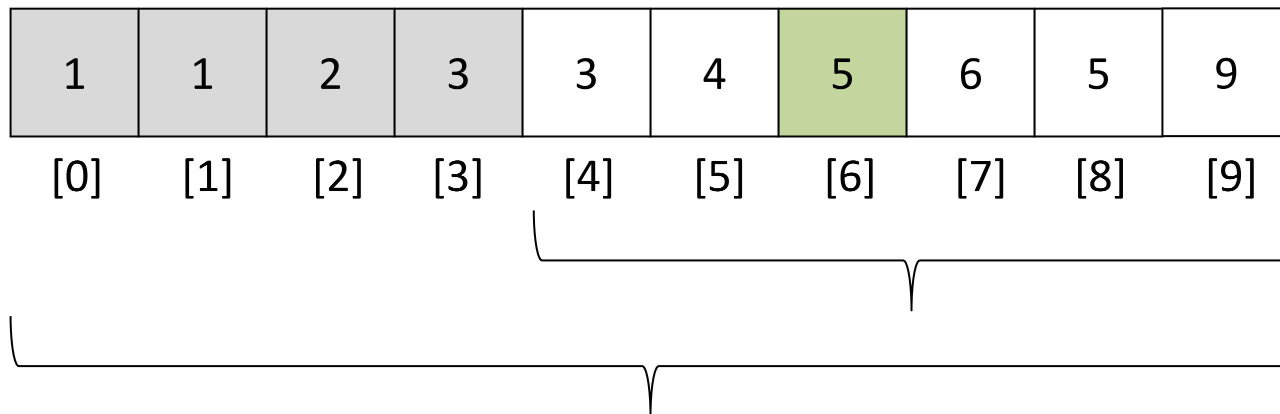
Quick Sort

(15) Selecione o pivô:



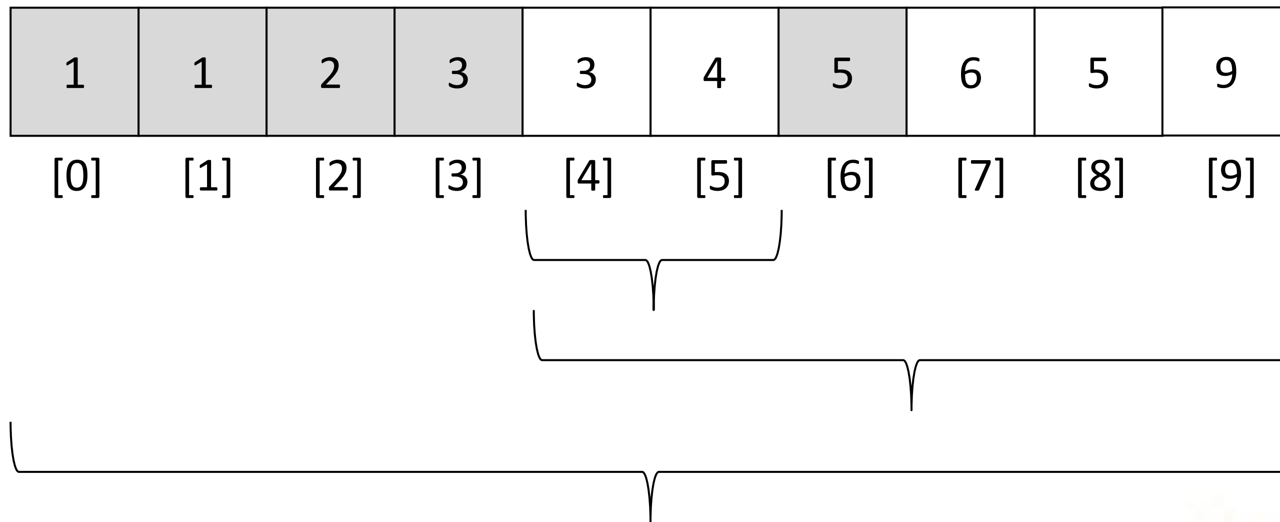
Quick Sort

(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:



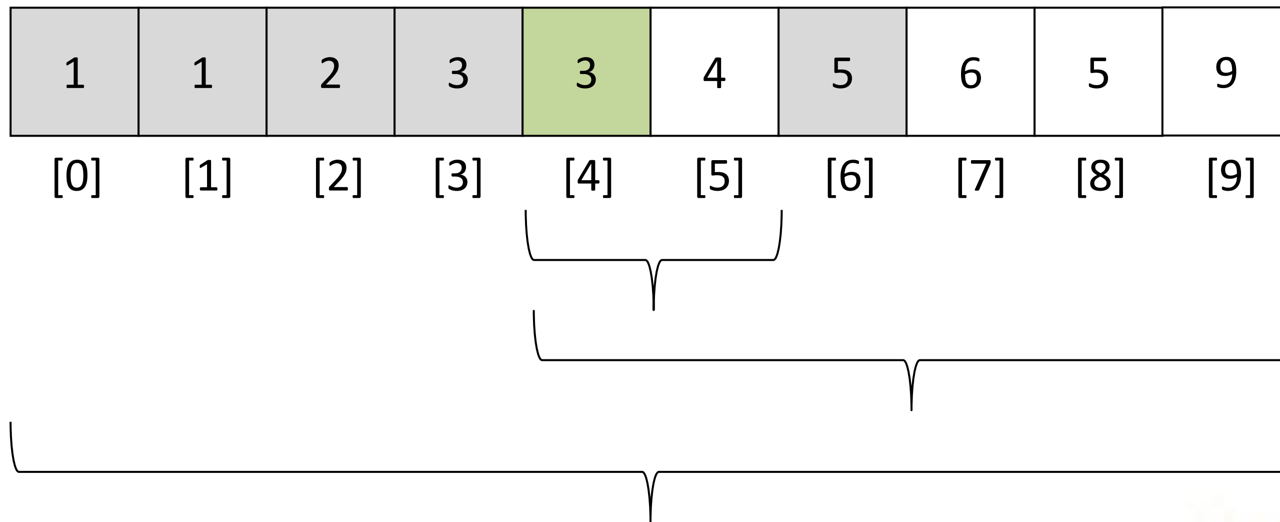
Quick Sort

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



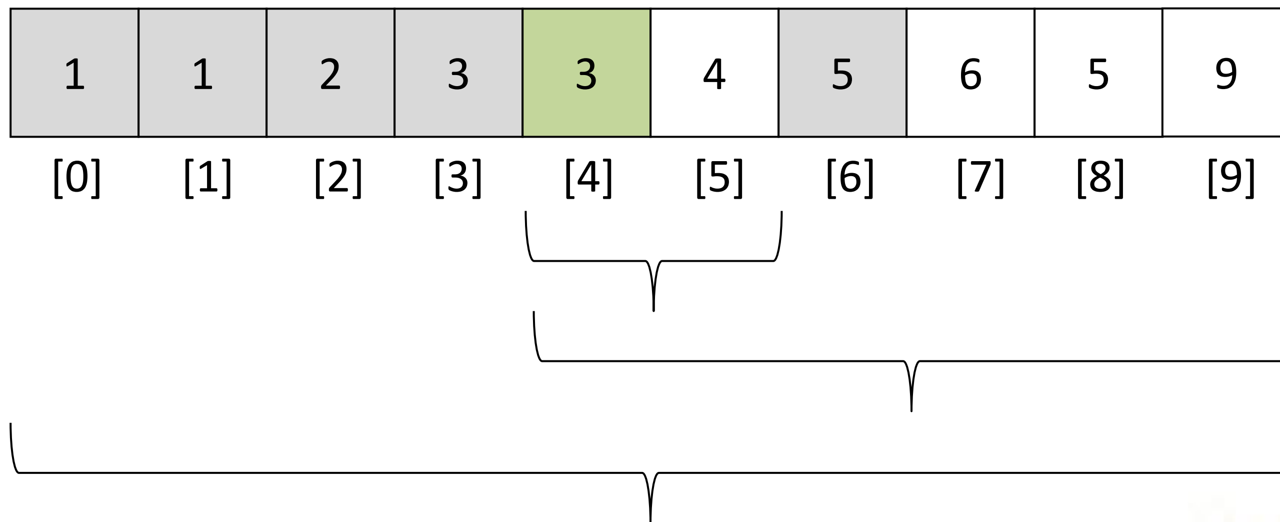
Quick Sort

(18) Selecione o pivô:



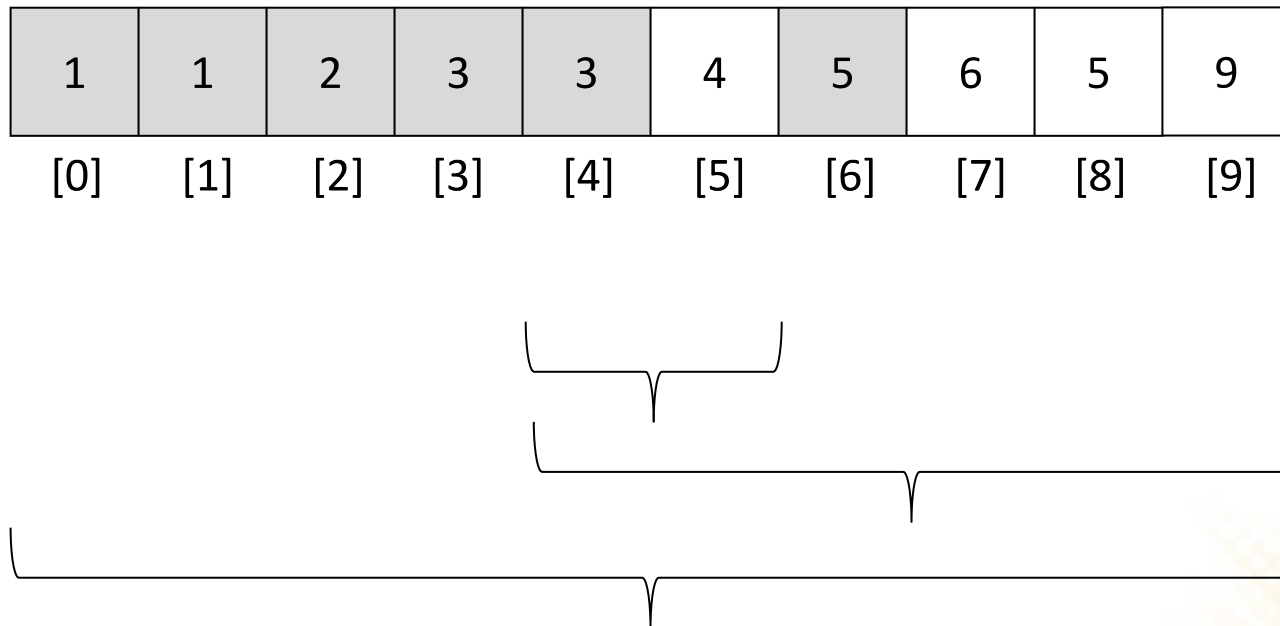
Quick Sort

(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:



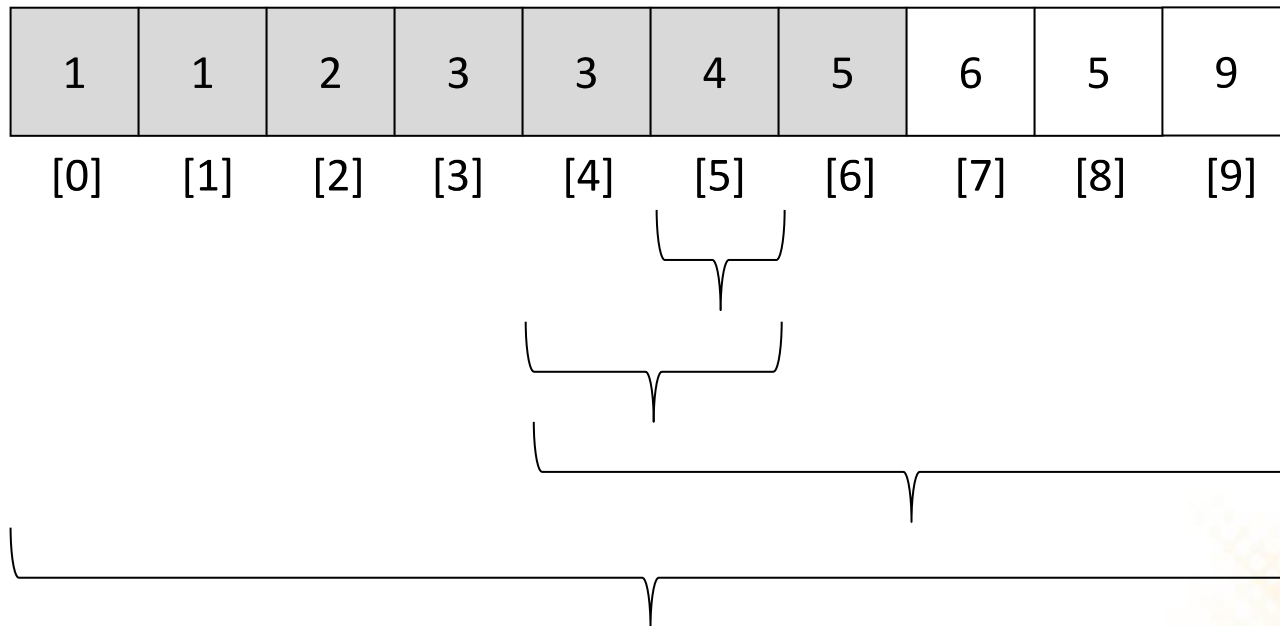
Quick Sort

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



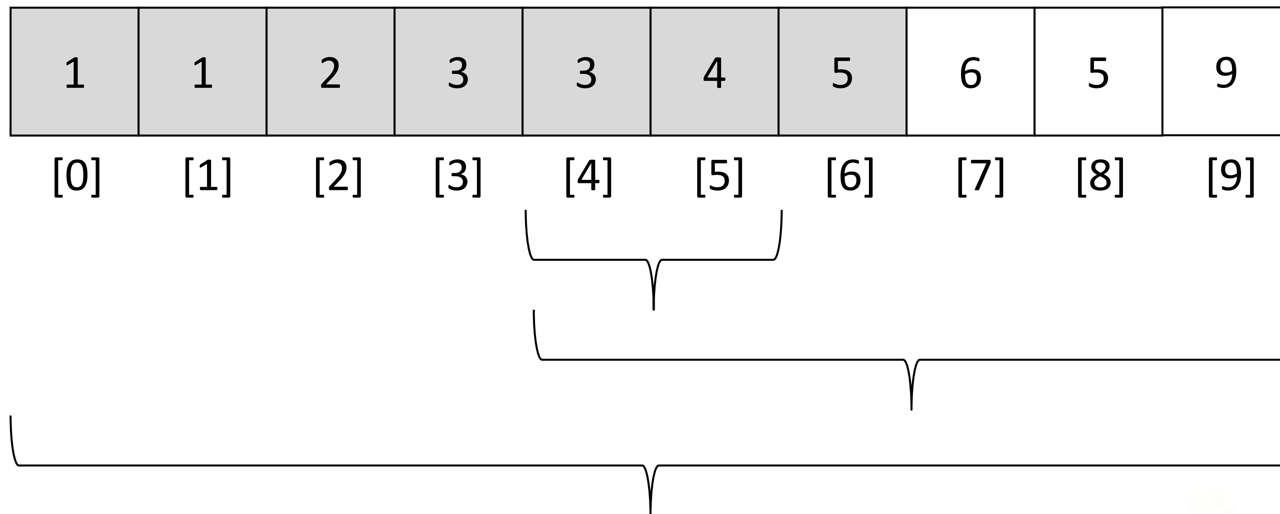
Quick Sort

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



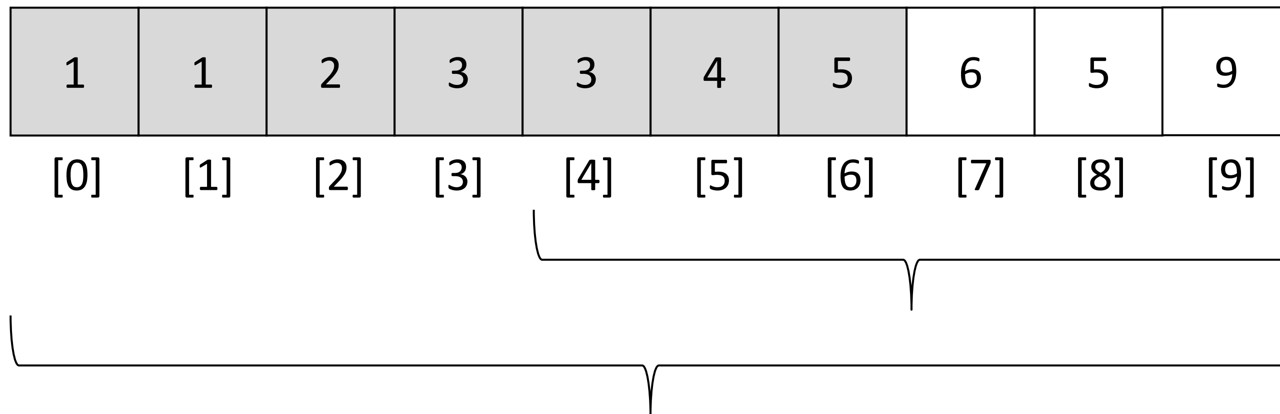
Quick Sort

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



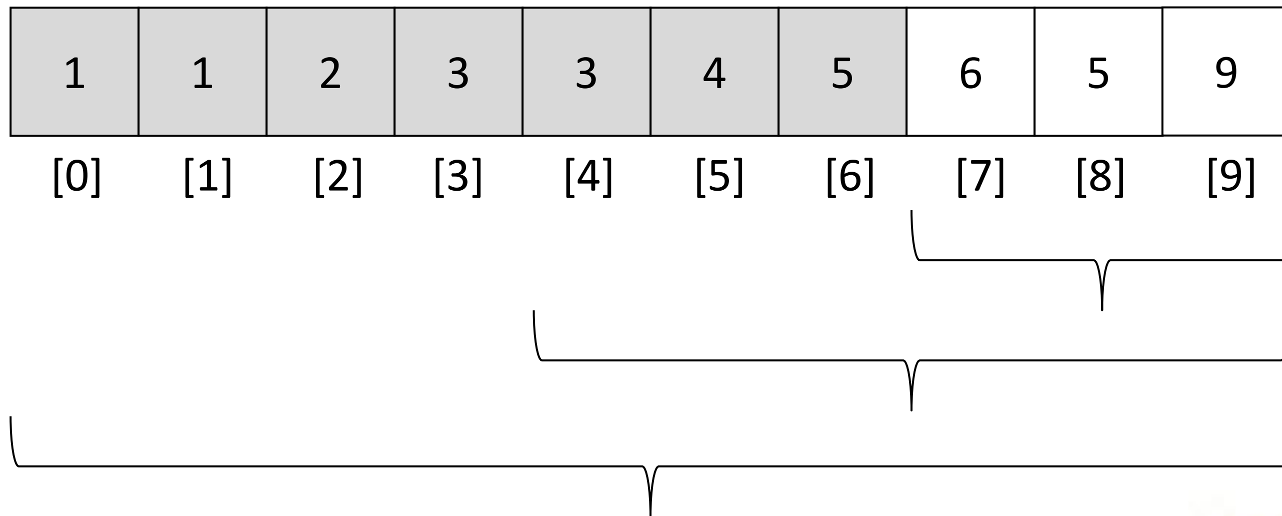
Quick Sort

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



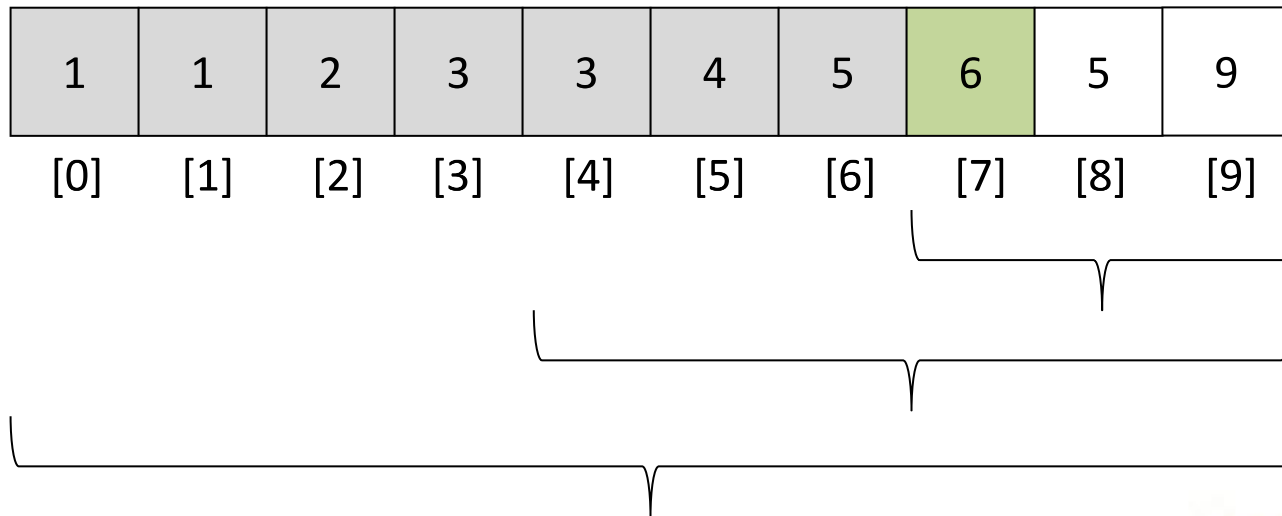
Quick Sort

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



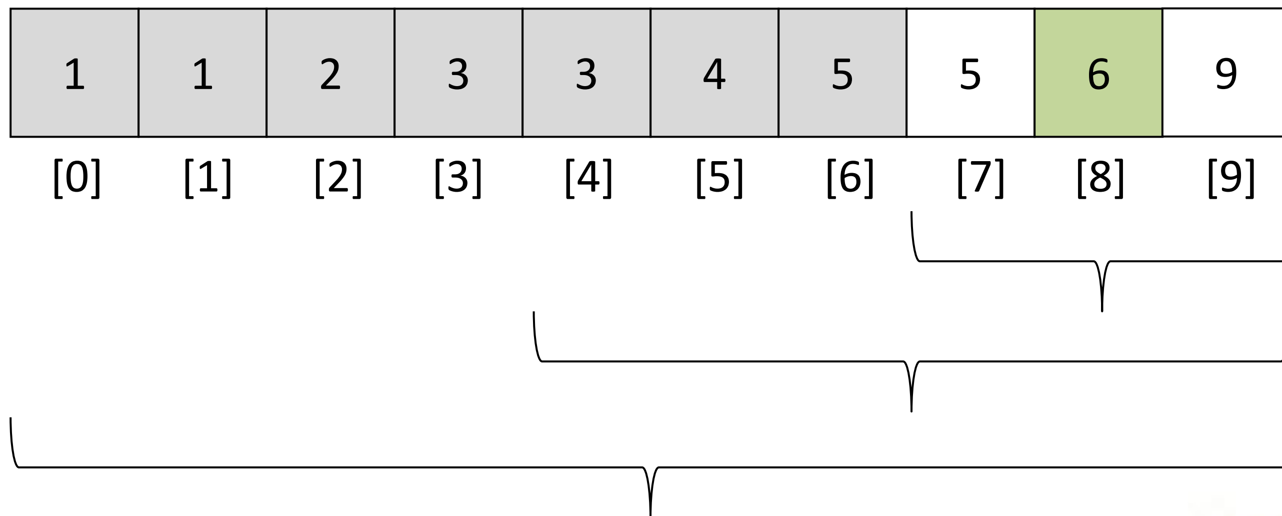
Quick Sort

(25) Selecione o pivô:



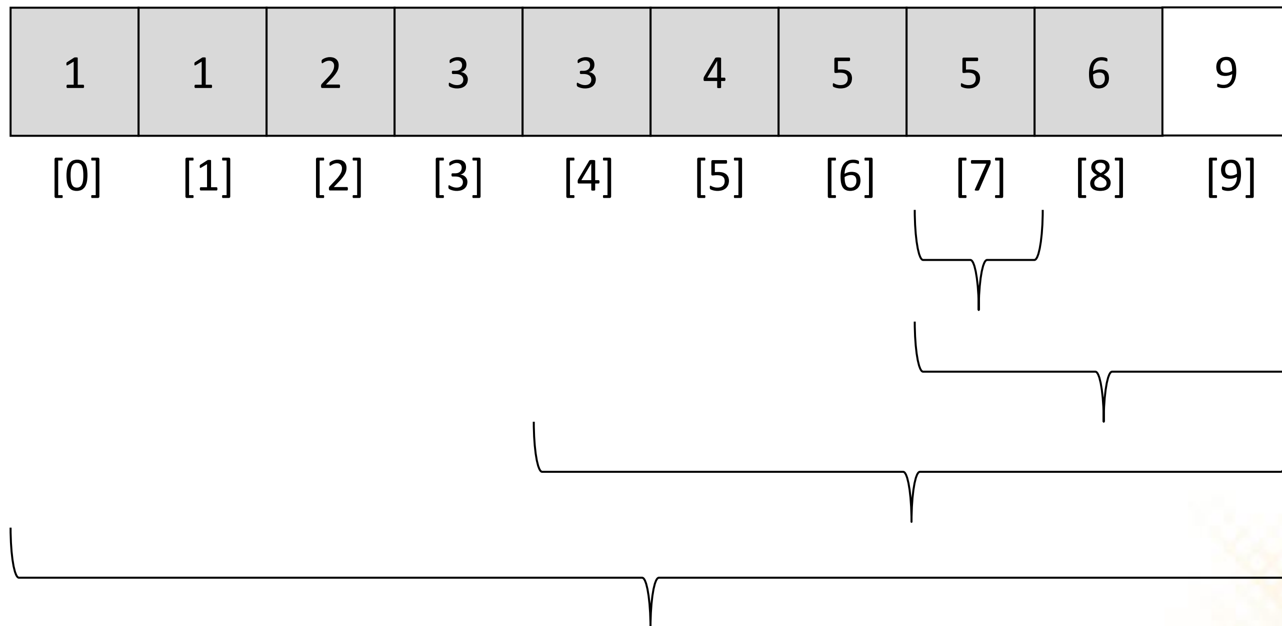
Quick Sort

(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:



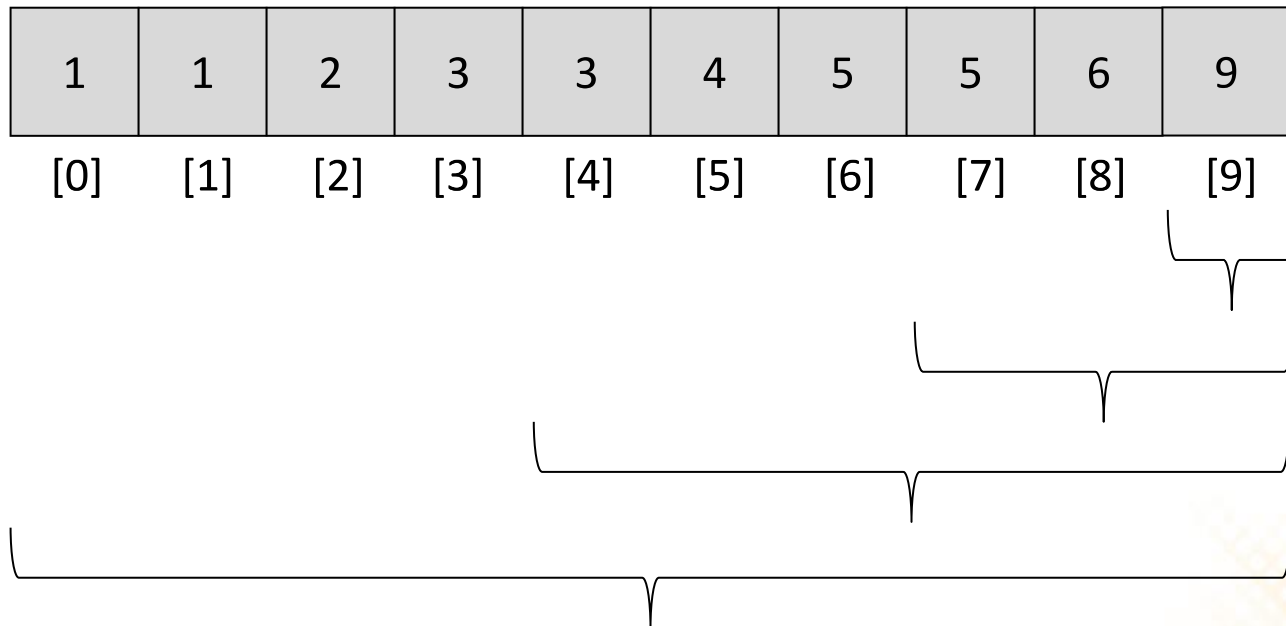
Quick Sort

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



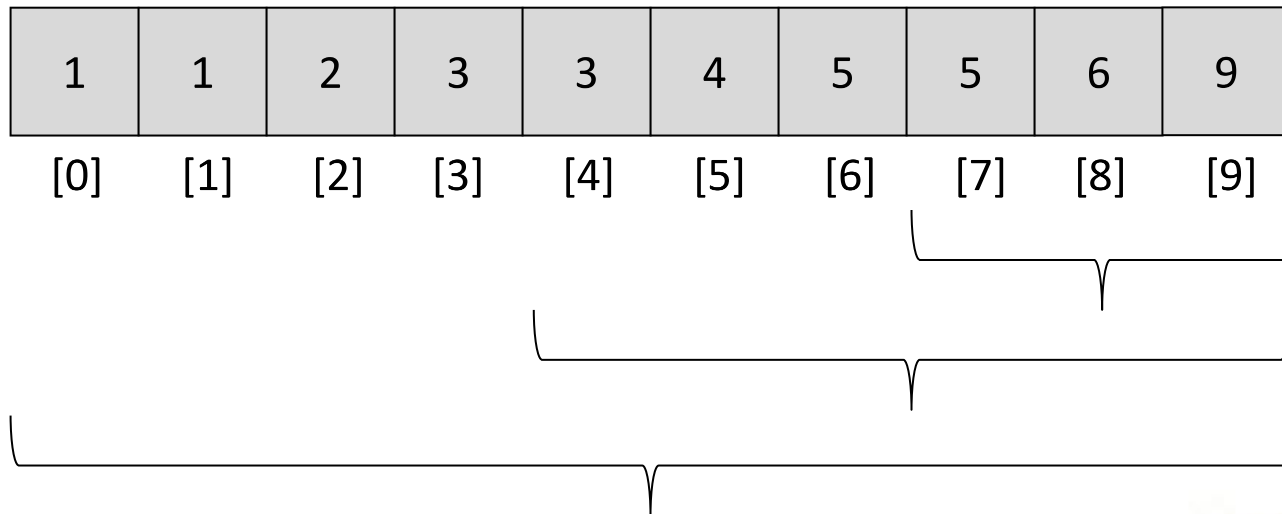
Quick Sort

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



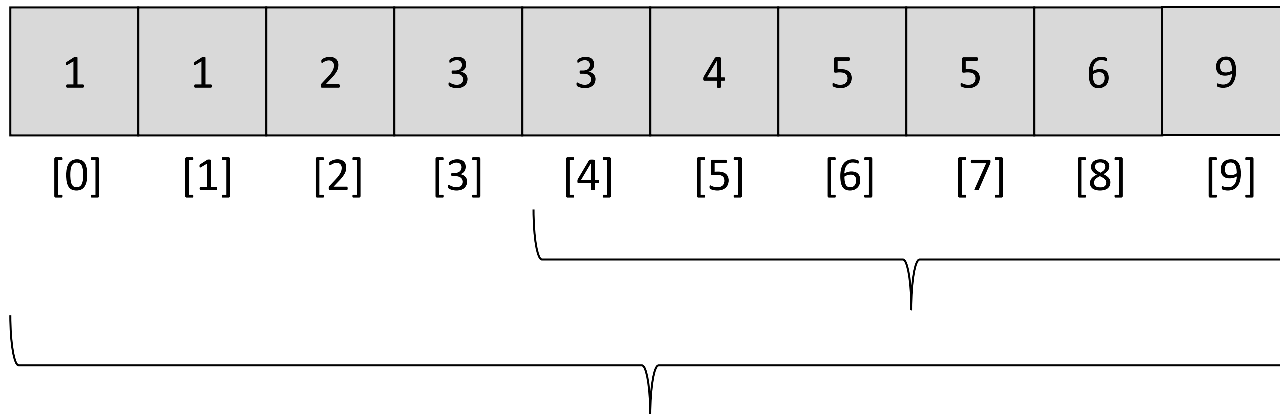
Quick Sort

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



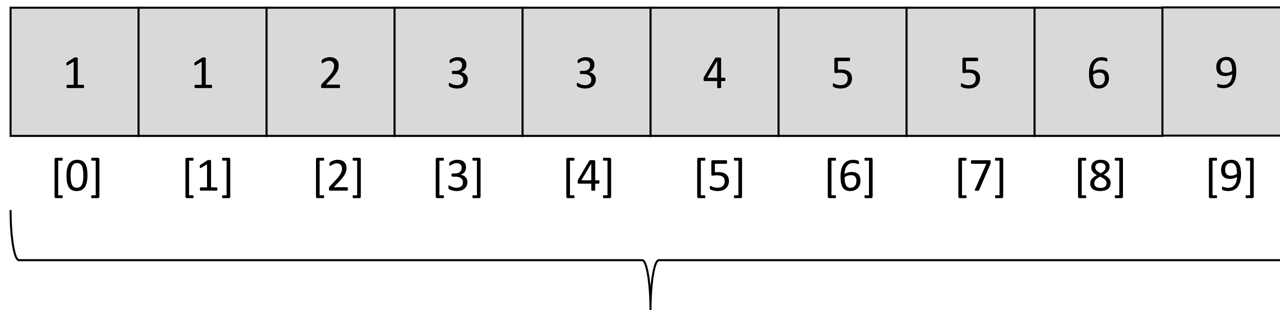
Quick Sort

(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: O(n log(n))
- **Pior caso:**
  - pivô é o maior elemento e algoritmo recai em ordenação bolha;
- **Caso médio:**
  - Complexidade: O(n log(n))

Quick Sort da `stdlib.h`

```
void qsort(void * v, int n, int tam,
           int (*cmp)(const void *, const void *));
```

- Parâmetros:
  - **v:** vetor de ponteiros genéricos
  - **n:** número de elementos do vetor
  - **tam:** tamanho em bytes de cada elemento (use sizeof para especificar)
  - **cmp:** ponteiro para a função que compara elementos genéricos. Ela deve retornar < 0 se a < b, >0 se a > b e 0 se a == b:

```
int nome(const void * a, const void * b);
```

Quick Sort da `stdlib.h`

```
void qsort(void * v, int n, int tam,
           int (*cmp)(const void *, const void *));
```

- Exemplo de função de comparação:

const é para garantir que a função não modificará os valores dos elementos

```
static int compFloat(const void * a, const void * b)
{
  float * aa = (float *)a; /* converte os ponteiros genericos */
  float * bb = (float *)b;
  if (*aa > *bb)
    return 1;
  else if (*aa < *bb)
    return -1;
  else
    return 0;
}
```

Quick Sort da `stdlib.h`

```
void qsort(void * v, int n, int tam,
           int (*cmp)(const void *, const void *));
```

- Exemplo de chamada:

```
int main (void)
{
   int i;
   float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};

   qsort(v, 8, sizeof(float), compFloat);

   for (i=0; i<8; i++)
      printf("%f \n", v[i]);

   return 0;
}
```

qsort – Exemplo 2

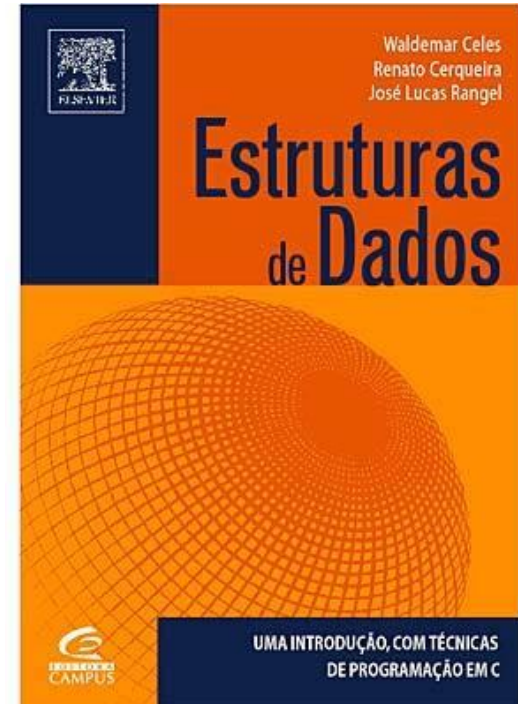
```
struct aluno
{
  int matricula;
  char nome[41];
};
typedef struct aluno Aluno;

static int compPStructStr(const void * a, const void * b)
{
  Aluno **aa = (Aluno **)a;
  Aluno **bb = (Aluno **)b;
  return strcmp((*aa)->nome, (*bb)->nome);
}

int main (void)
{
  Aluno *alunos[6];
  Aluno **p;
  int i;
  alunos[0] = (Aluno*)malloc(sizeof(Aluno));
  alunos[0]->matricula = 82135123;
  strcpy(alunos[0]->nome, "Julia");
```

```
  alunos[1] = (Aluno*)malloc(sizeof(Aluno));
  alunos[1]->matricula = 51364125;
  strcpy(alunos[1]->nome, "Joao");

  alunos[2] = (Aluno*)malloc(sizeof(Aluno));
  alunos[2]->matricula = 62151578;
  strcpy(alunos[2]->nome, "Bruno");

  alunos[3] = (Aluno*)malloc(sizeof(Aluno));
  alunos[3]->matricula = 35641215;
  strcpy(alunos[3]->nome, "Pedro");

  alunos[4] = (Aluno*)malloc(sizeof(Aluno));
  alunos[4]->matricula = 45612681;
  strcpy(alunos[4]->nome, "Maria");

  alunos[5] = (Aluno*)malloc(sizeof(Aluno));
  alunos[5]->matricula = 2654951;
  strcpy(alunos[5]->nome, "Ana");

  qsort(alunos, 6, sizeof(Aluno*), compPStructStr);

  for (i = 0; i < 6; i++)
  printf("%s - %d \n", alunos[i]->nome, alunos[i]->matricula);

  return 0;
}
```

Leitura Complementar

- Waldemar Celes, Renato Cerqueira, José Lucas Rangel, **Introdução a Estruturas de Dados**, Editora Campus (2004).

- **Capítulo 16 – Ordenação**



Exercícios

Lista de Exercícios 08 – Ordenação