Algoritmos e Estruturas de Dados II

2º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues
Email: edwaldo.rodrigues@uemg.br

UNIVERSIDADE
DO ESTADO DE MINAS GERAIS
UEMG
UNIDADE DIVINÓPOLIS

Métodos de Ordenação - QuickSort

QuickSort

- Para uma ampla variedade de situações, é o método mais rápido que se conhece

- Provavelmente, o mais utilizado, junto ao MergeSort

- A cada passo do QuickSort, o problema de ordenação é dividido em dois problemas menores que são ordenados de maneira independente

QuickSort

- A parte complicada do método é a partição do problema em problemas menores

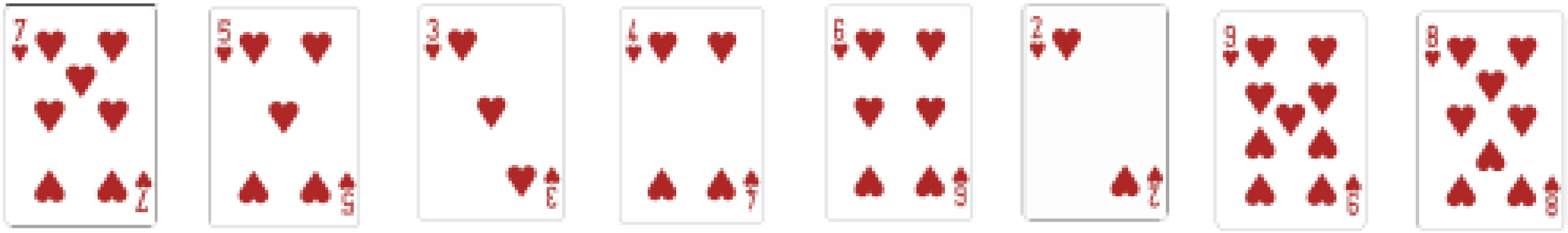
- O processo de partição é feito a partir da escolha de um pivô x

- Escolhido o pivô, um arranjo é dividido em duas partes:

  - A parte da esquerda com elementos menores ou iguais a x
  - A parte da direita com elementos maiores ou iguais a x

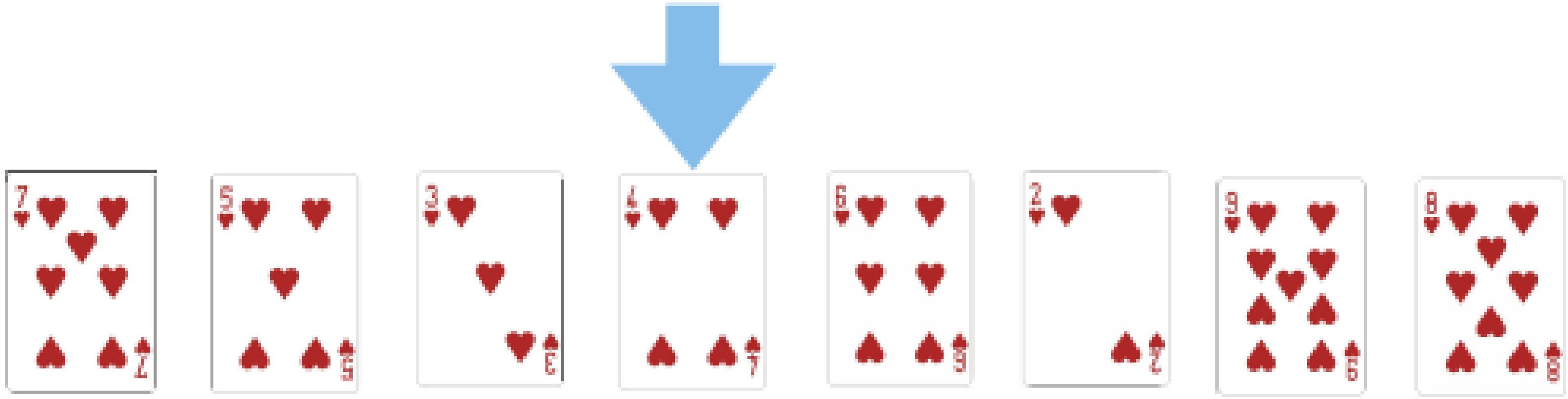
QuickSort

- Considere um arranjo com os seguintes elementos



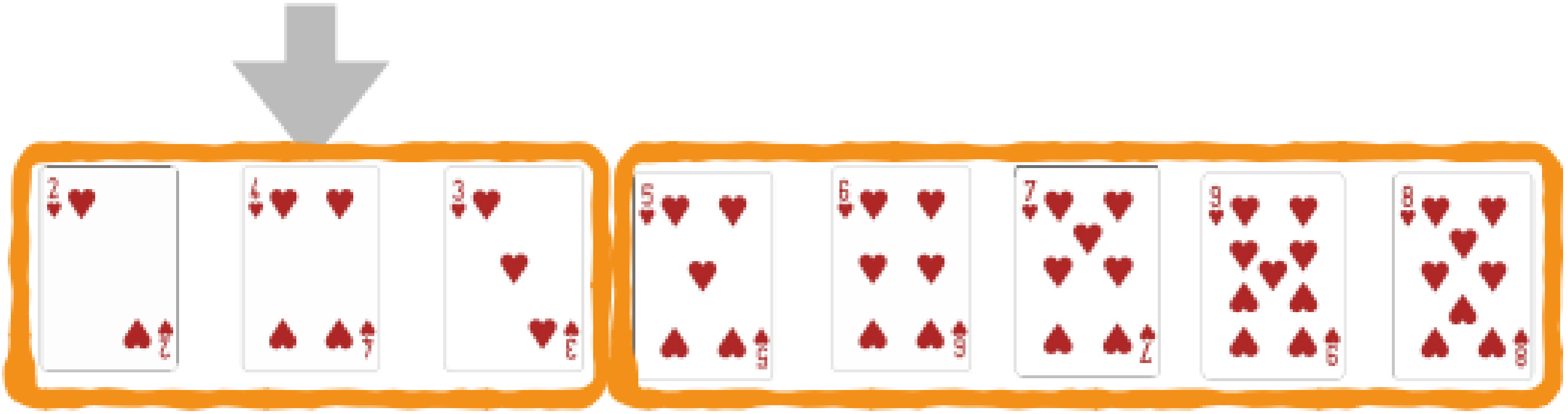
QuickSort

- Escolheremos um elemento como pivô



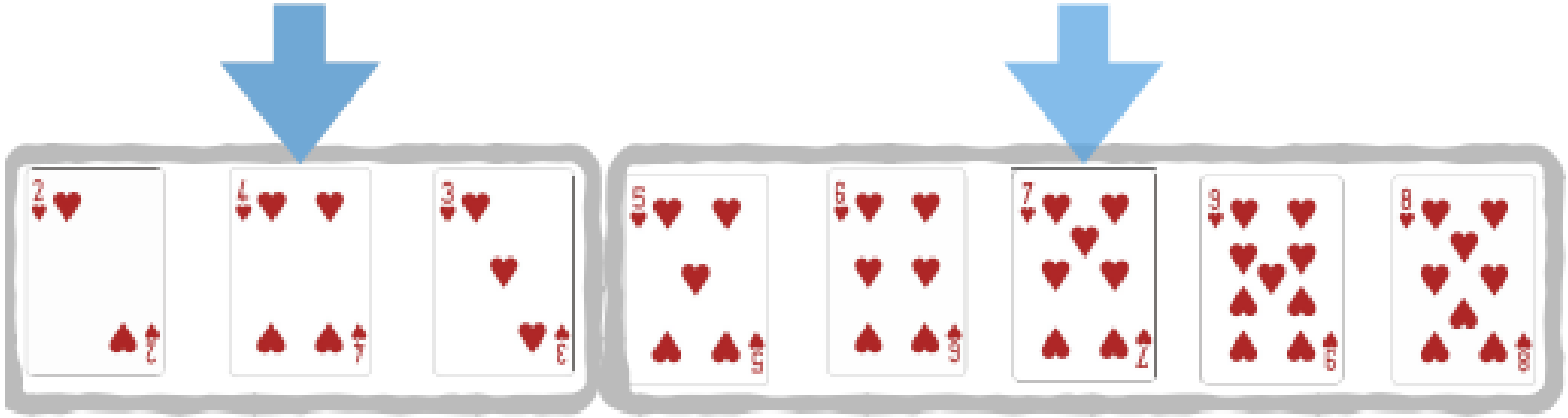
QuickSort

- Particionamos o arranjo de modo que elementos menores que o pivô fiquem à esquerda e elementos maiores fiquem à direita



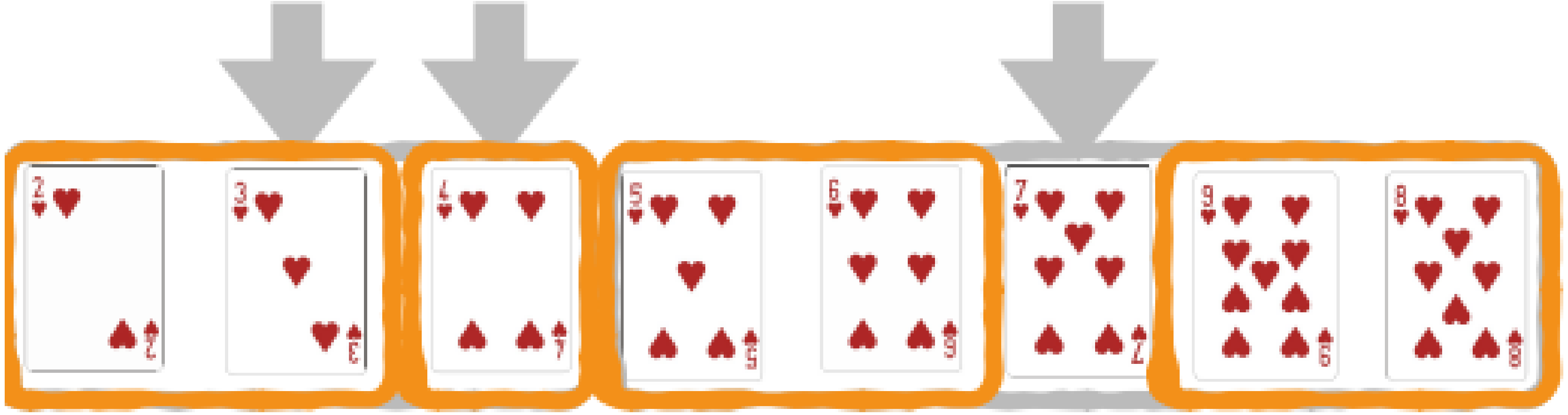
QuickSort

- Repetimos o processo para cada subproblema



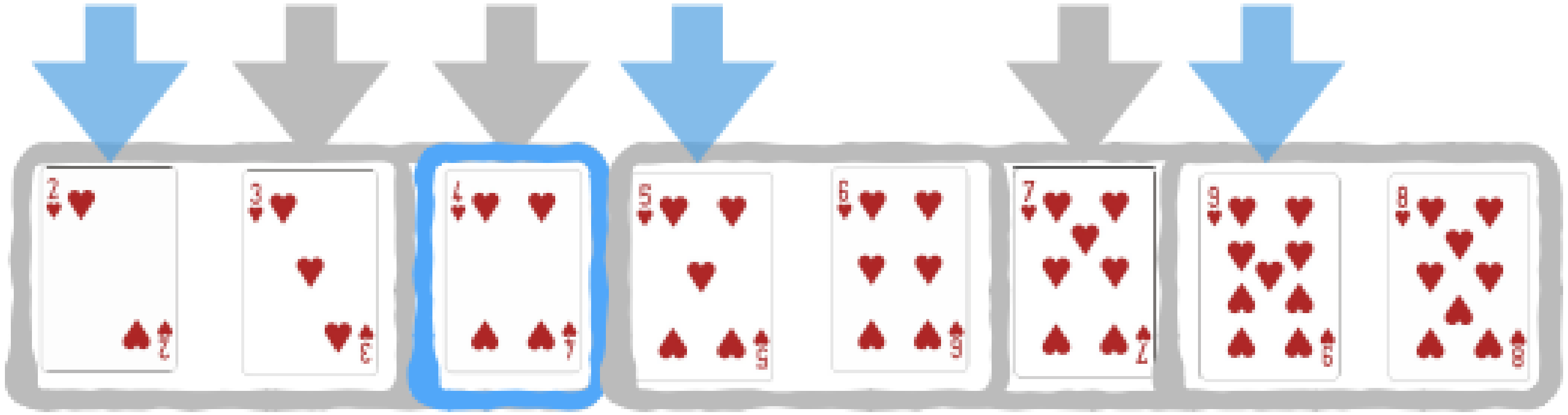
QuickSort

- Particionamos novamente o arranjo com os maiores à direita e menores fiquem à esquerda



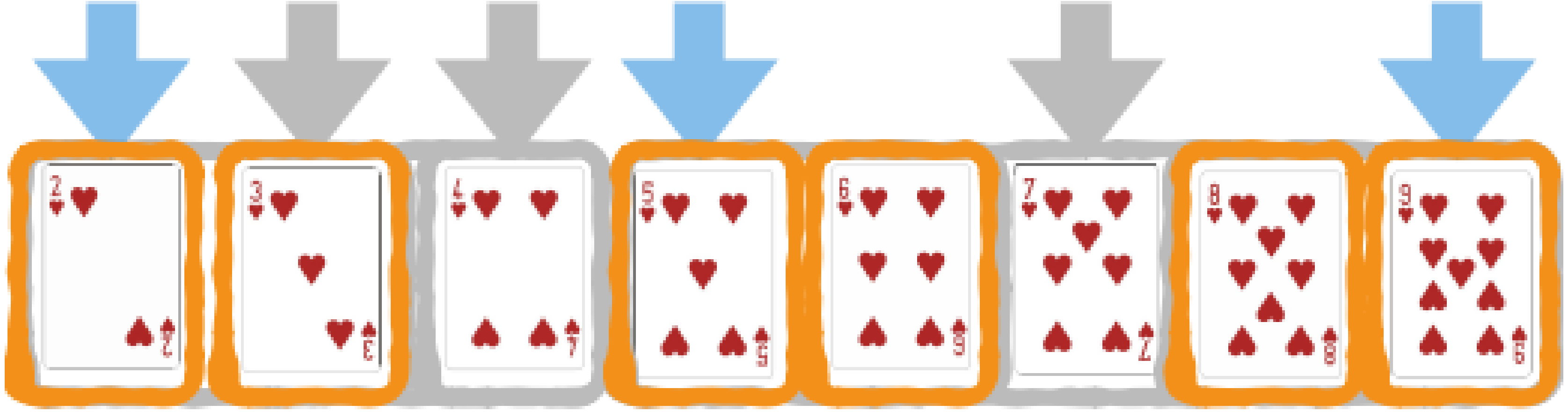
QuickSort

- Repetiremos o processo para cada subproblema. Porém, os arranjos de tamanho 1 já são considerados como ordenados



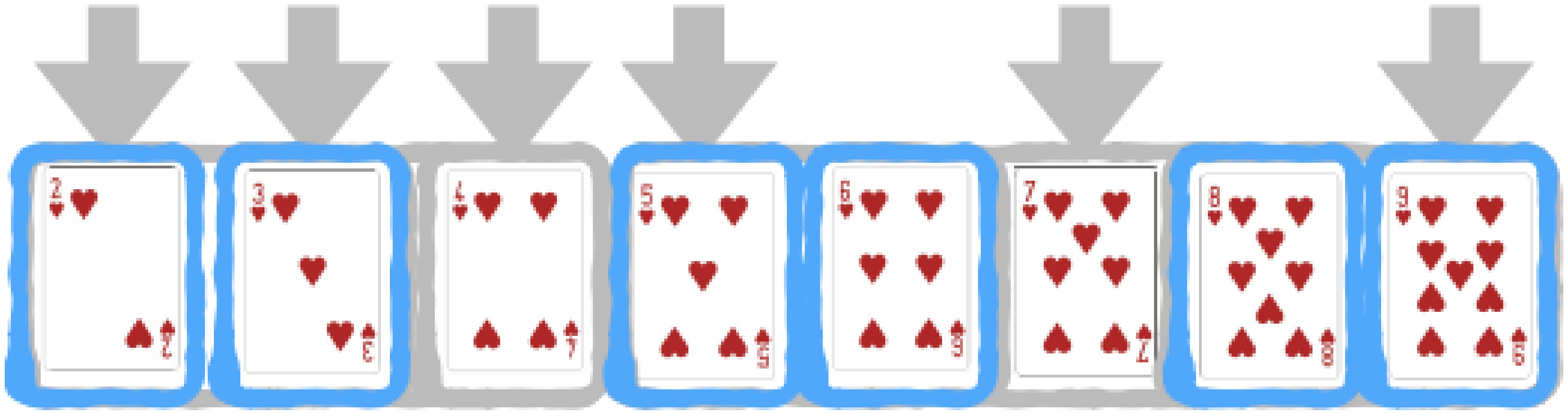
QuickSort

- Este é o resultado do novo processo de partição



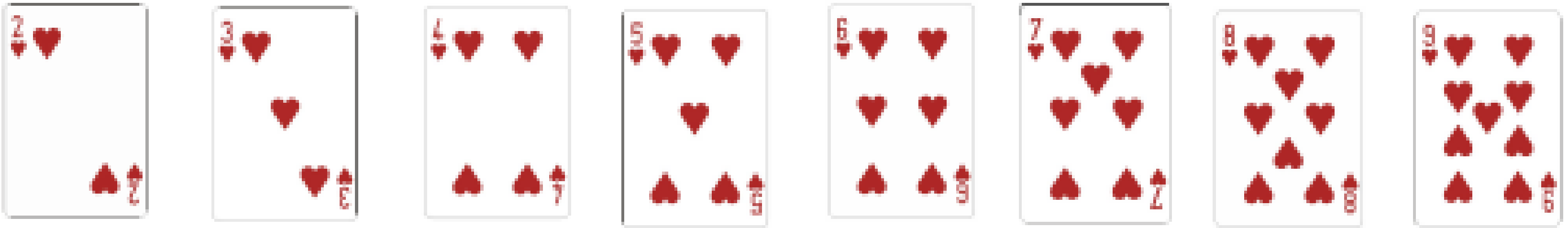
QuickSort

- Todos os subarranjos de tamanho 1 podem ser considerados ordenados



QuickSort

- Com todos os subarranjos ordenados, sabemos que o arranjo original está ordenado

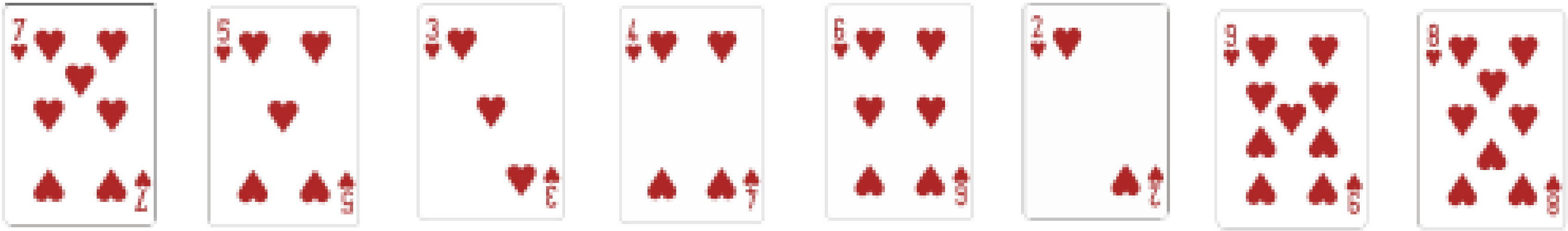


QuickSort

- Para fazer o particionamento:
  - Escolhemos um pivô x
  - Percorremos o arranjo a partir da esquerda até que a[i] >= x
  - Percorremos o arranjo a partir da direita até que a[j] <= x
  - Trocamos a[i] com a[j]
  - Continuamos até que i e j se cruzem

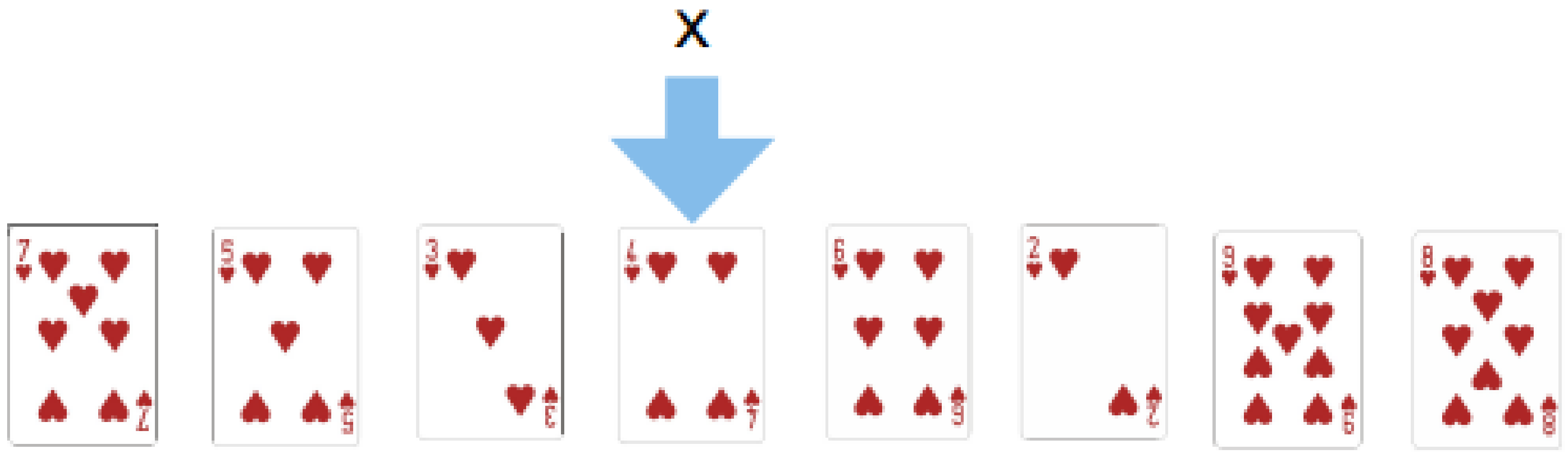
QuickSort

- Considere um arranjo com os seguintes elementos



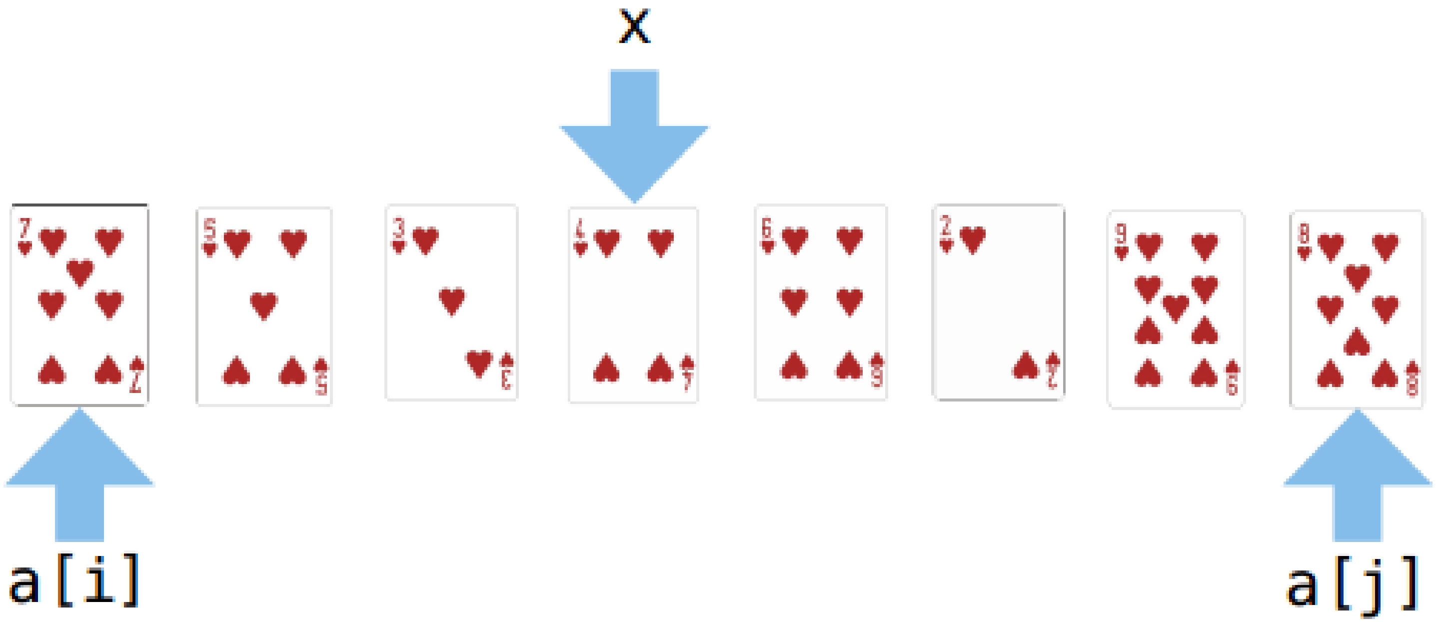
QuickSort

- Escolhemos um arranjo como pivô



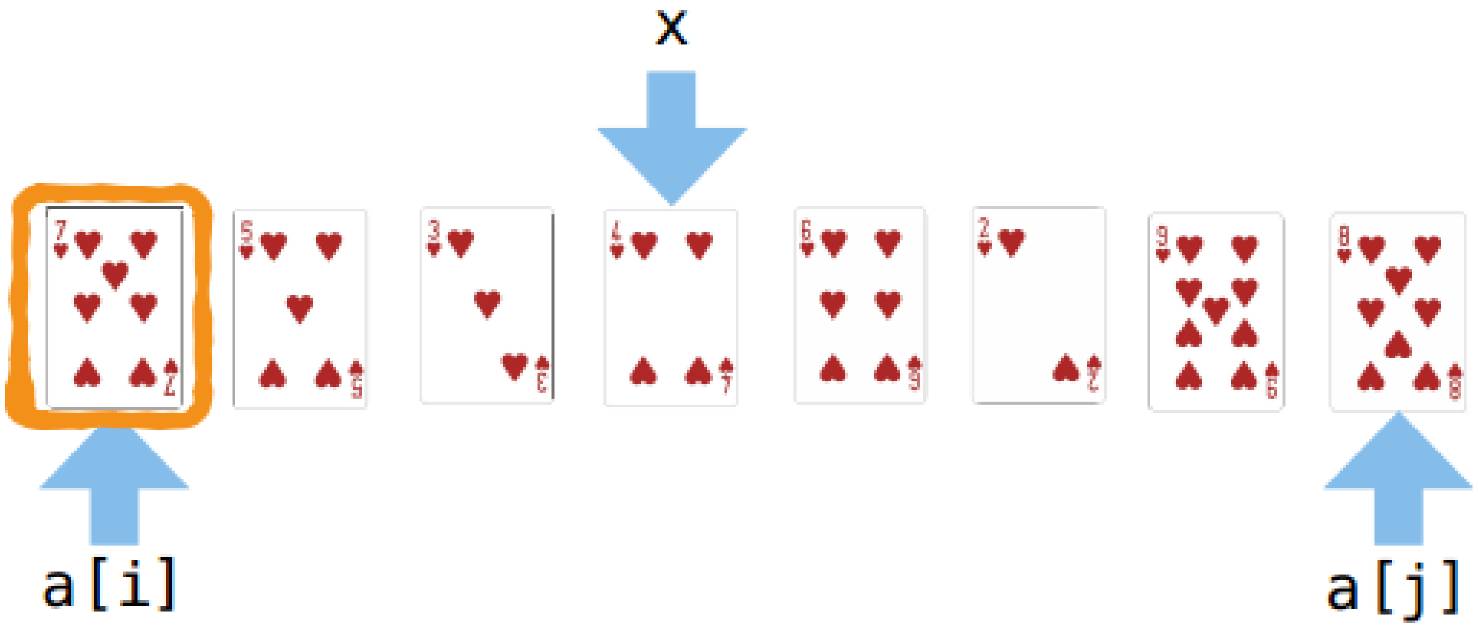
QuickSort

- Definimos índices i e j que marcam o início e o fim do arranjo



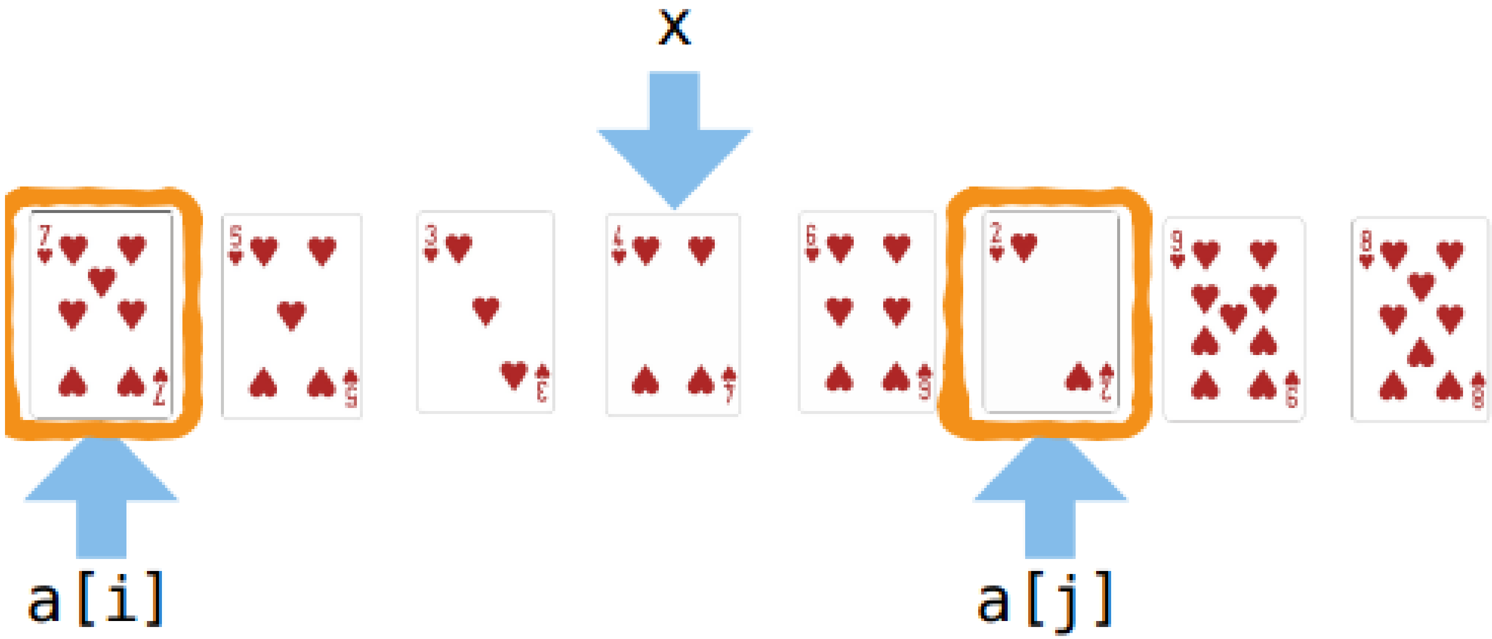
QuickSort

- Movimentamos i para a direita até encontrar um elemento maior ou igual a x. Neste caso, o elemento é o próprio 7



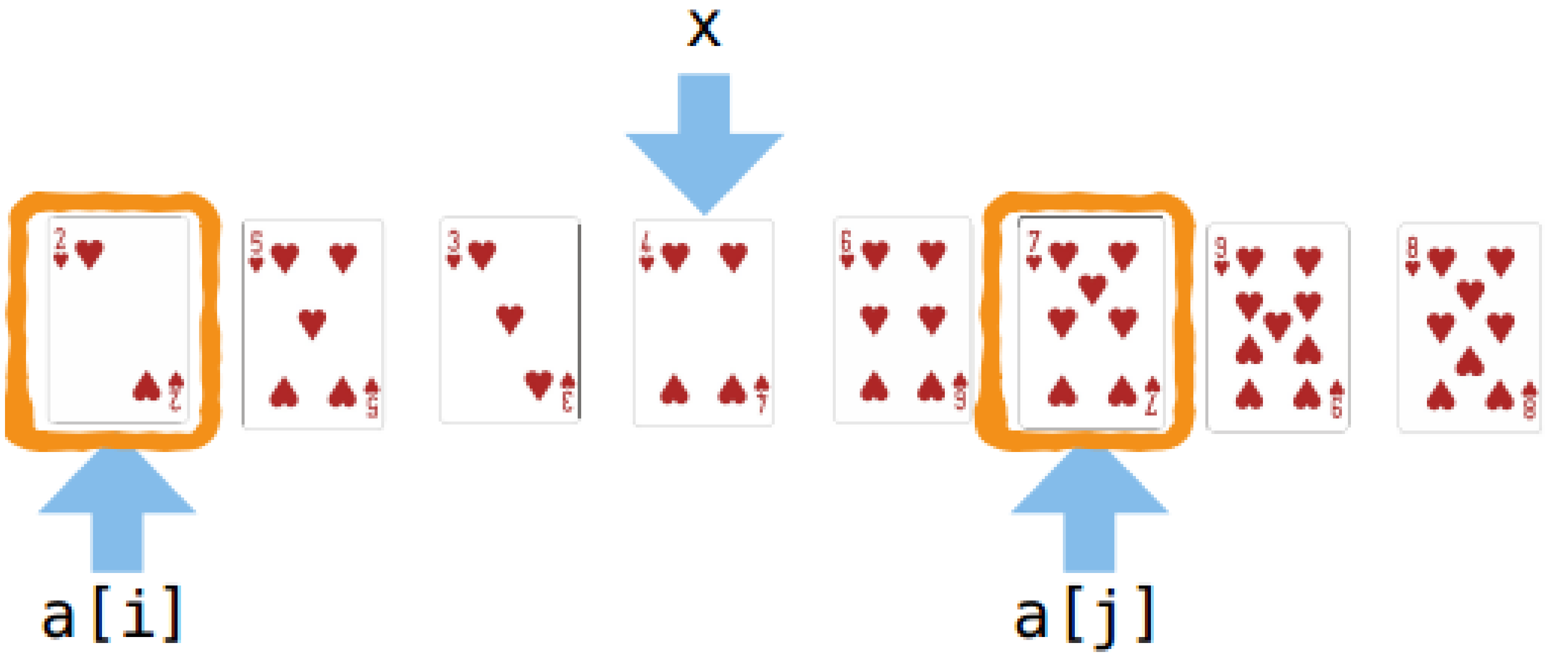
QuickSort

- Movimentamos j para a esquerda até encontrar um elemento menor ou igual a x. O primeiro elemento nesta condição é o 2



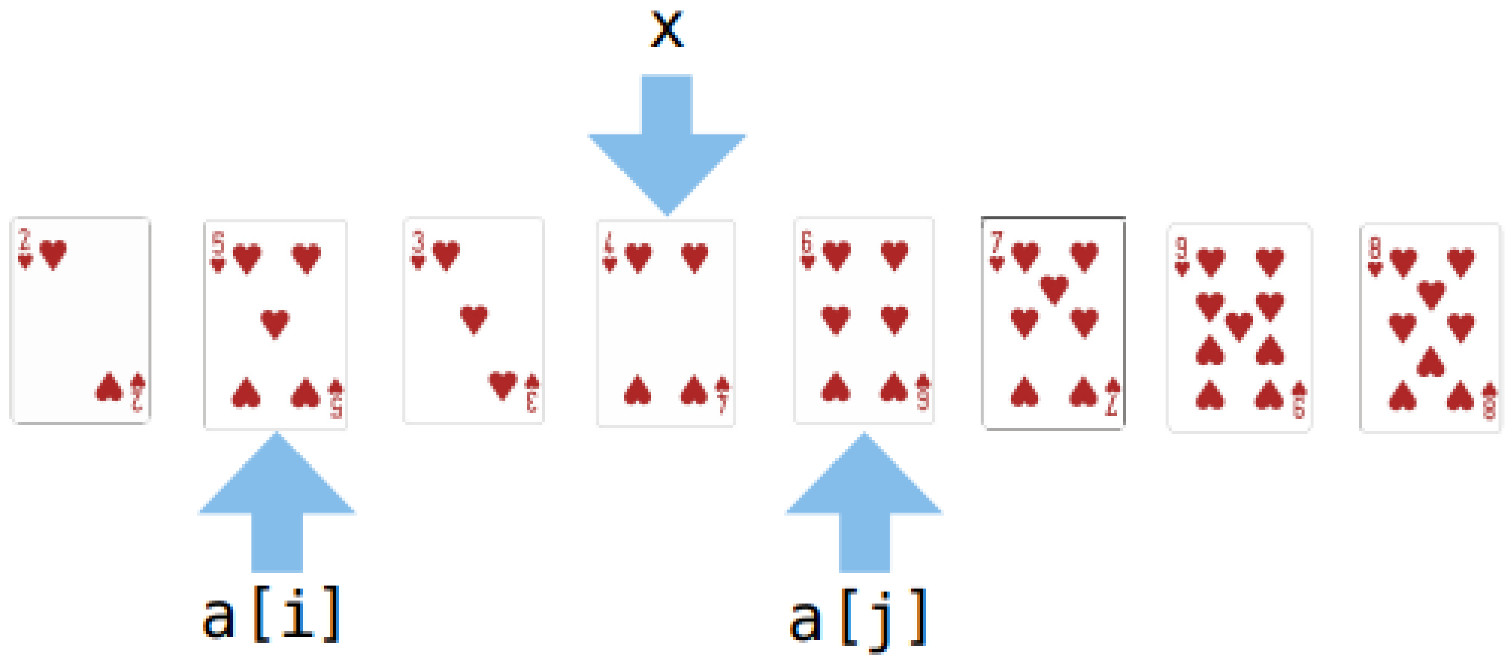
QuickSort

- Trocamos a[i] com a[j]



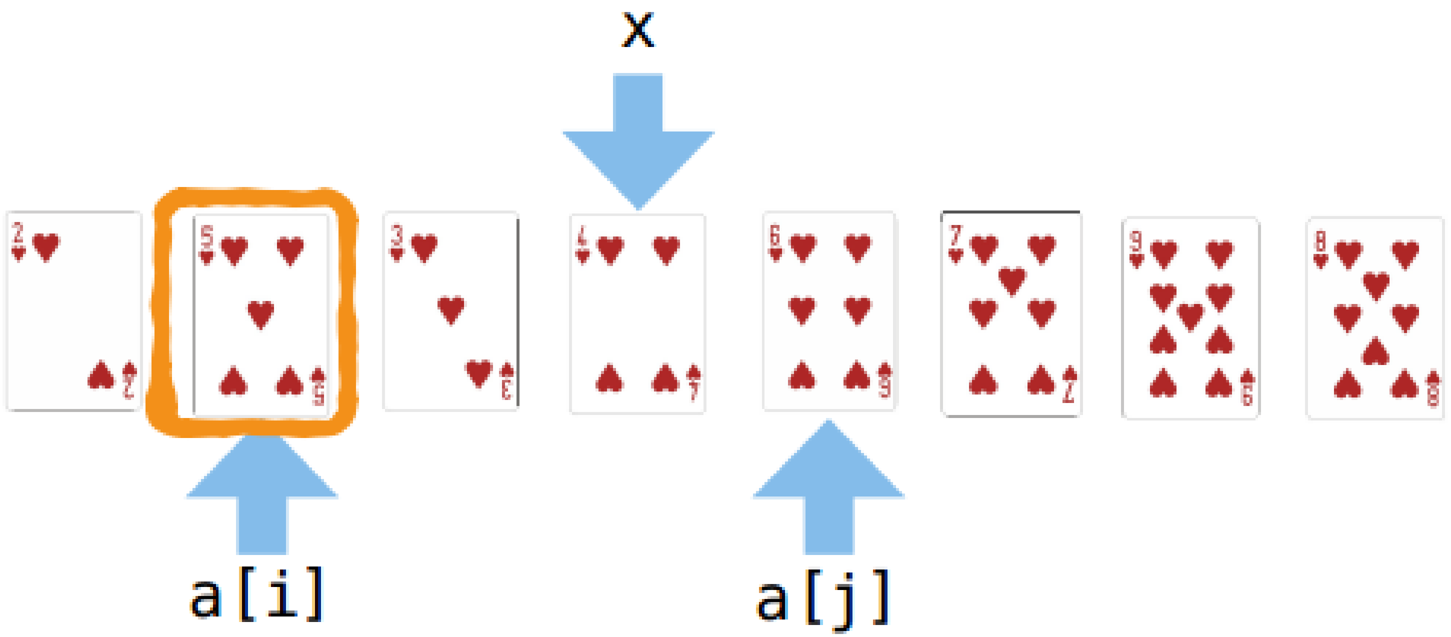
QuickSort

- Deslocamos i e j e continuamos o processo



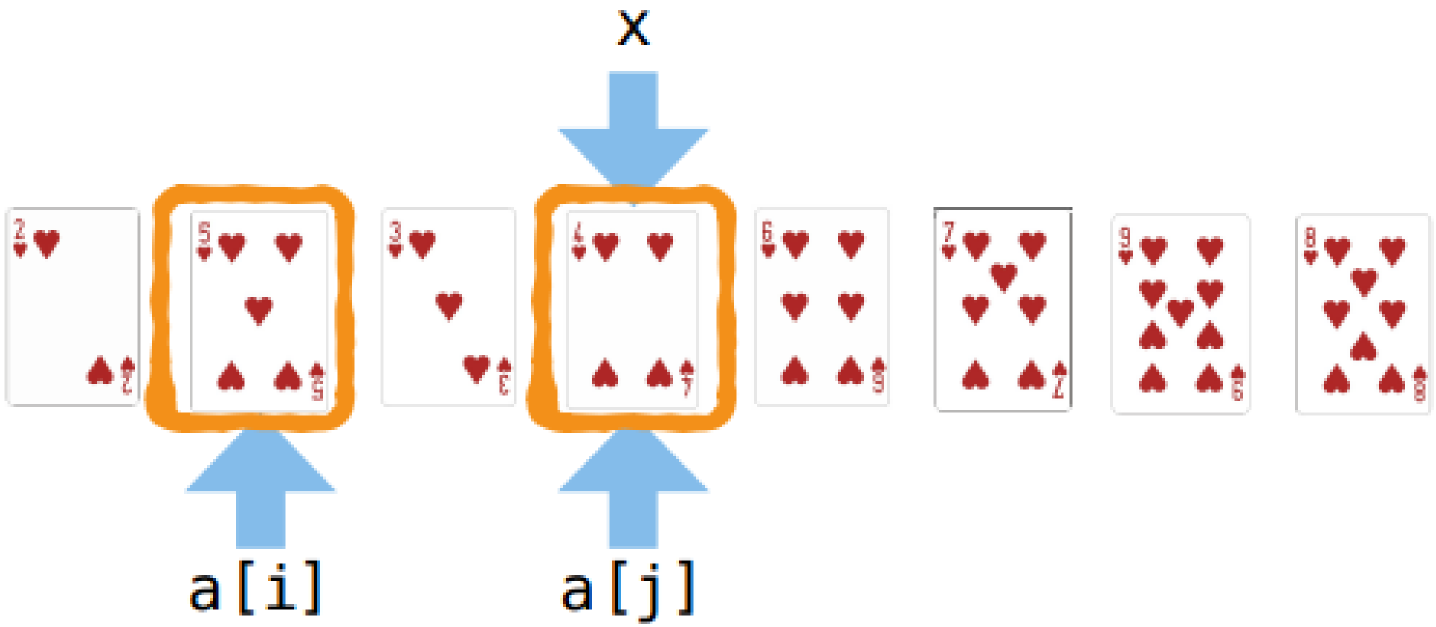
QuickSort

- Deslocamos i até encontrar um elemento a[i] maior ou igual a x. Este elemento já é o 5



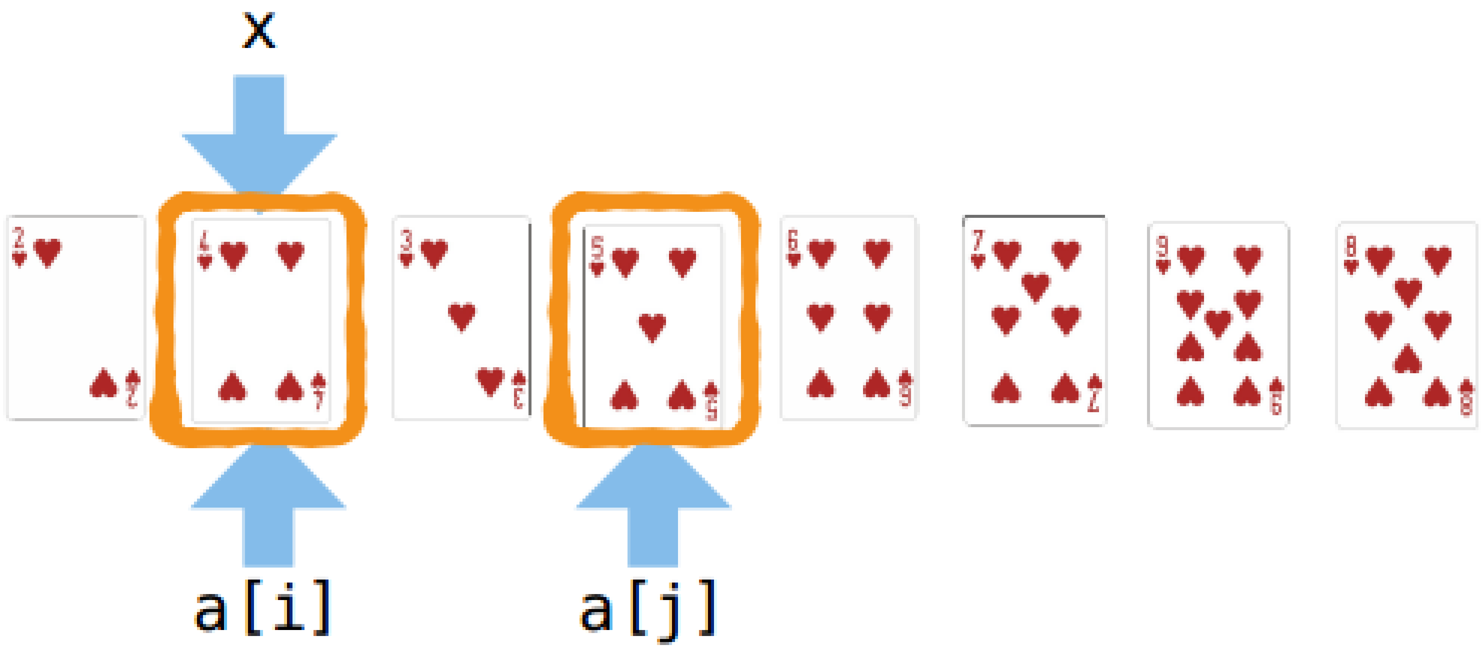
QuickSort

- Deslocamos j até encontrar um elemento a[j] menor ou igual a x. Este elemento é o 4



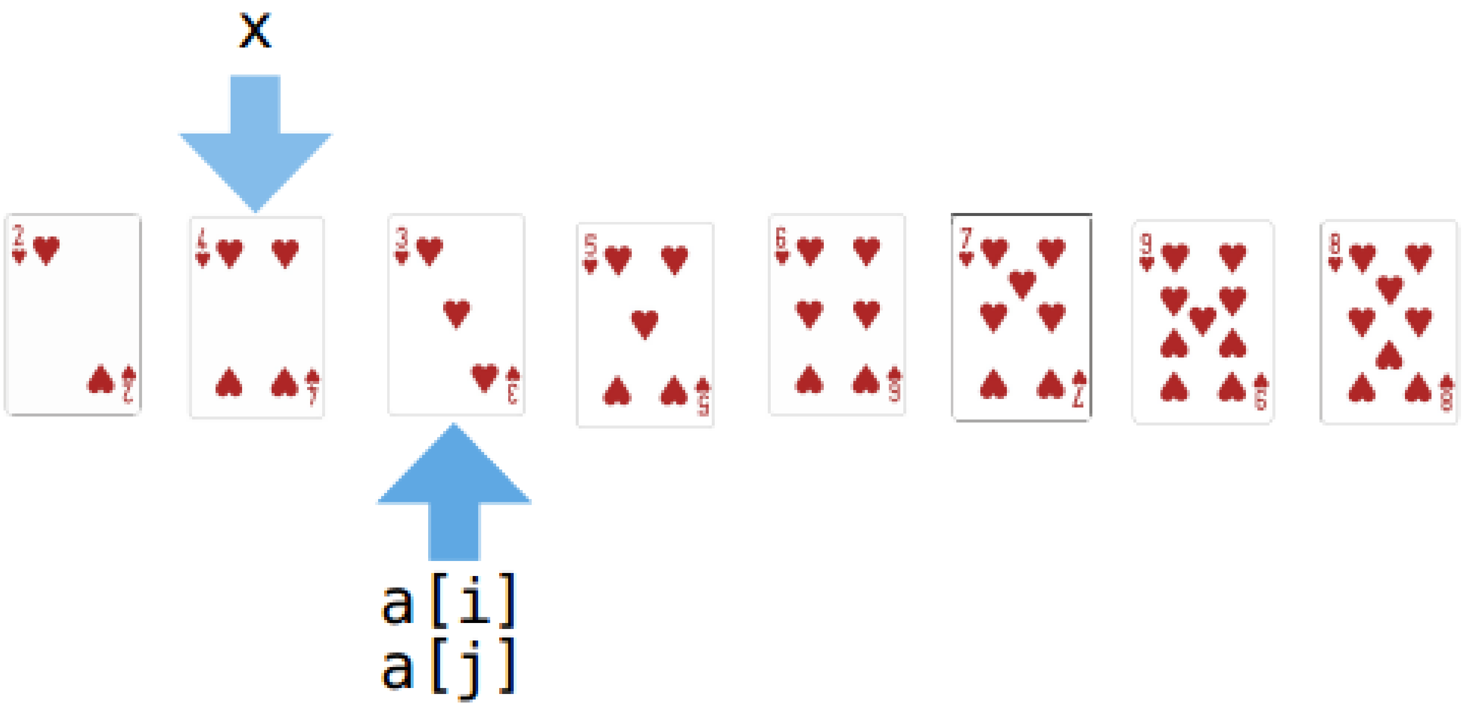
QuickSort

- Trocamos a[i] com a[j]



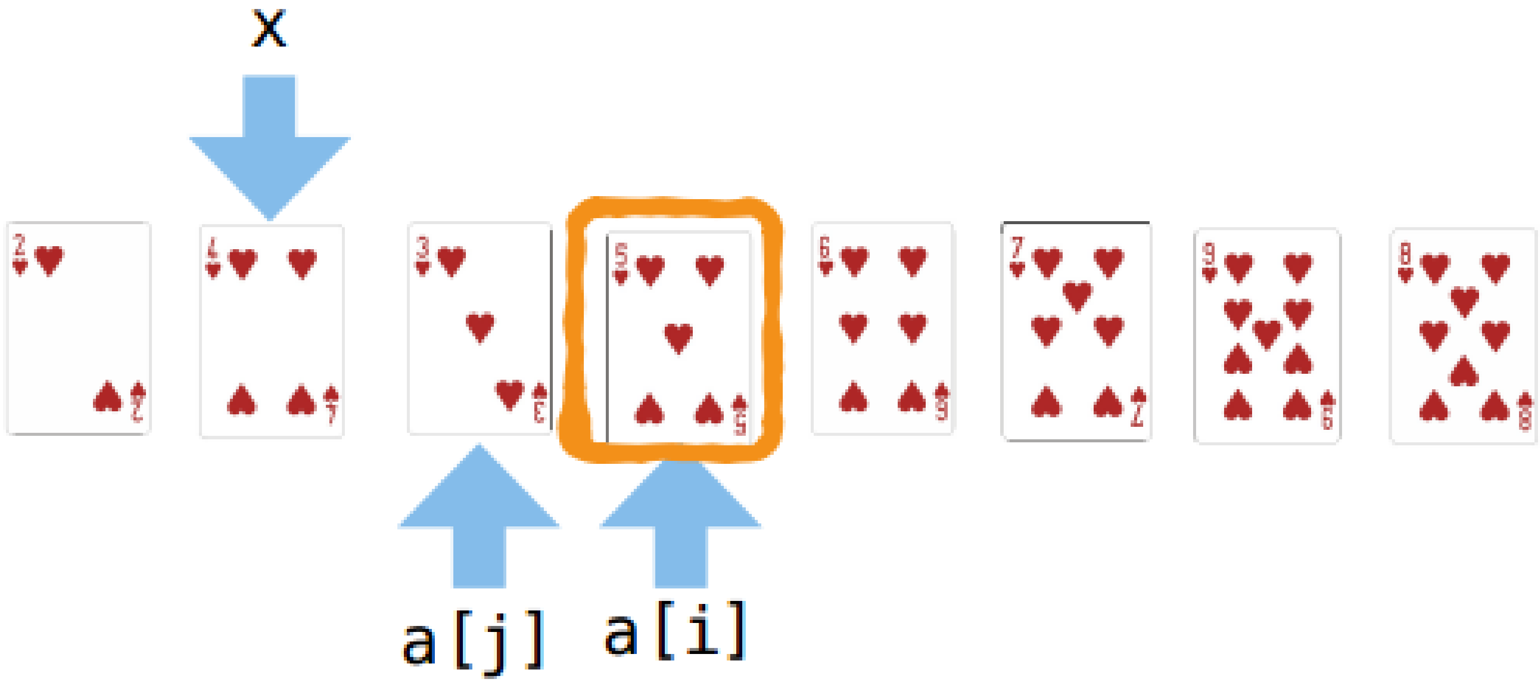
QuickSort

- Deslocamos i e j e continuaremos o processo



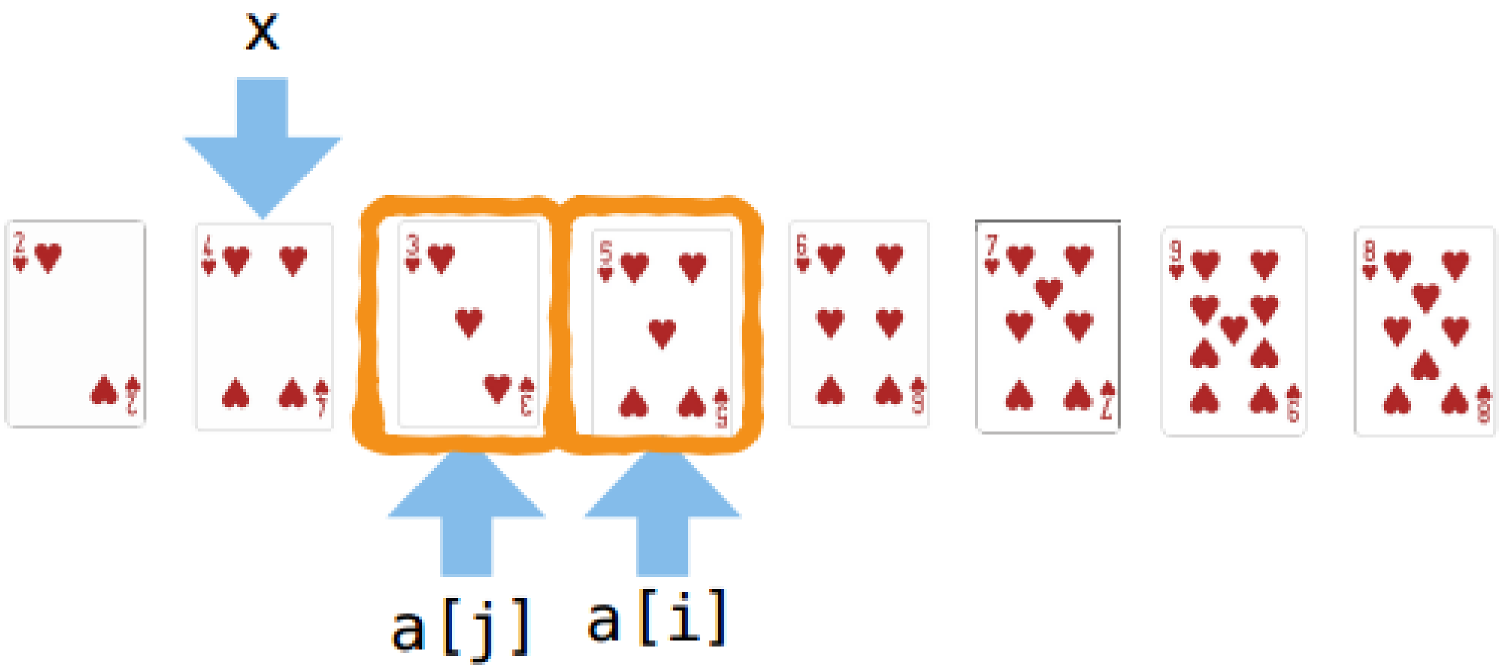
QuickSort

- Deslocamos i até encontrar um elemento a[i] maior ou igual a x



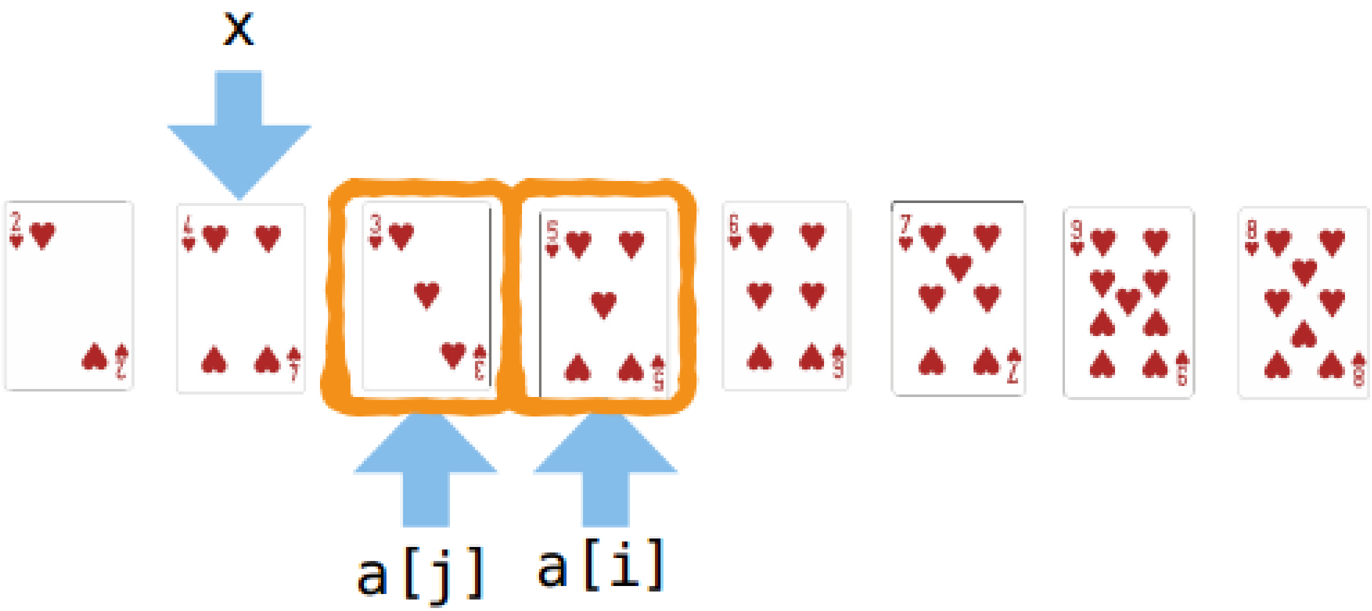
QuickSort

- Deslocamos j até encontrar um elemento a[j] menor ou igual a x. Este elemento já é o 3



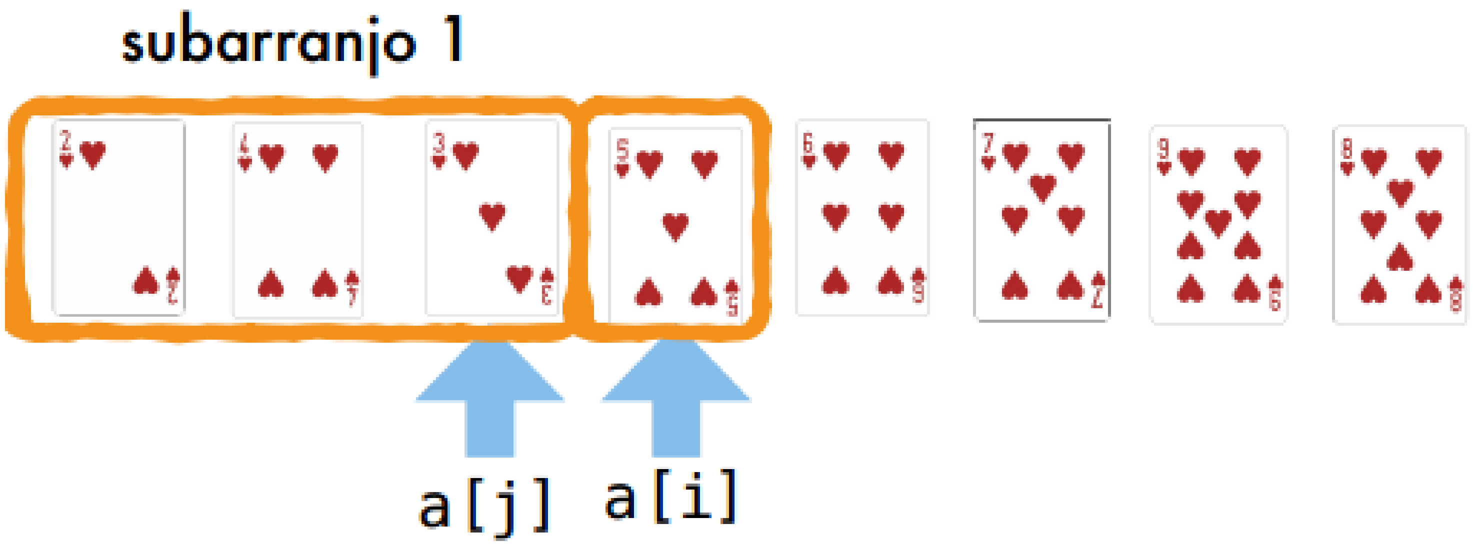
QuickSort

- Desta vez, porém, os valores de i e j se cruzaram e por isto não faremos a troca e encerramos a função



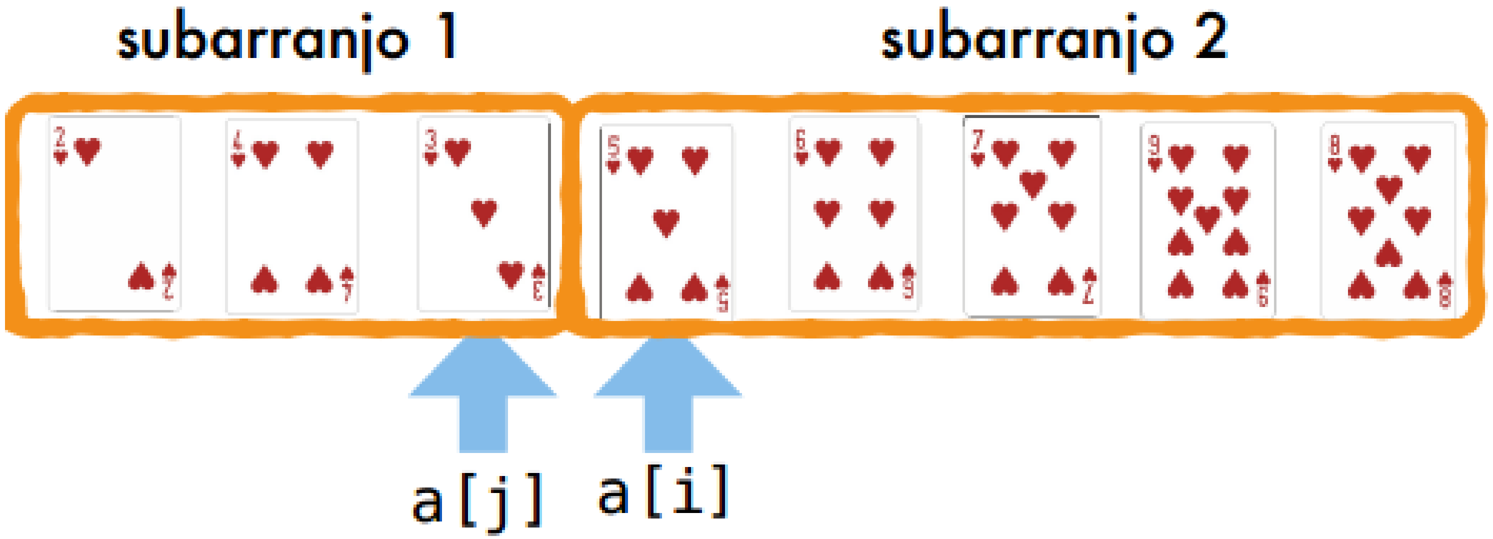
QuickSort

- Os itens da esquerda até o elemento a[j] formam um subarranjo com elementos menores ou iguais ao pivô



QuickSort

- Os itens da direita a partir do elemento a[i] formam um subarranjo com elementos maiores ou iguais ao pivô



QuickSort

- Exemplo de partição:

Passo	7	5	3	4	6	2	9	8
1	2	5	3	4	6	7	9	8
2	2	4	3	5	6	7	9	8
3	2	4	3	5	6	7	9	8

Movimentação	Ordenado
Desordenado	Pivô

QuickSort - Partição

- Nos exemplos anteriores, o pivô foi escolhido como o elemento na posição (i+j)/2

QuickSort - Partição

- Exemplo de aplicação do QuickSort

Passo	7	5	3	4	6	2	9	8
1	2	4	3	5	6	7	9	8
2	2	3	4	5	6	7	9	8
3	2	3	4	5	6	7	8	9

Subarranjo

Ordenado

Desordenado

Pivô

QuickSort - Partição

- Este é o algoritmo que particiona o arranjo a[esq]...a[dir] nos subarranjos a[esq]...a[j] e a[i]...a[dir]

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

QuickSort - Partição

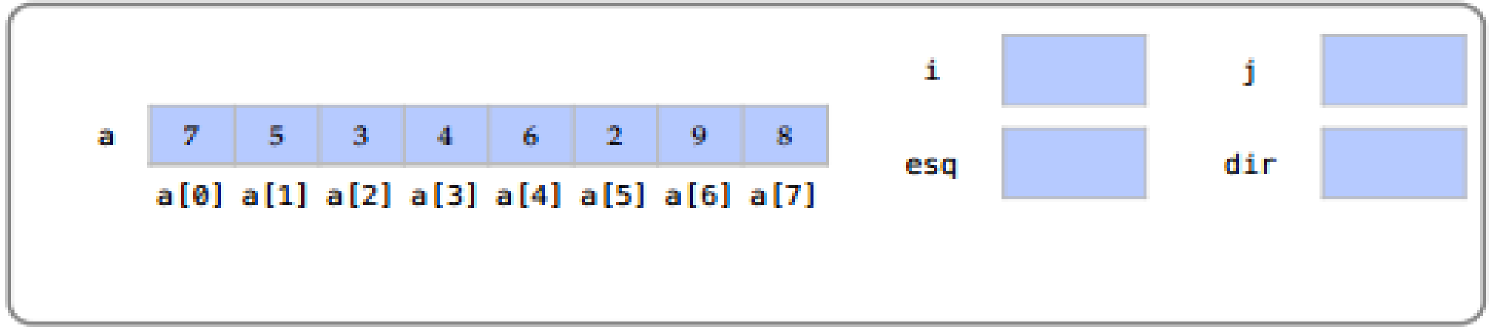
- O laço interno do algoritmo de partição é muito simples. Por isto, o algoritmo QuickSort é tão rápido

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

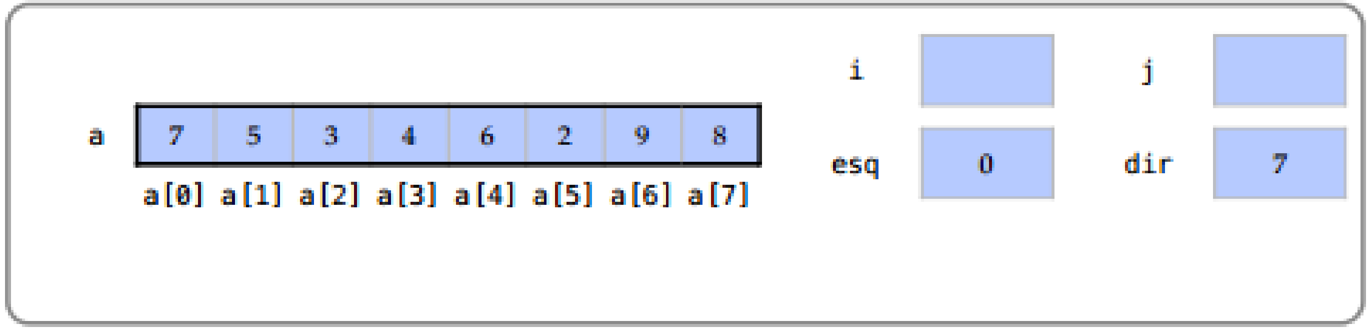
A função de partição recebe vários parâmetros.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

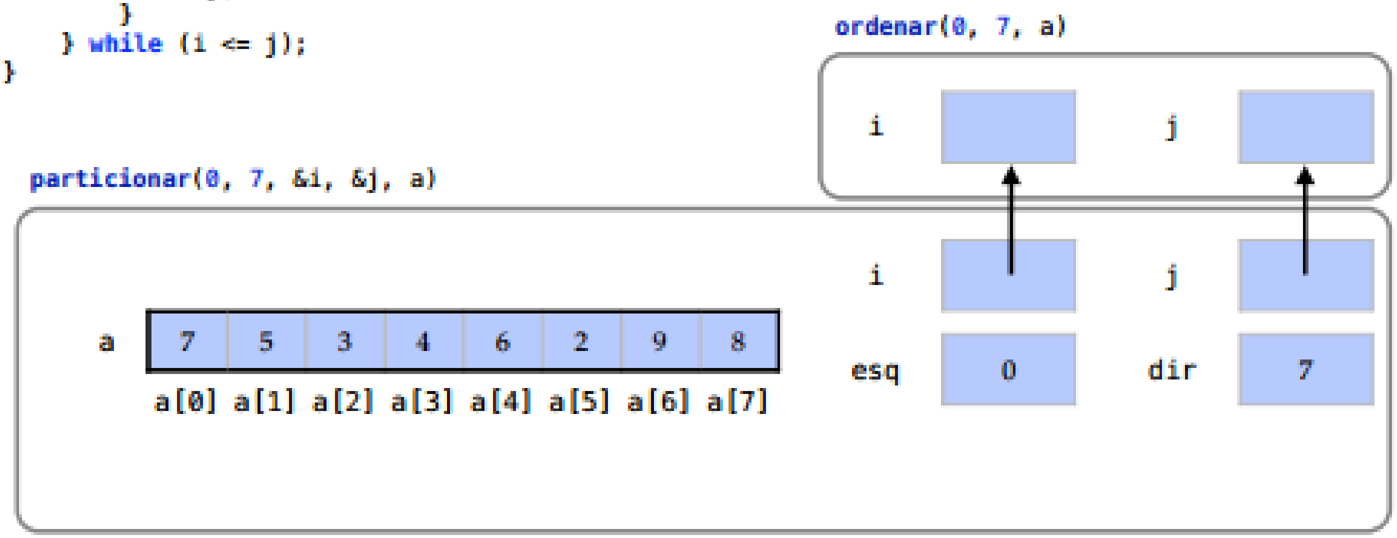
Os parâmetros `esq` e `dir` indicam qual subarranjo de `a[]` queremos particionar. Neste exemplo, particionaremos todo o arranjo: de `a[0]` a `a[7]`.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

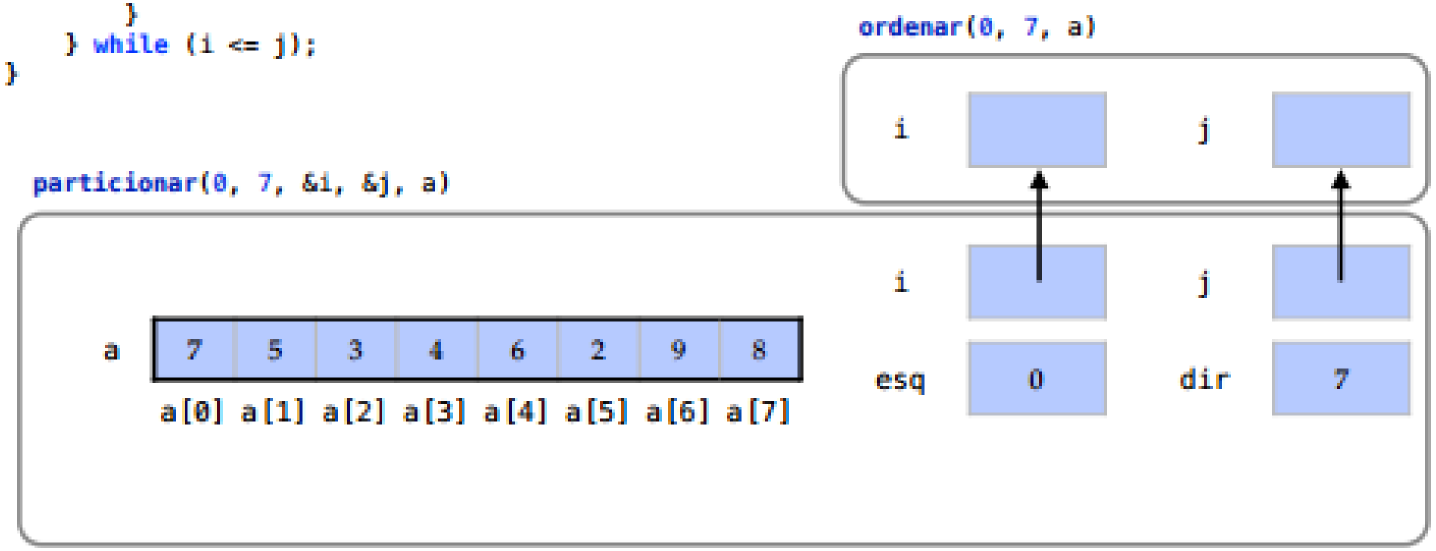
Os parâmetros i e j são passados por referência. Eles pertencem originalmente à função chamadora do quicksort.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

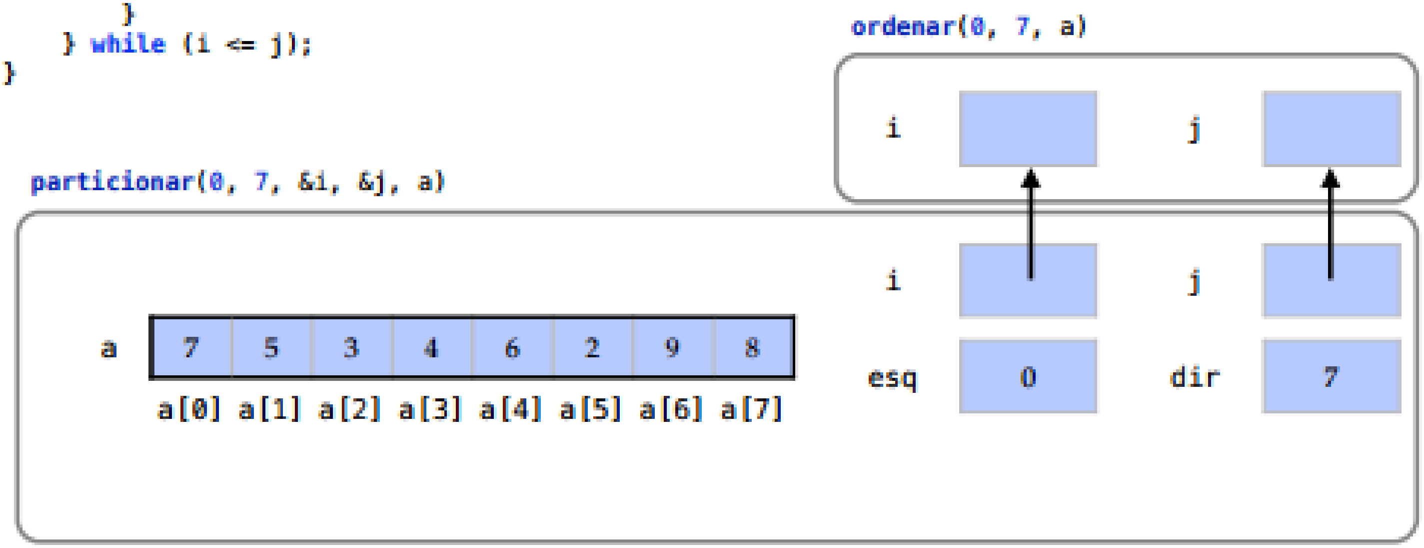
Ao fim do algoritmo, estas referências i e j dirão à função chamadora quais são os subarranjos particionados.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

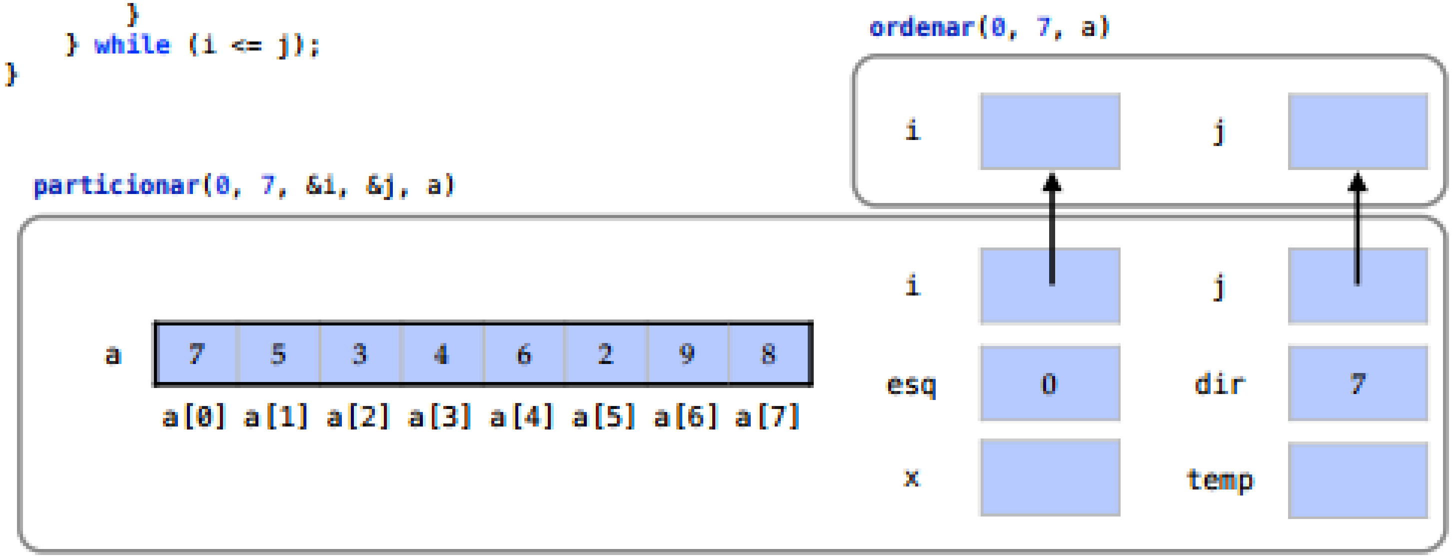
Todo o arranjo a[] é também enviado à função. Como sabemos, arranjos são endereços na memória e por isto são apenas enviados por referência.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

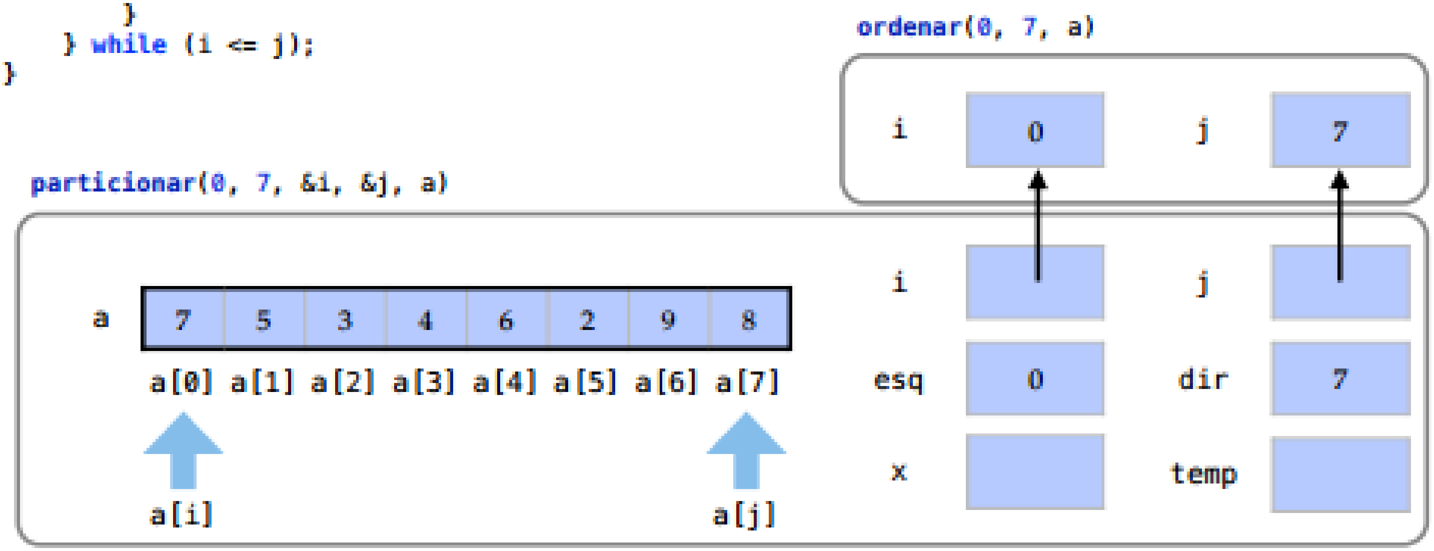
Criamos então a variável x, para guardar o elemento pivô, e uma variável temporária temp para fazer trocas.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

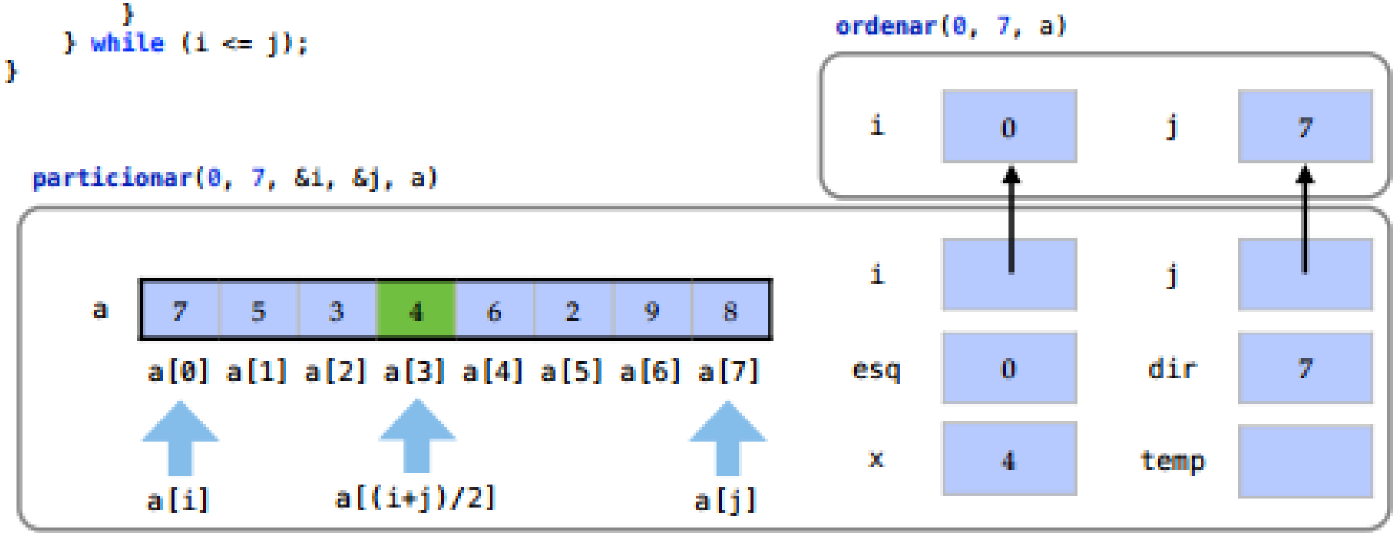
Os índices i e j são inicializados nos extremos do arranjo a ser particionado.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

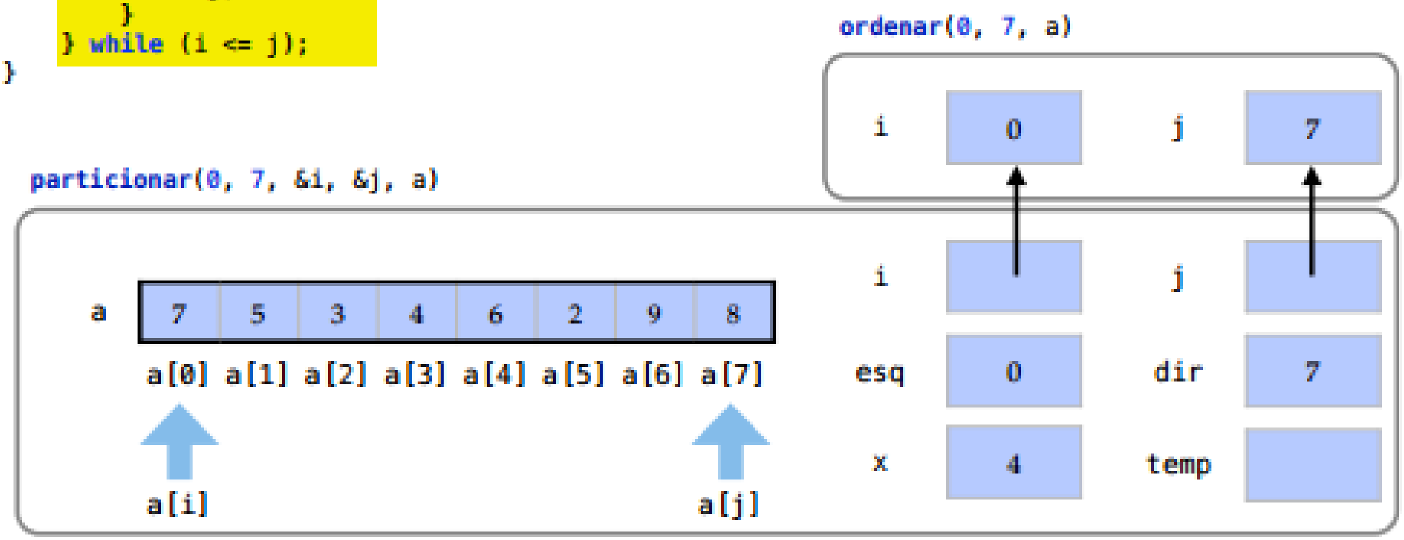
O elemento do meio é então escolhido como pivô. De acordo com a estratégia, outro pivô poderia ter sido escolhido.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

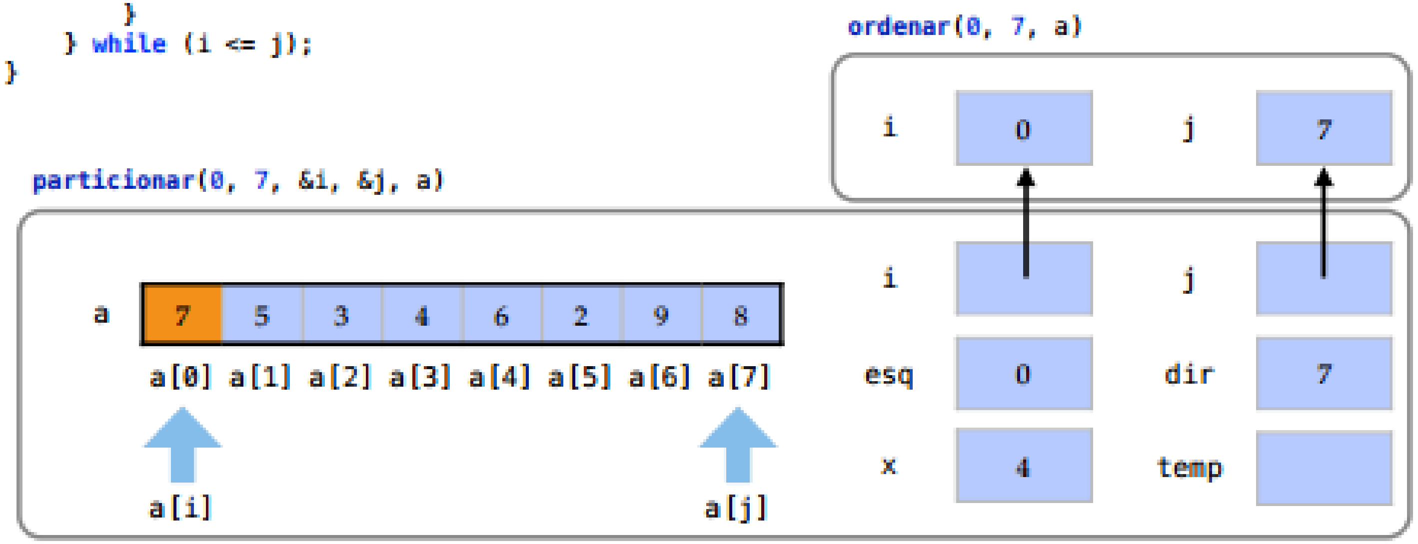
Neste grande laço, vamos fazer as trocas enquanto os índices `i` e `j` não tenham se cruzado.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

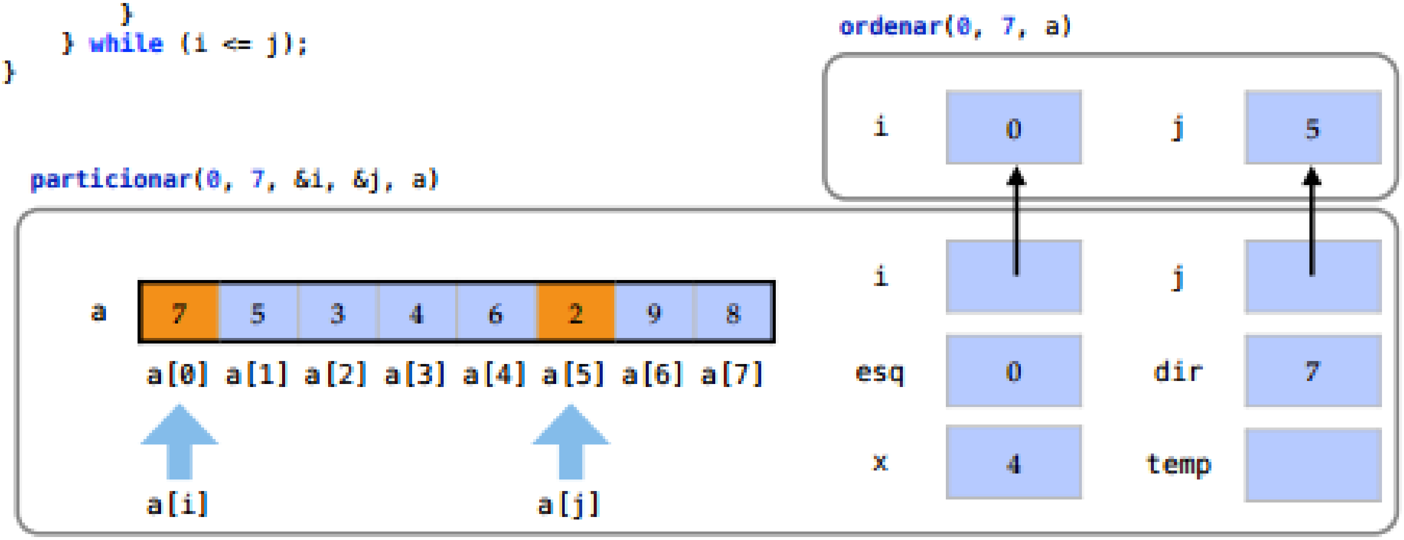
Deslocamos o índice i até encontrar o primeiro elemento a[i] maior ou igual a x.



QuickSort - Partição

```c
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

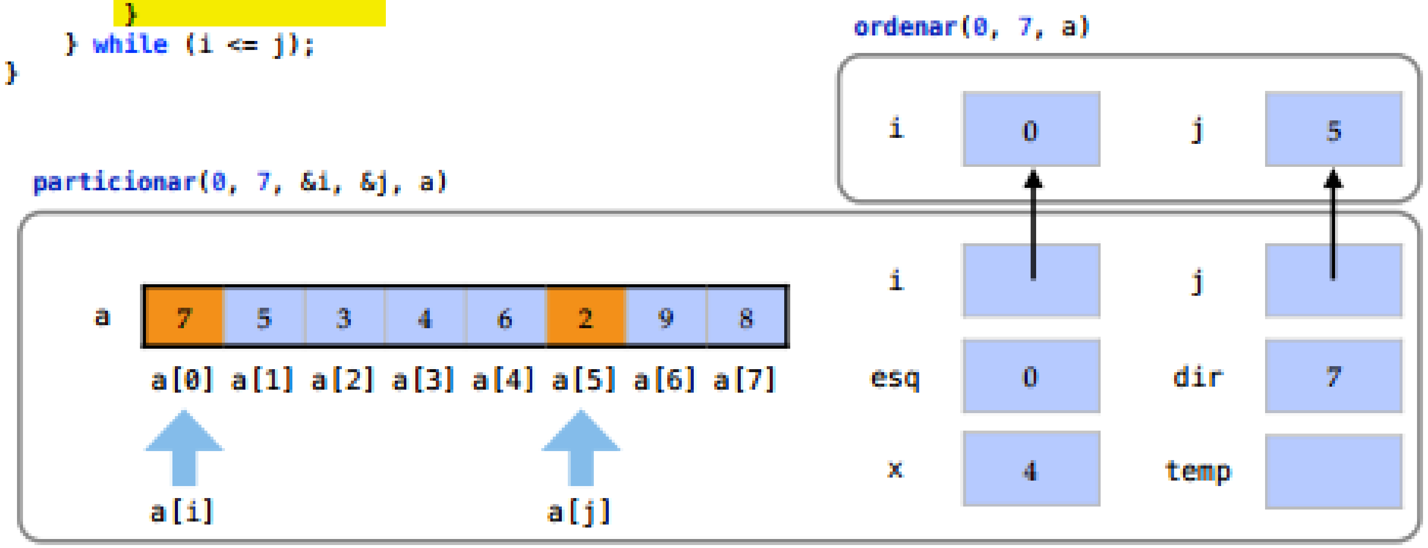
Deslocamos o índice j até encontrar o primeiro elemento a[j] menor ou igual a x.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

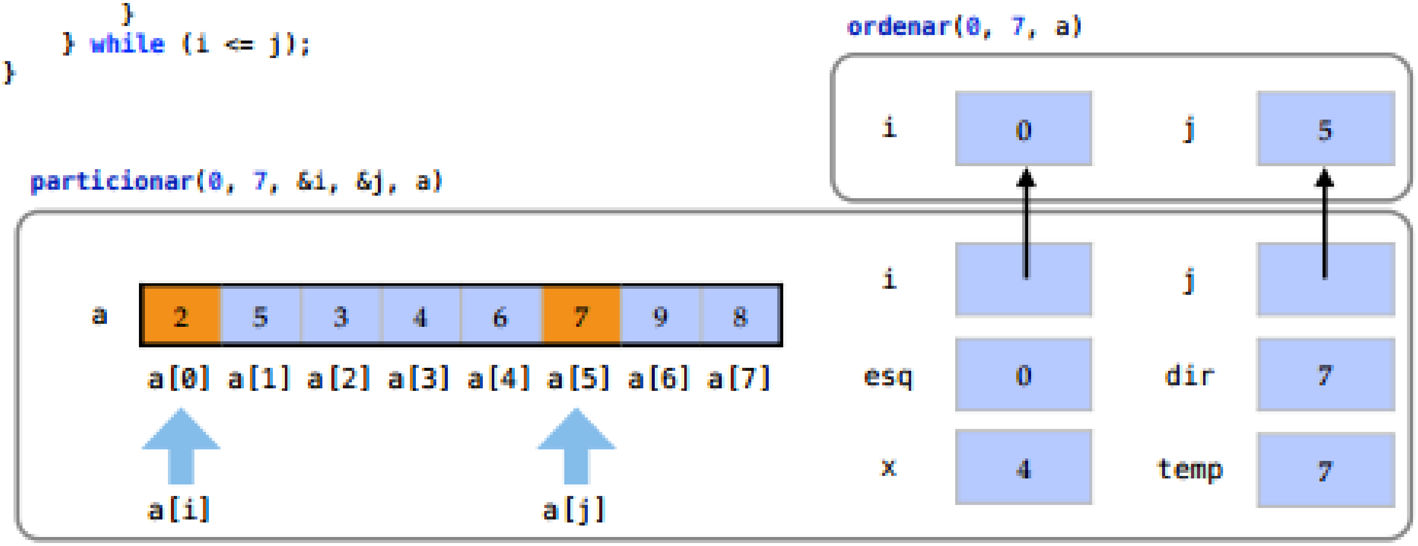
Se neste deslocamento, os índices não cruzaram...



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

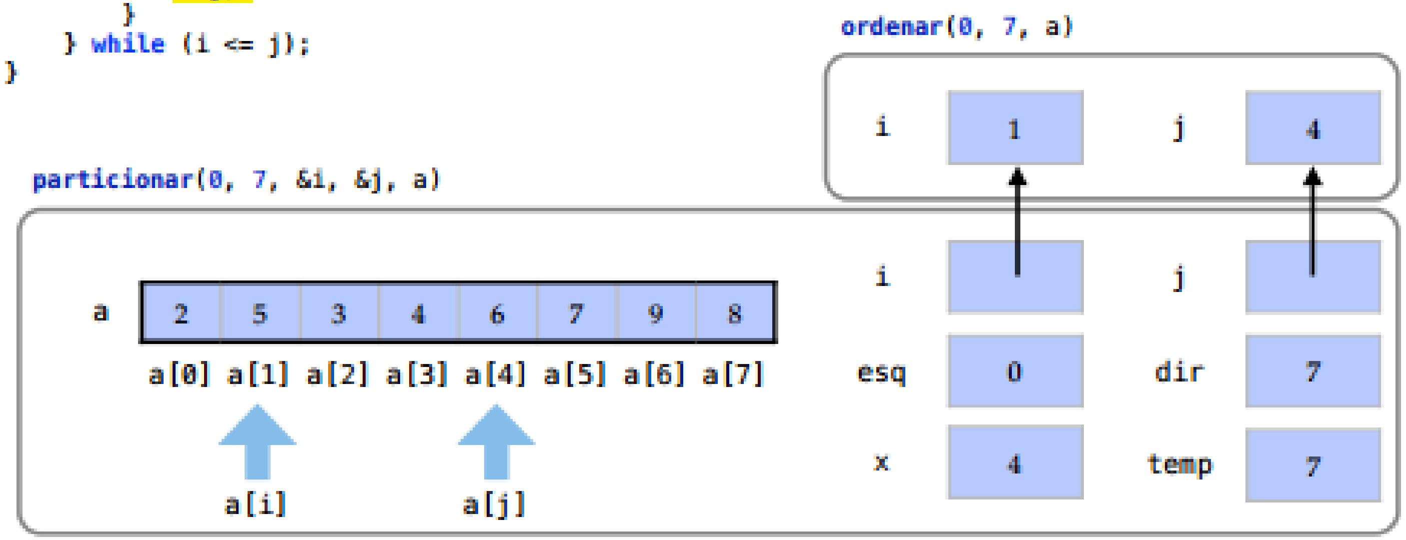
Trocaremos a[i] com a[j]



QuickSort - Partição

```c
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

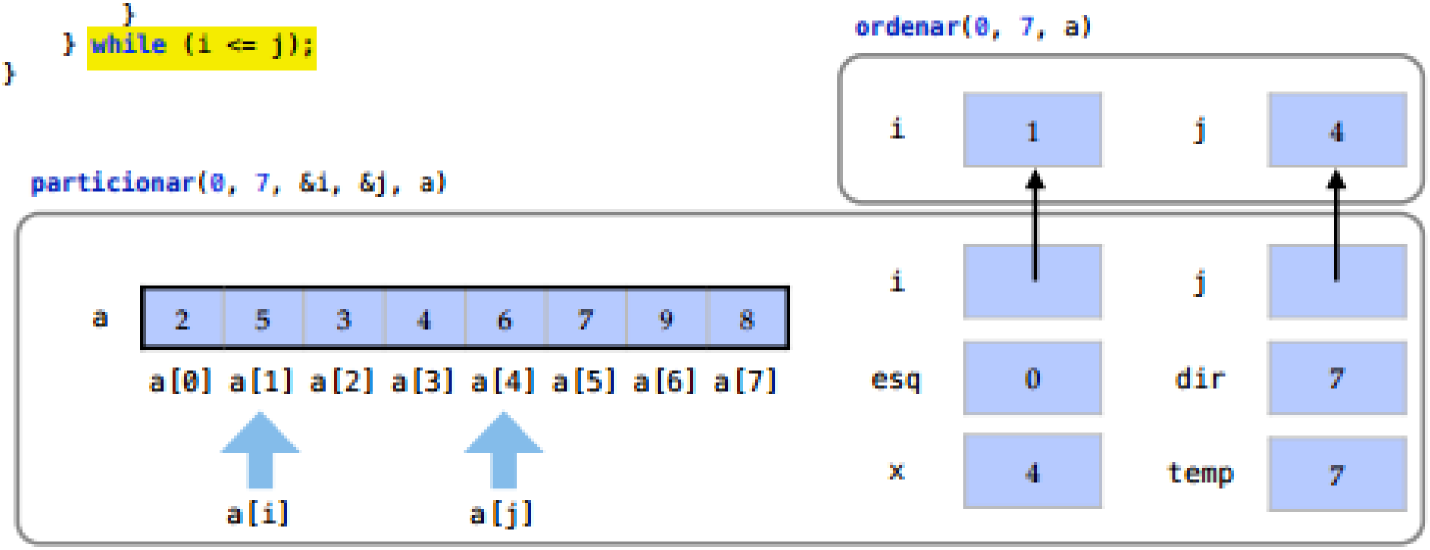
Deslocamos i e j em mais uma posição



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

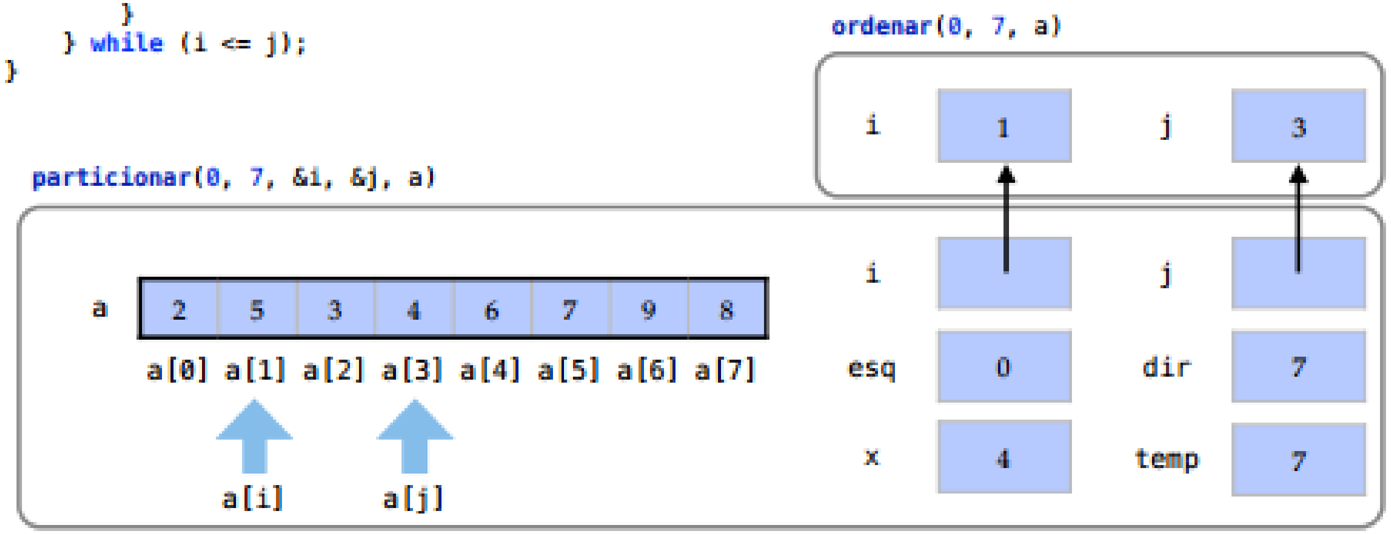
Todo o laço externo se repete enquanto os índices não se cruzarem.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

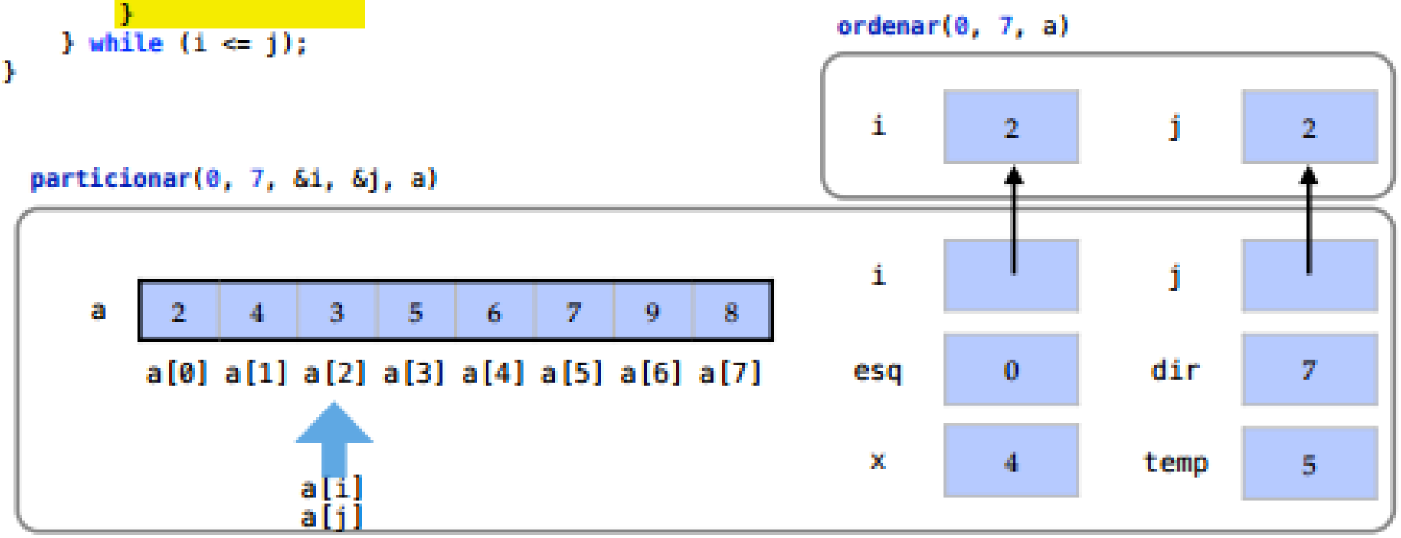
Deslocamos os índices i e j.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

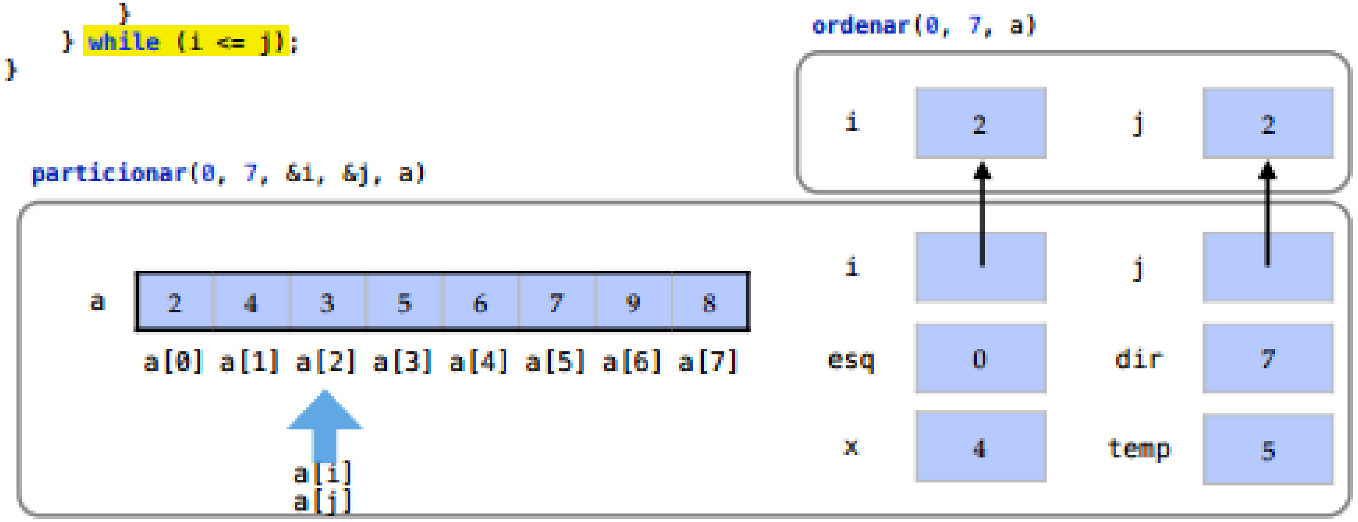
Como os índices não estão cruzados, trocamos `a[i]` com `a[j]` e deslocamos `i` e `j` em uma posição.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

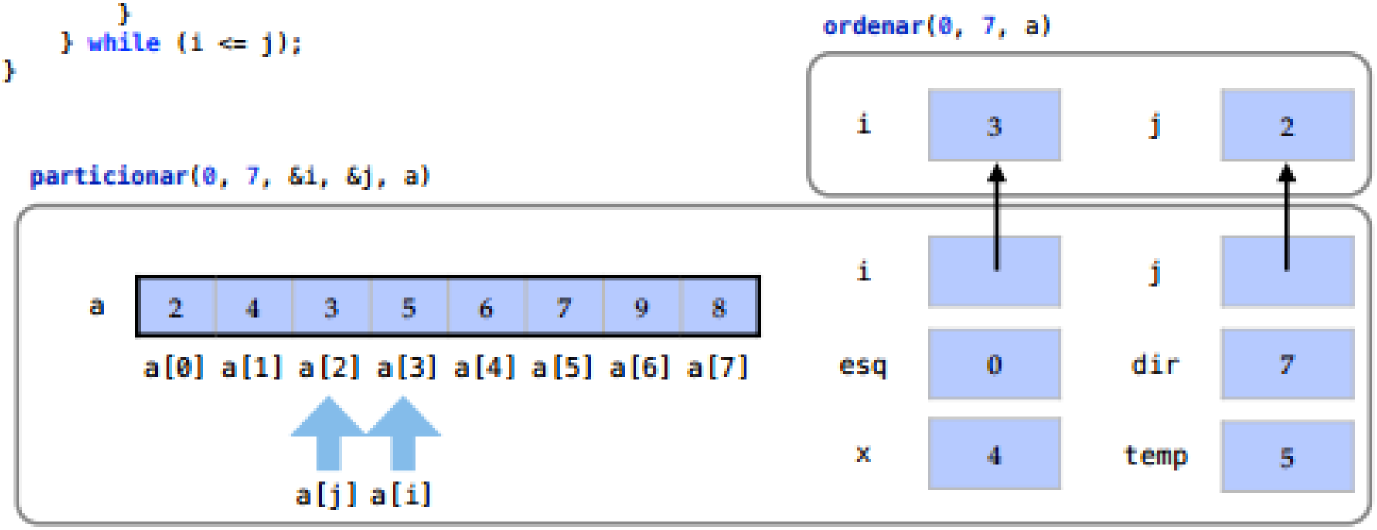
Os índices i e j estão na mesma posição mas ainda não se cruzaram.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

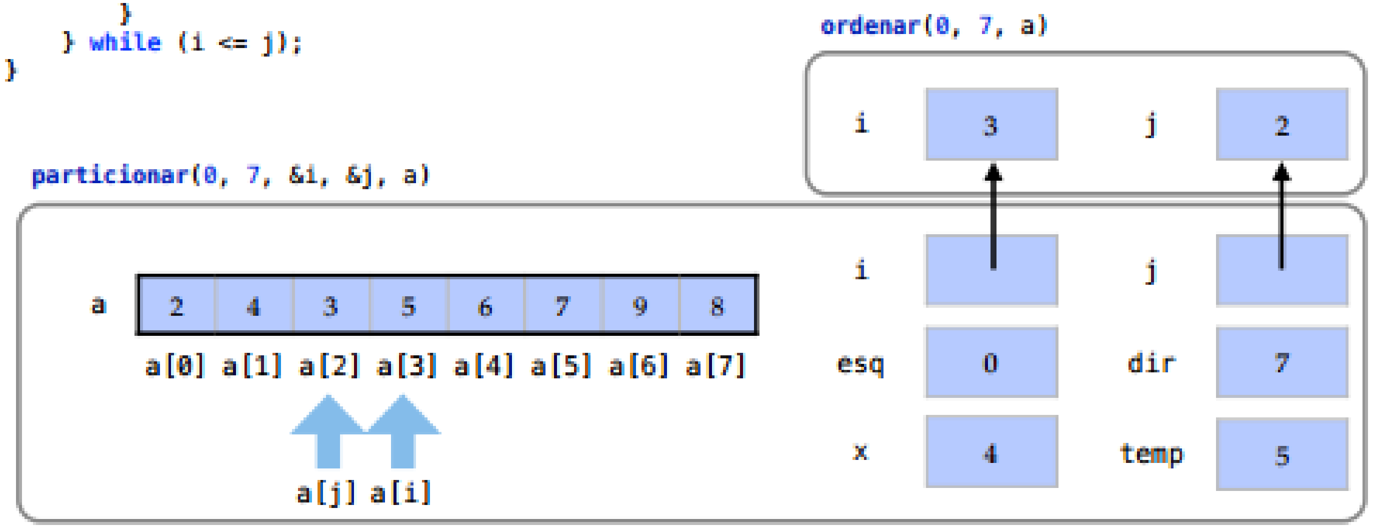
O primeiro a[i] maior ou igual a x é a[3].



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

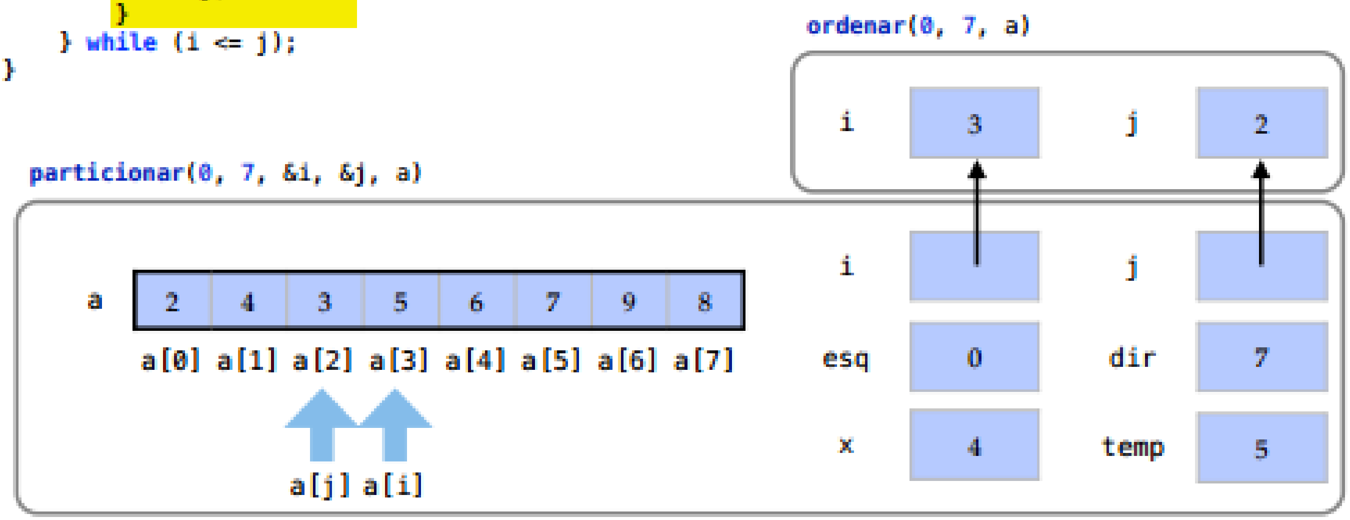
O elemento a[j] já é menor ou igual a x.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

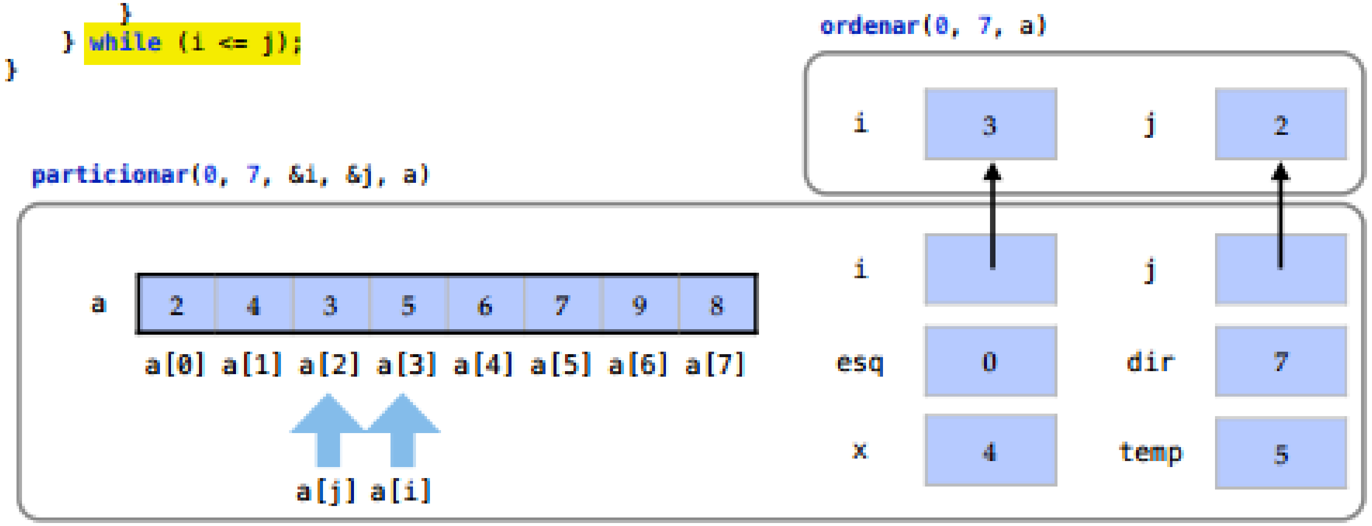
Como os índices já se cruzaram, a troca não é feita.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

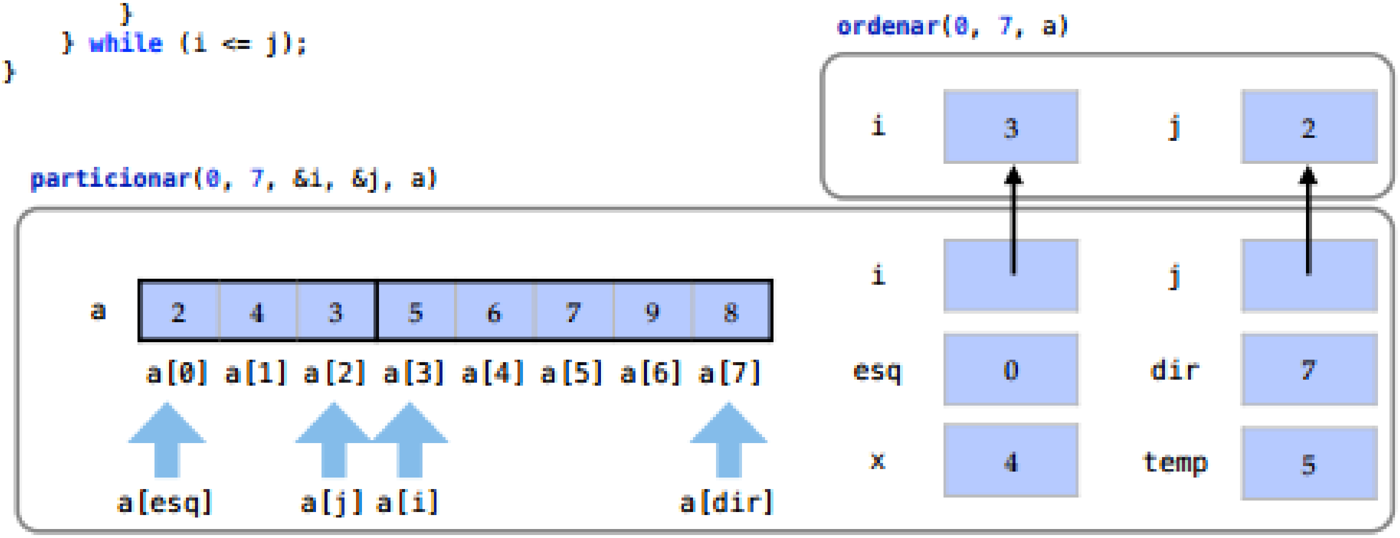
A segunda constatação de que os índices já se cruzaram encerra a função.



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

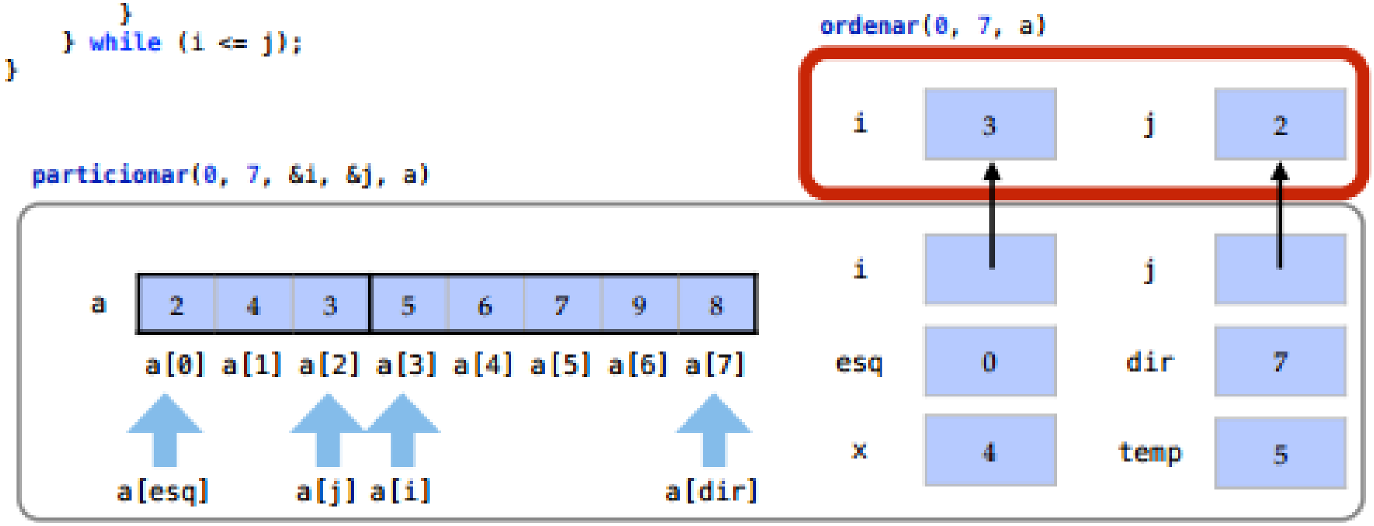
Como resultado, temos dois subarranjos. Um de a[0] até a[2] e outros de a[3] até a[7].



QuickSort - Partição

```
void particionar(int esq, int dir, int &i, int &j, int a[]) {
    int x, temp;
    i = esq;
    j = dir;
    x = a[(i + j) / 2];
    do
    {
        while (x > a[i]){
            ++i;
        }
        while (x < a[j]){
            --j;
        }
        if (i <= j){
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
}
```

Como i e j foram passados por referência, a função chamadora sabe que os subarranjos são de a[esq] até a[j] e de a[i] até a[dir].



QuickSort

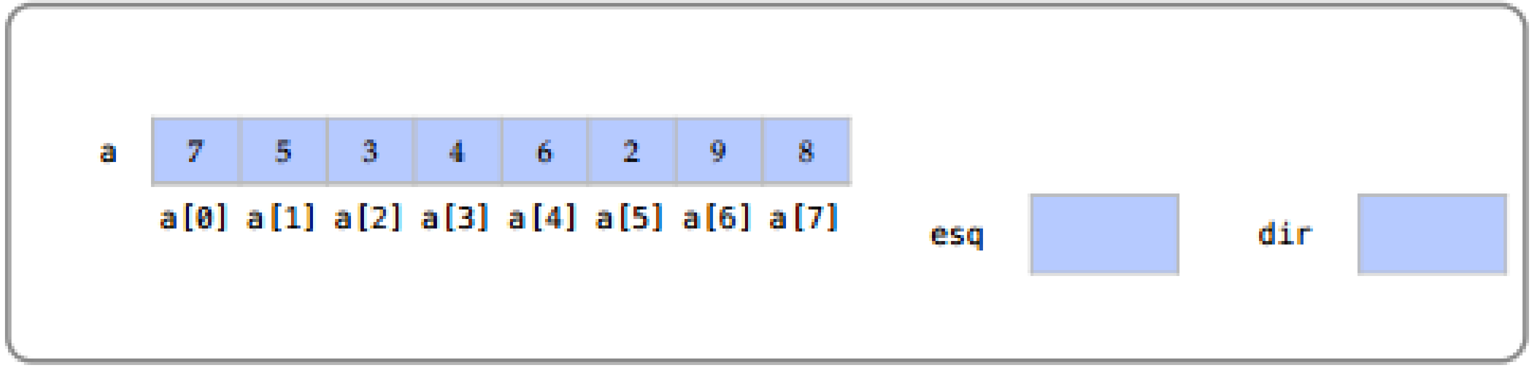
- Tendo pronta a função para o processo de partição, o algoritmo de ordenação se torna conceitualmente muito simples

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

QuickSort

- Esta função ordena o arranjo a[] entre as posições esq e dir

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Para ordenar todo arranjo, chamamos: ordenar(0, 7, a)

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

ordenar(0, 7, a)

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

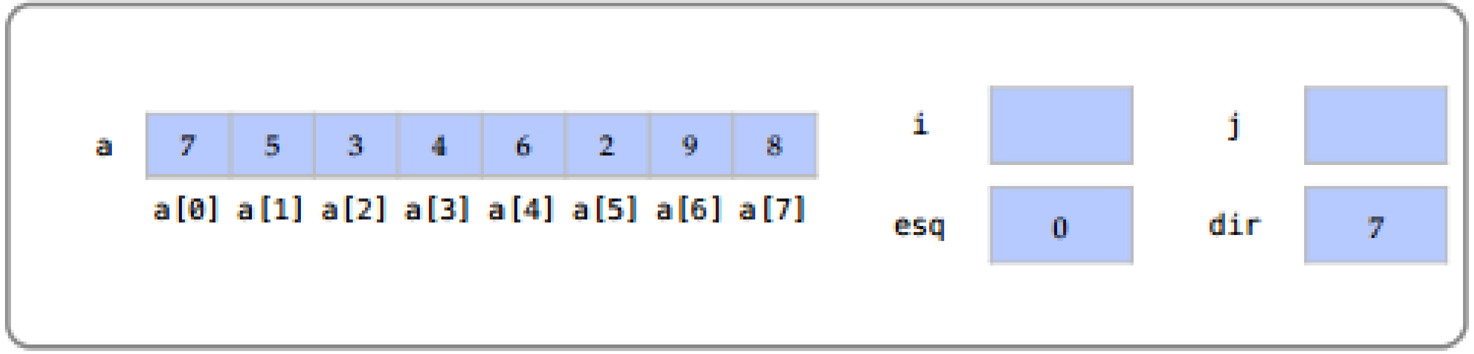
esq: 0   dir: 7

QuickSort

- Criamos então os índices i e j, que indicarão quais são os subarranjos após a partição

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

ordenar(0, 7, a)

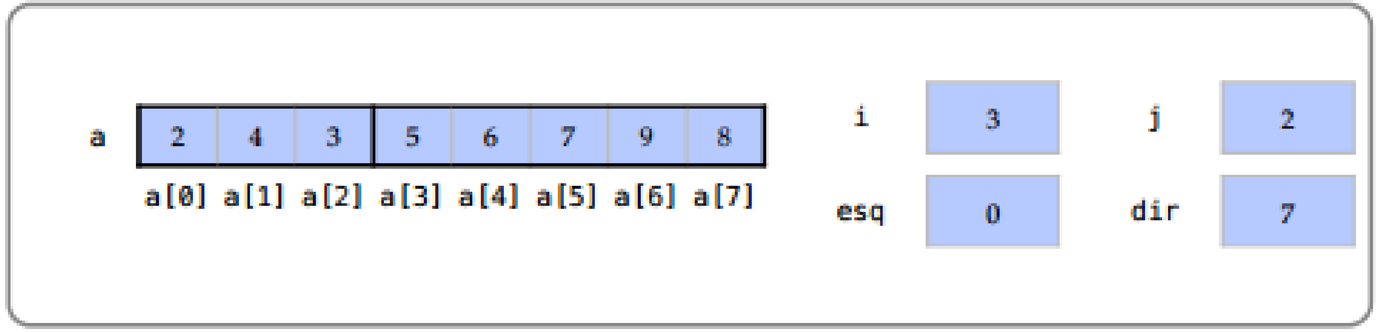


QuickSort

- Particionamos o arranjo entre as posições a[esq] e a[dir]. Os índices i e j indicam onde termina o primeiro subarranjo e onde começa o segundo

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

```
ordenar(0, 7, a)
```

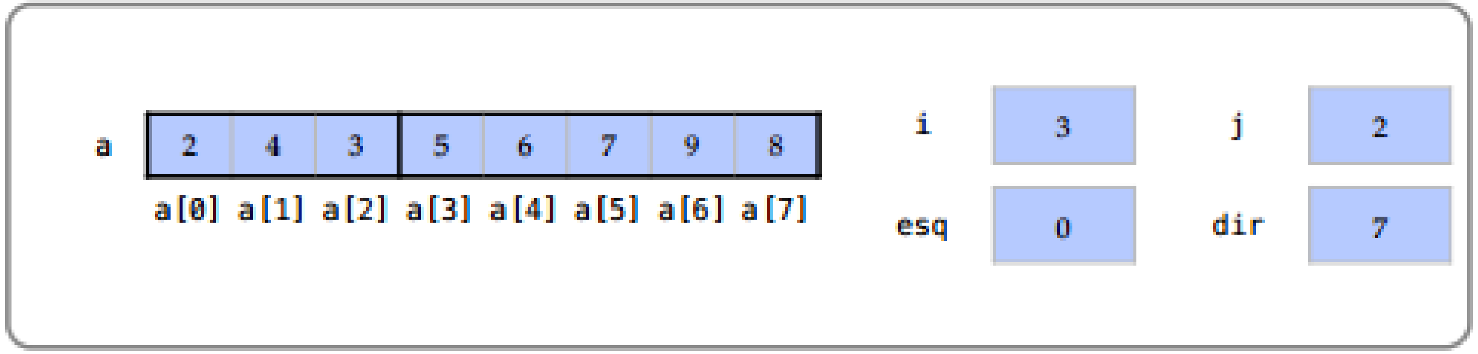


QuickSort

- Se o valor de j atingir o valor de esq, o subarranjo tem apenas um elemento e não precisa ser ordenado

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

ordenar(0, 7, a)

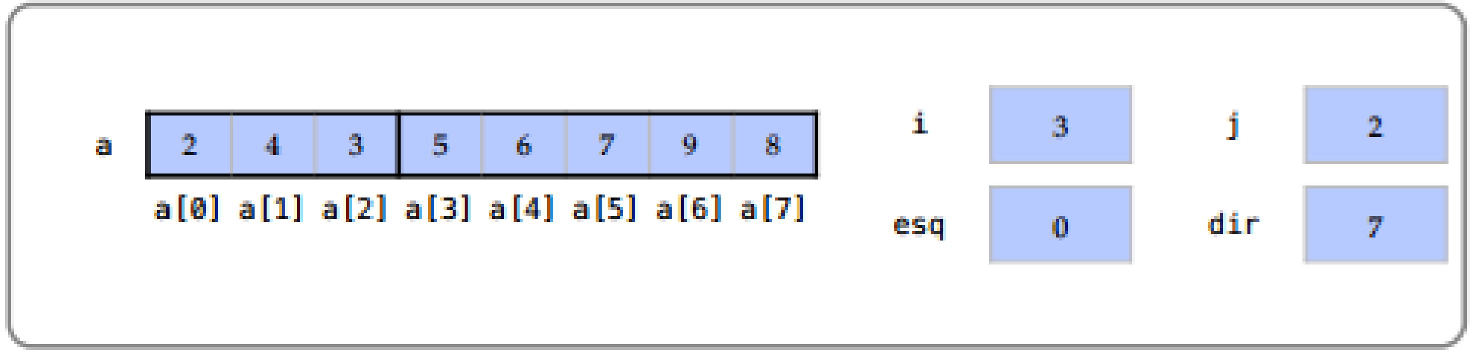


QuickSort

- Como o subarranjo de a[0] a a[2] tem mais de 1 elemento, usamos a própria função para ordená-lo

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

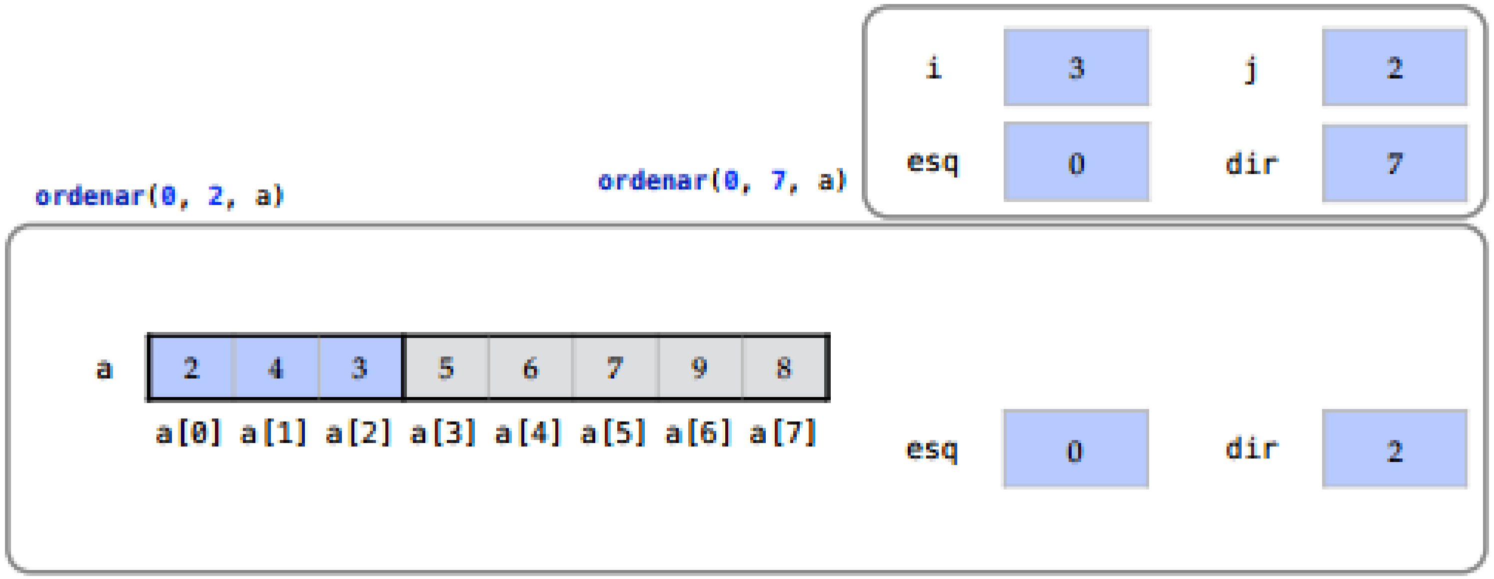
ordenar(0, 7, a)



QuickSort

- A função chamadora vai para a pilha e executamos a função que ordenará a[] das posições 0 a 2

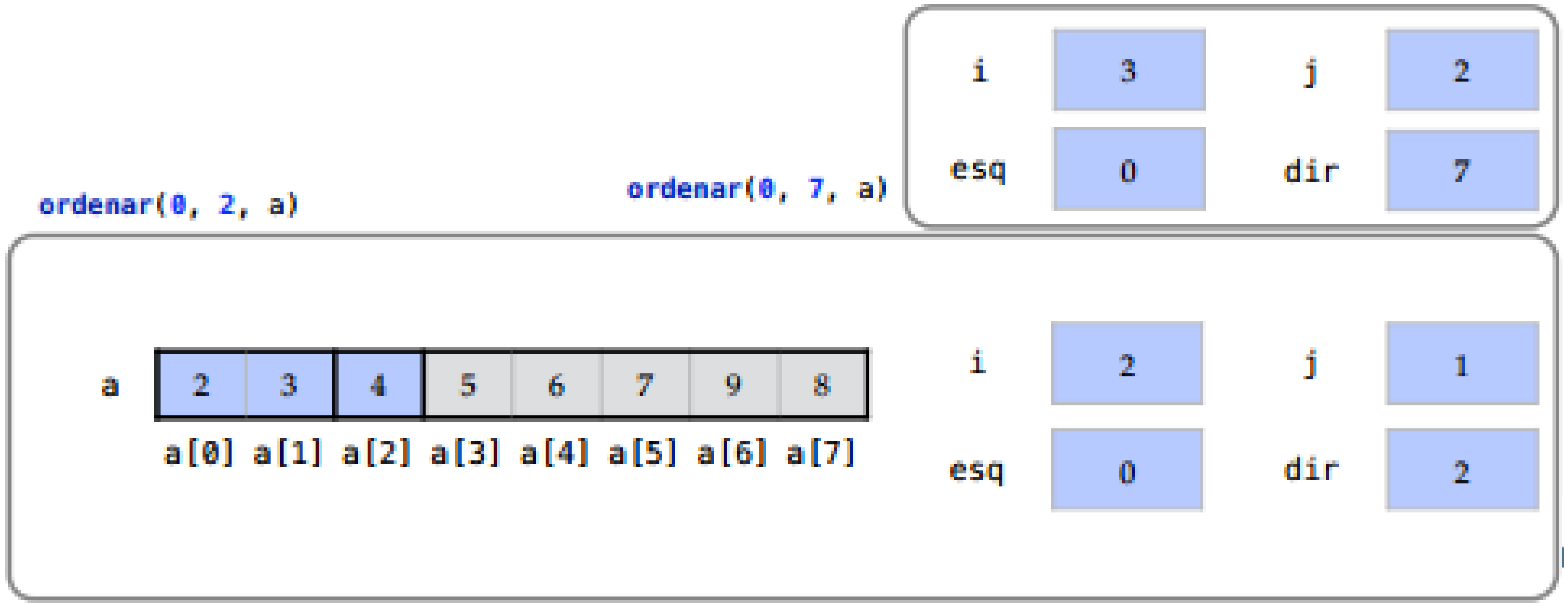
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Os índices são criados e o subarranjo é particionado

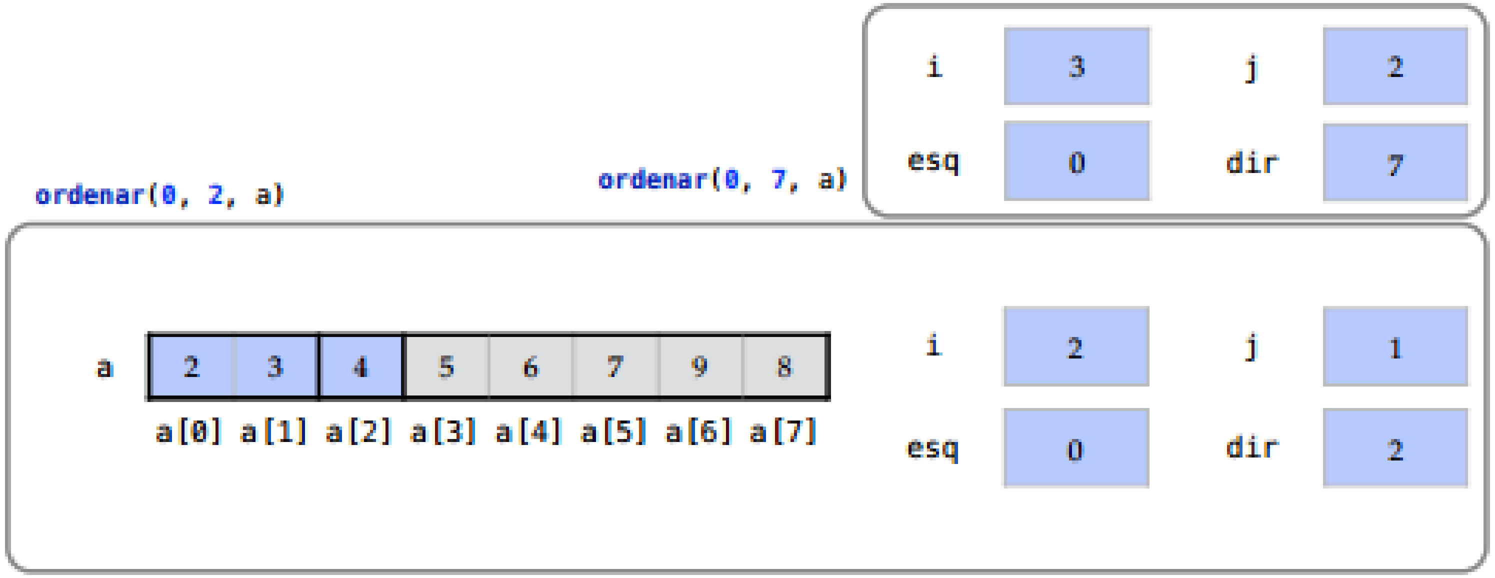
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Esta função é jogada na pilha e chamamos a função de ordenação para o subarranjo de a[esq] até a[j]

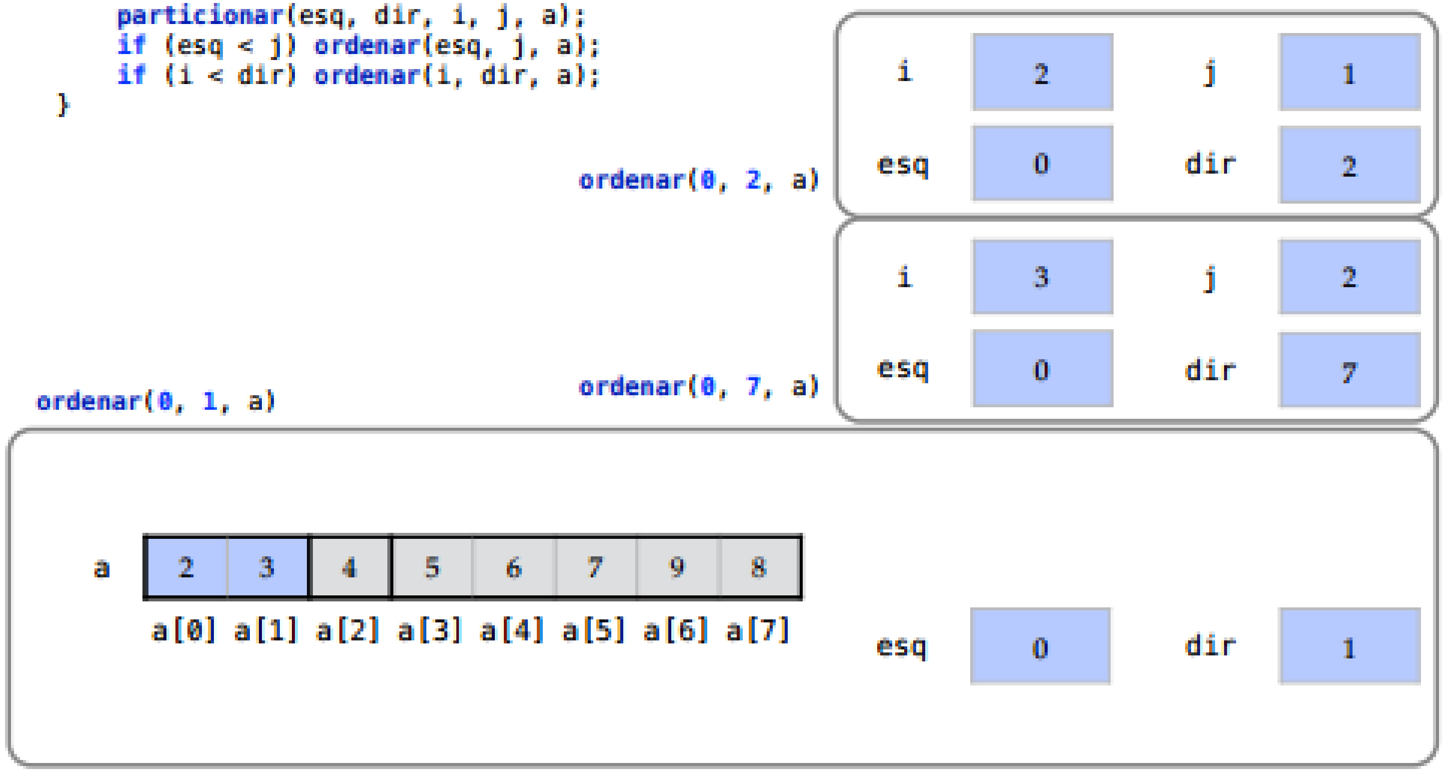
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Esta função é jogada na pilha e chamamos a função de ordenação para o subarranjo de a[esq] até a[j]

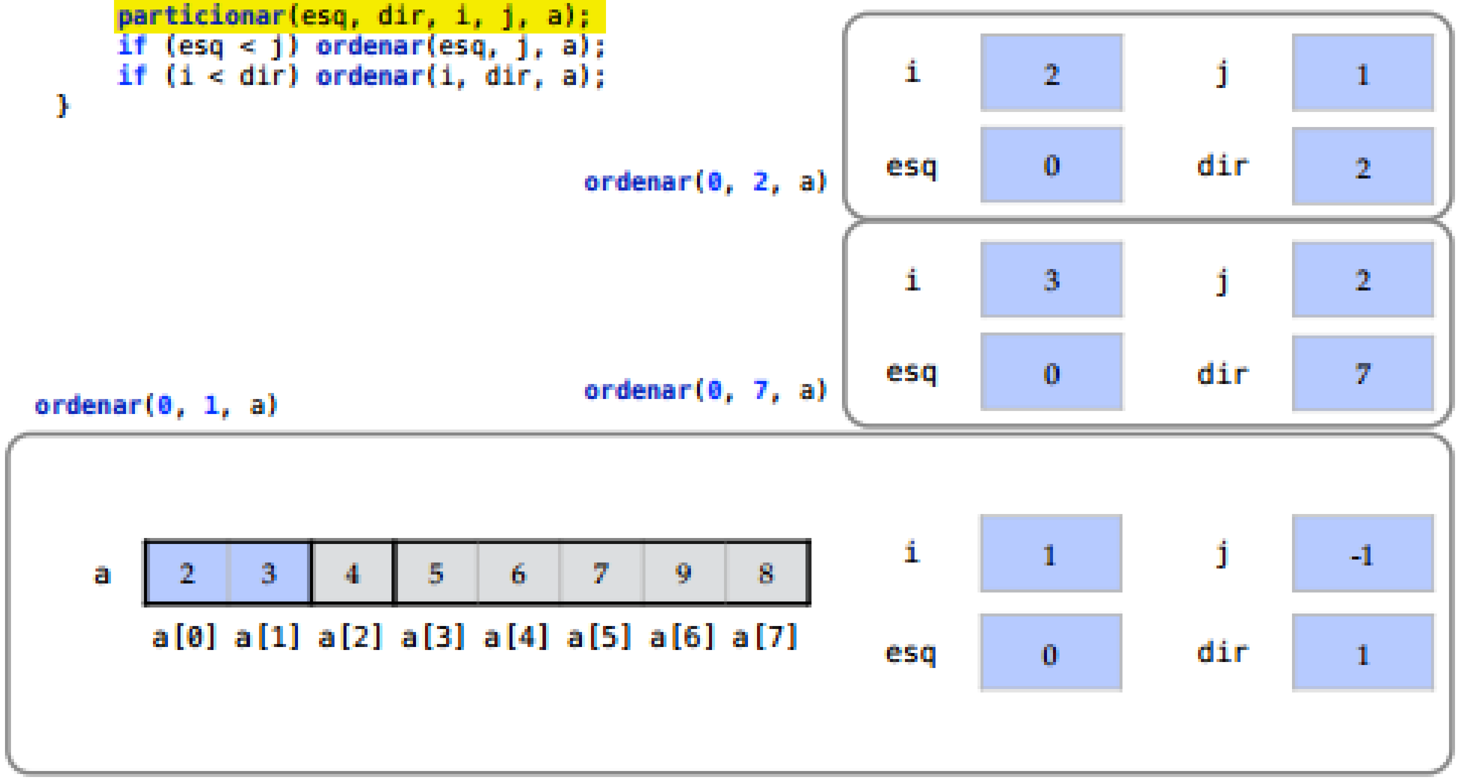
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Os índices são criados e o subarranjo particionado

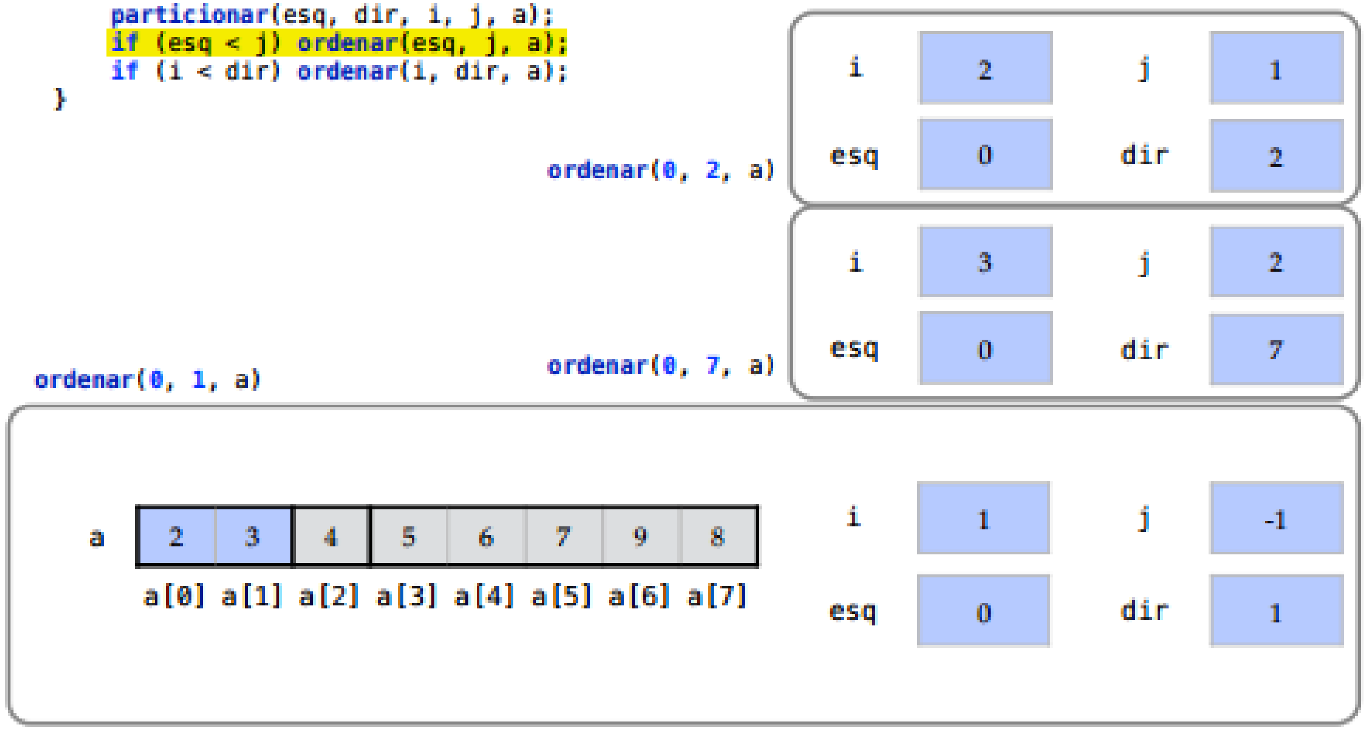
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Como o subarranjo da esquerda não tem nenhum elemento, não executamos esta ordenação

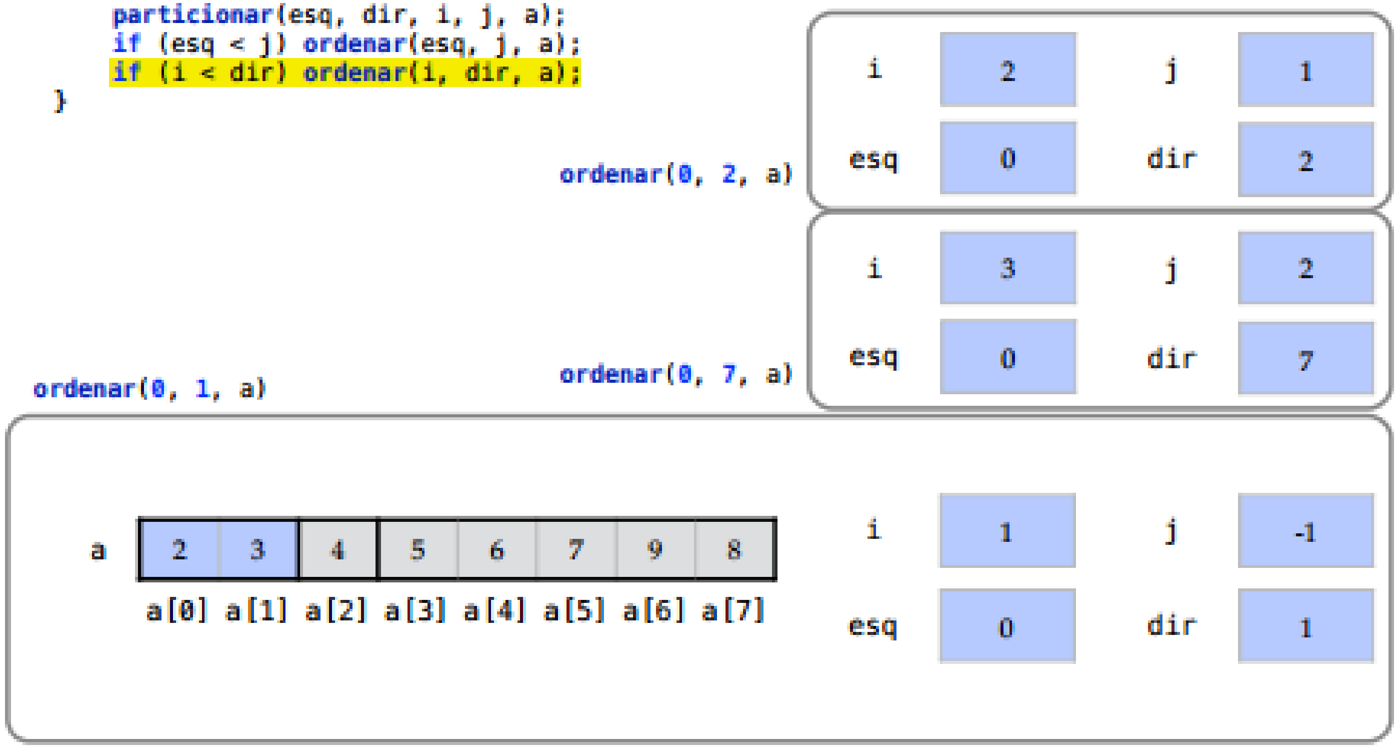
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

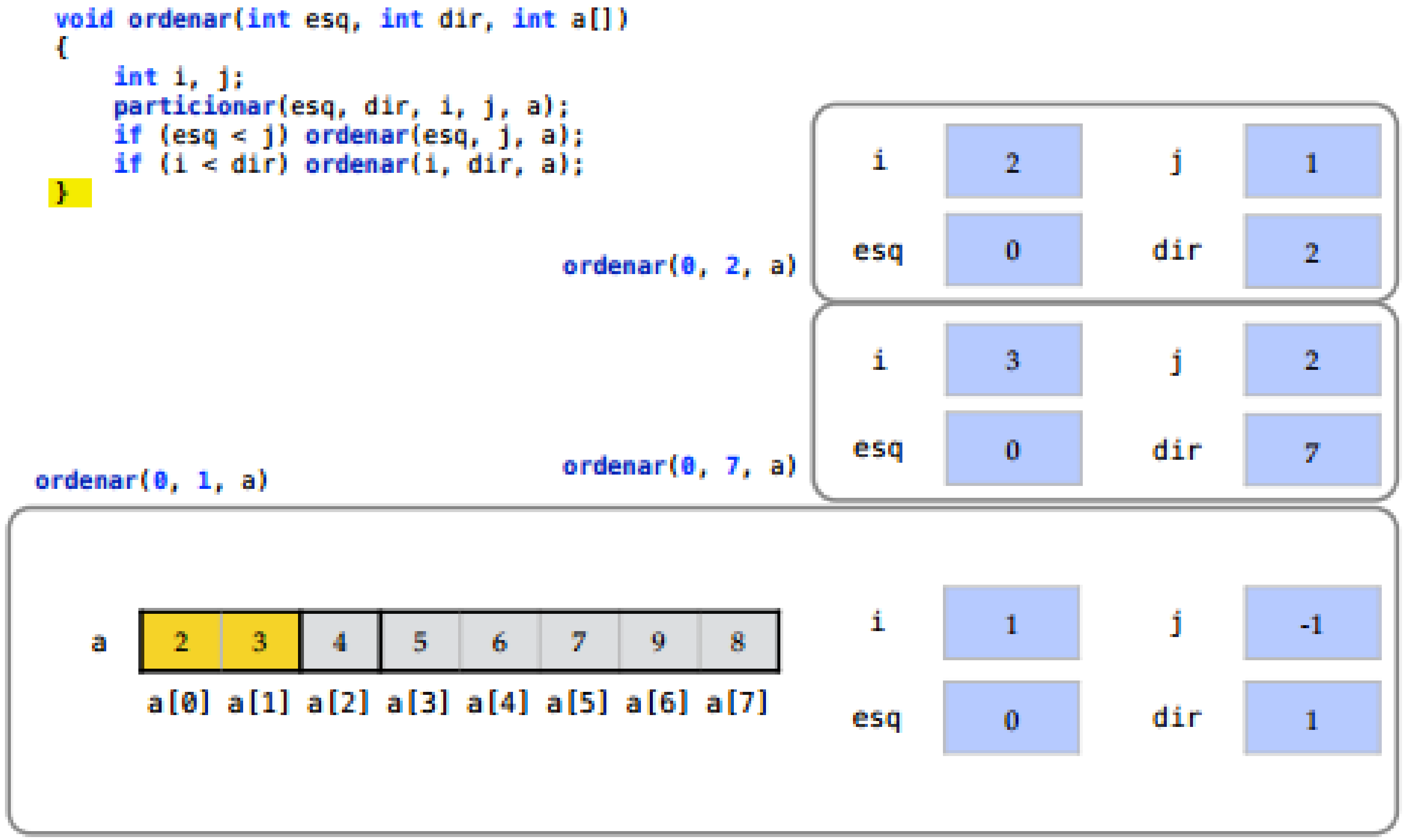
- Como o subarranjo da direita tem apenas 1 elemento, damos este subarranjo como ordenado

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

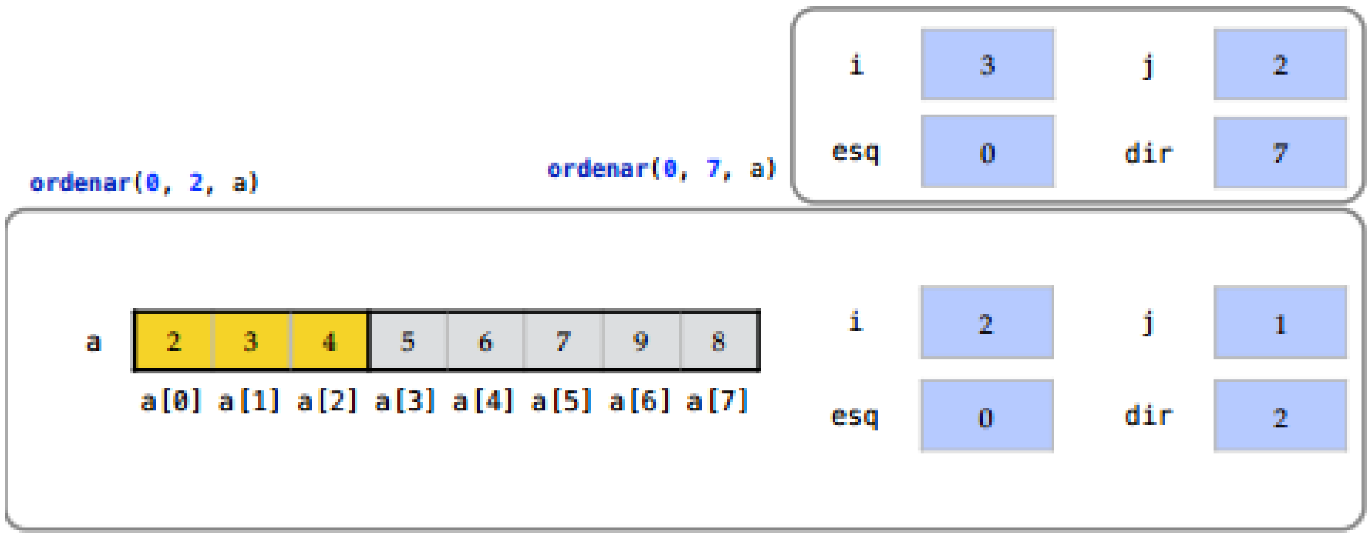
- Ao fim desta função, temos o subarranjo de 0 até 1 ordenado e voltaremos com a função do topo da pilha



QuickSort

- O próximo comando desta função seria ordenar o subarranjo da direita, mas isto não é necessário pois este subarranjo tem apenas 1 elemento

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

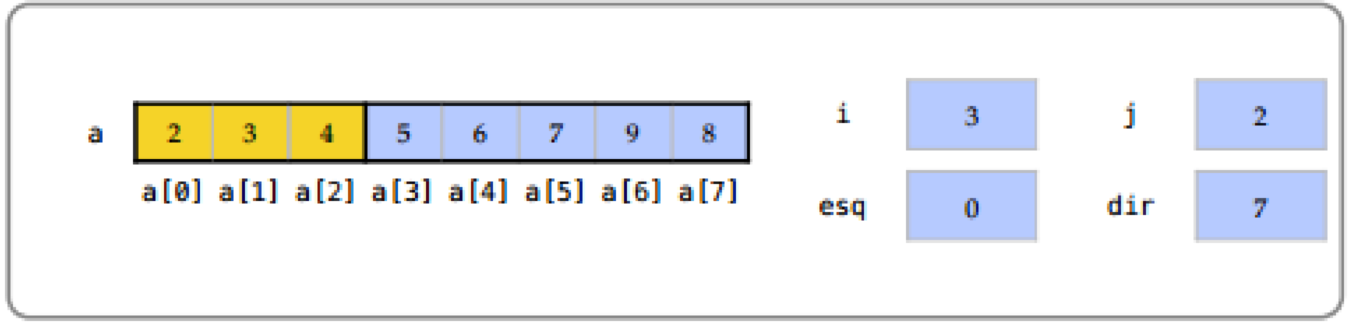


QuickSort

- Encerramos a função e voltamos para a função do topo da pilha. Esta função chama então a ordenação dos elementos de a[3] até a[7]

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

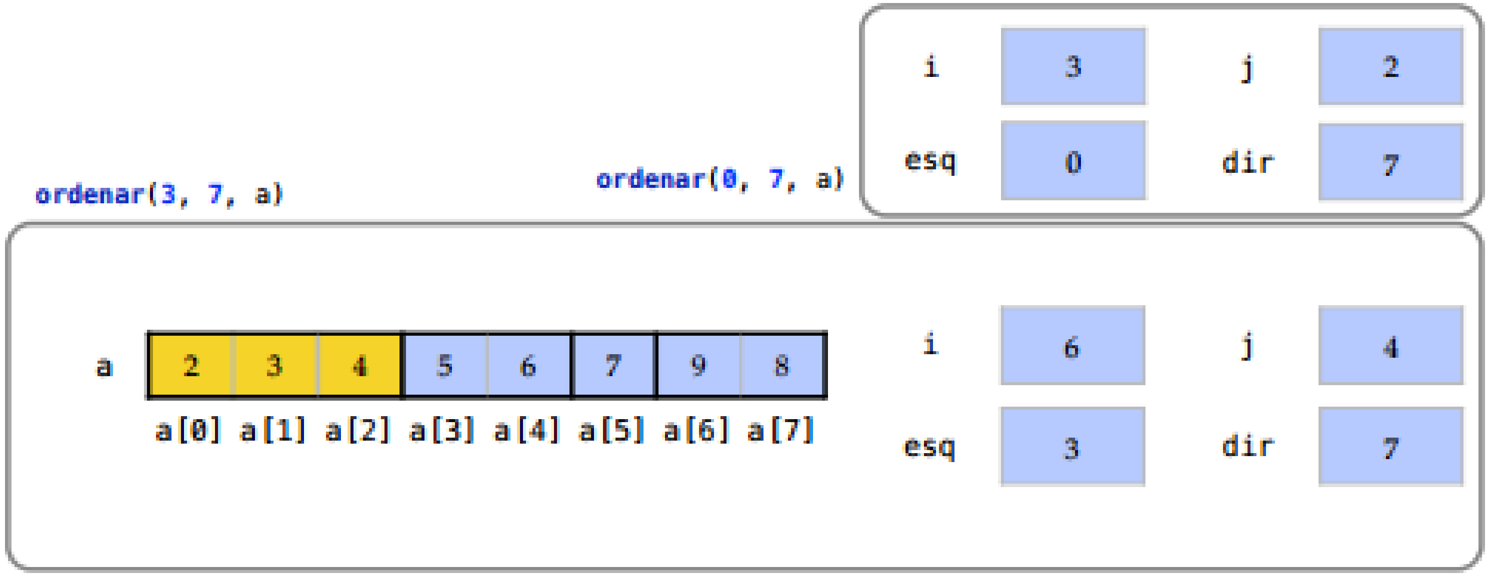
`ordenar(0, 7, a)`



QuickSort

- Criamos os índices e particionamos este subarranjo

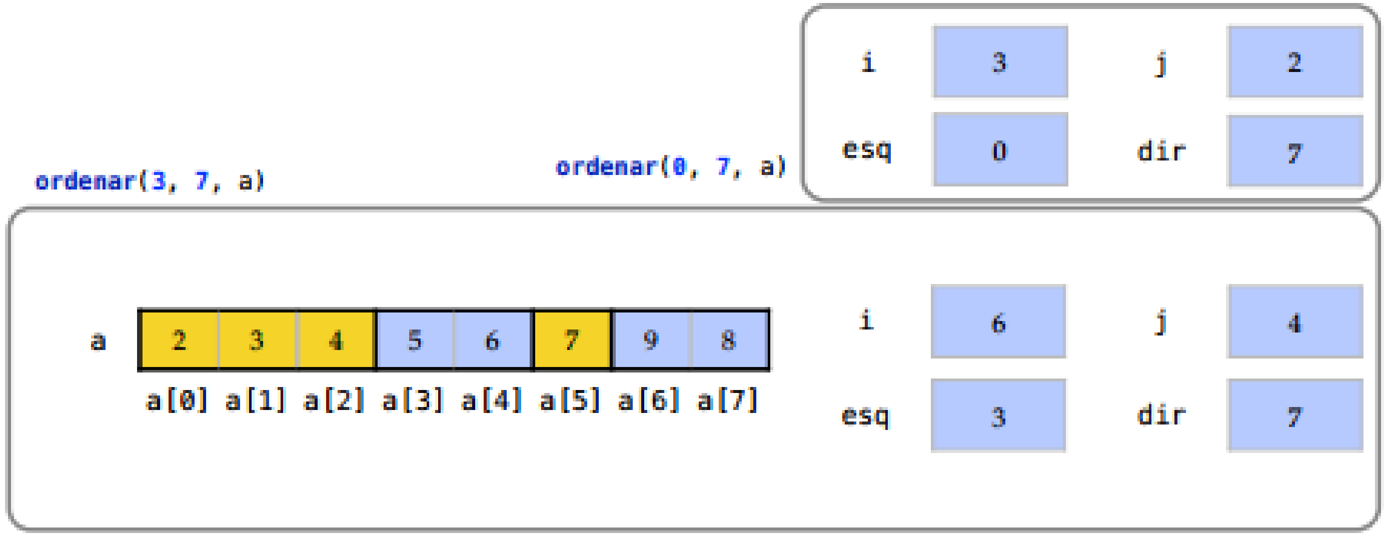
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Entre os subarranjos de a[3] até a[4] e de a[6] até a[7] temos isolado o a[5], que por está sozinho, está também, por definição, ordenado

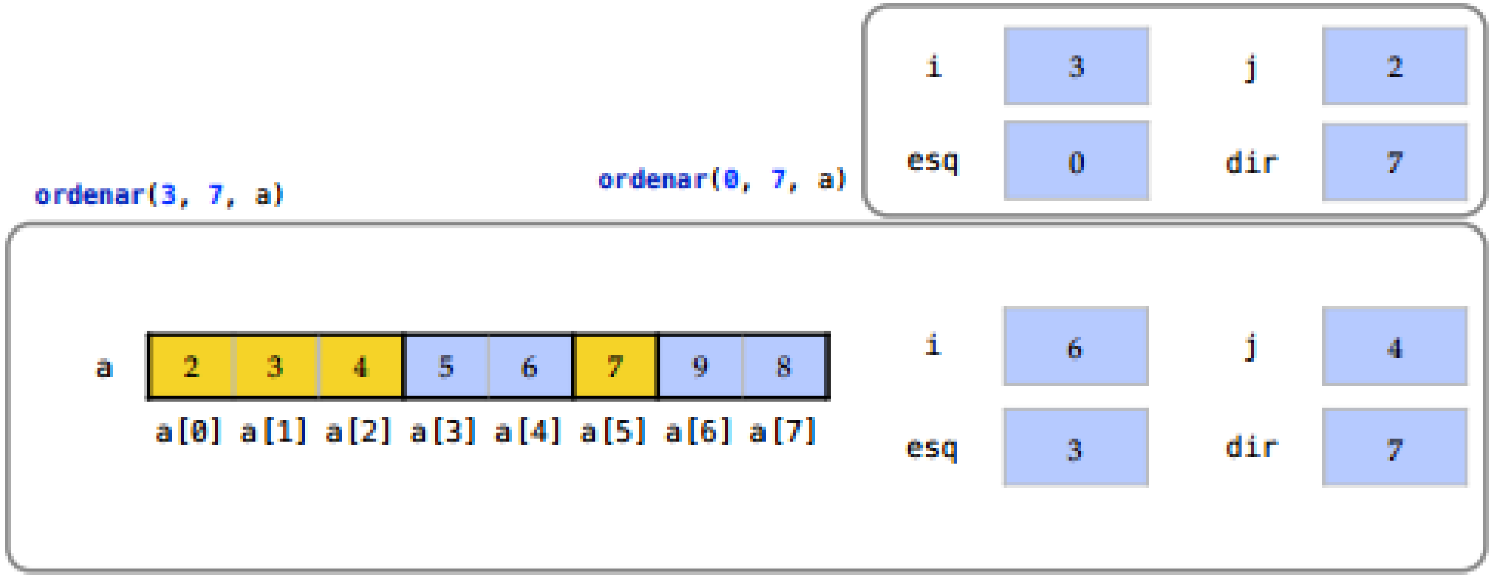
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Com uma chamada recursiva, ordenamos a[3] e a[4]

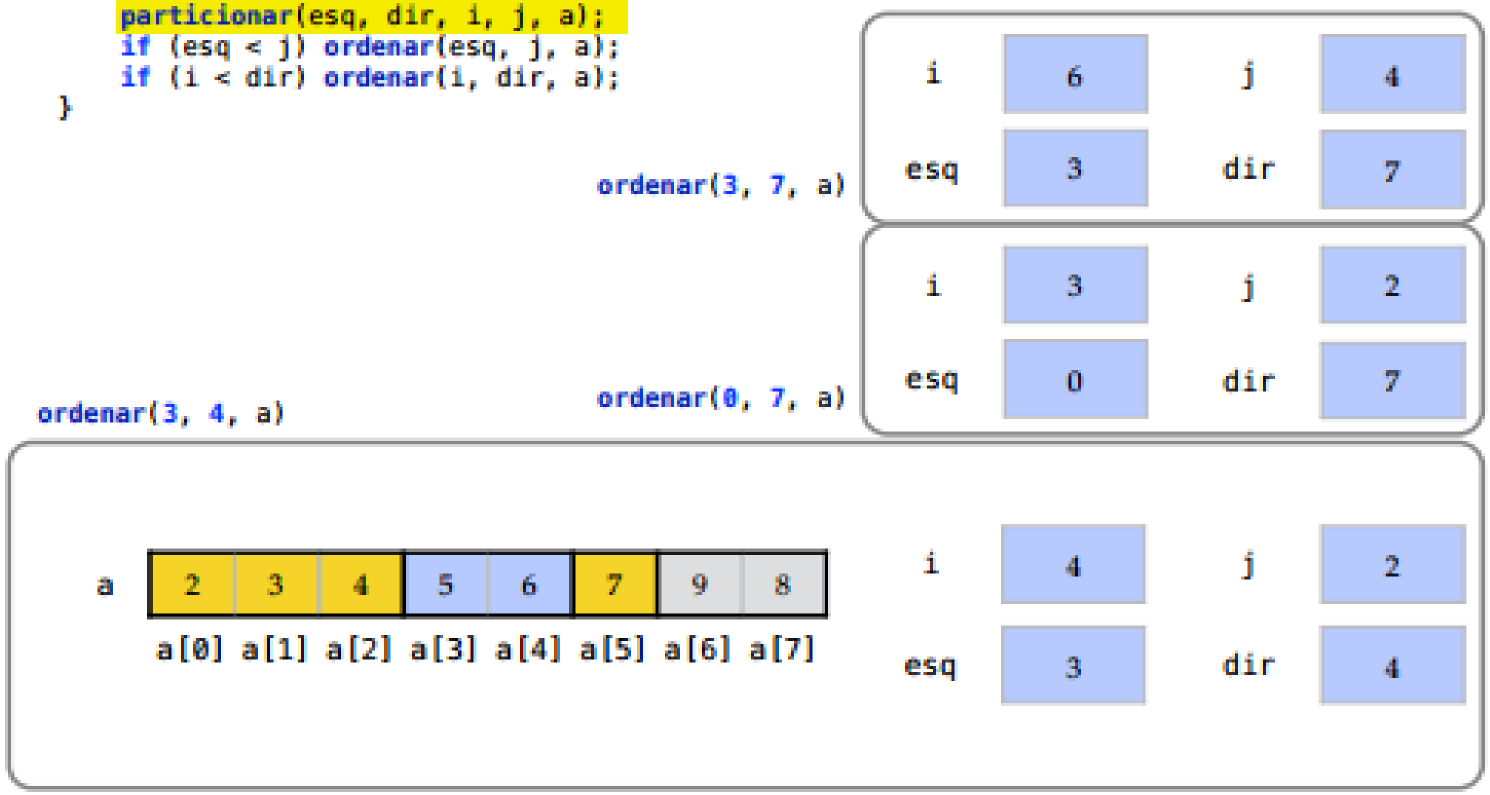
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Criamos os índices e particionamos

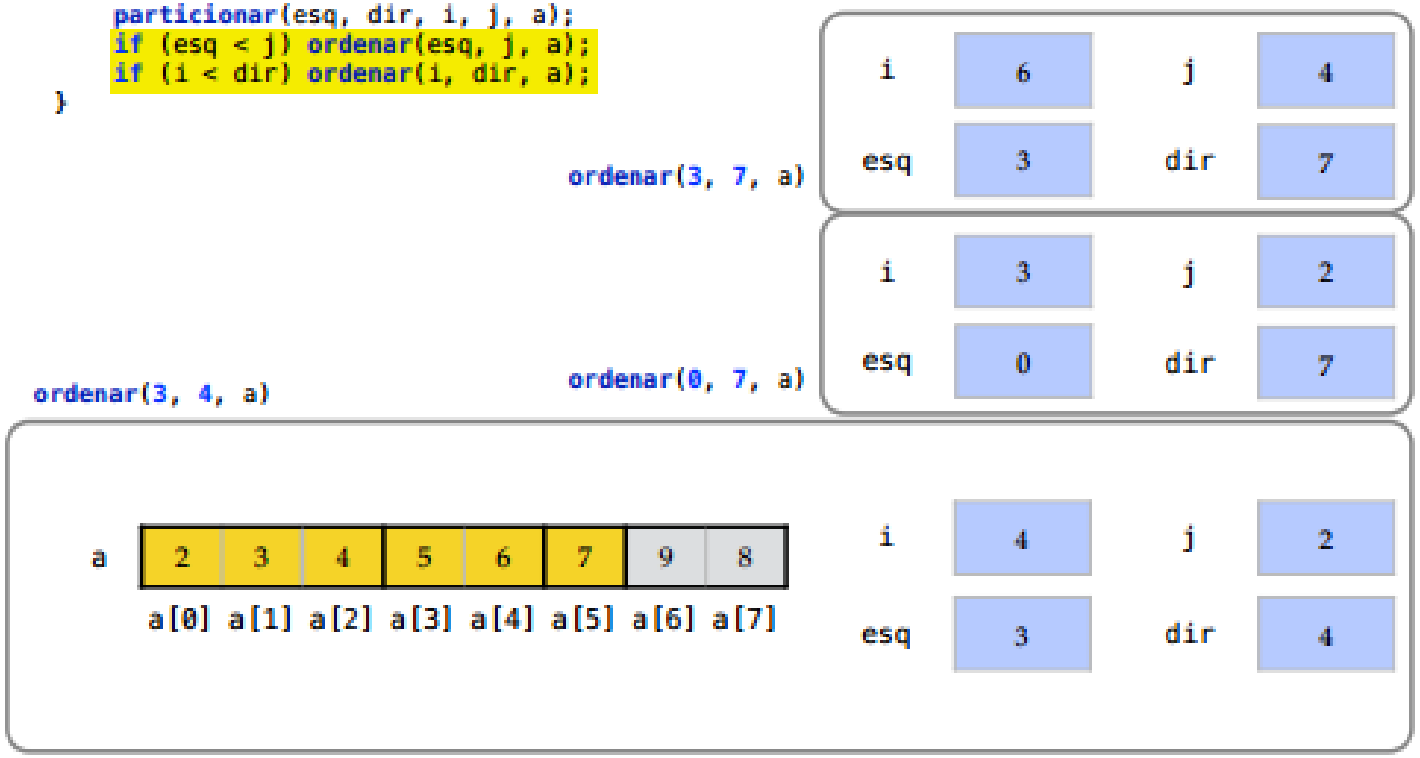
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Como nenhum dos subarranjos têm mais de um elemento, então, todos os elementos estão ordenados

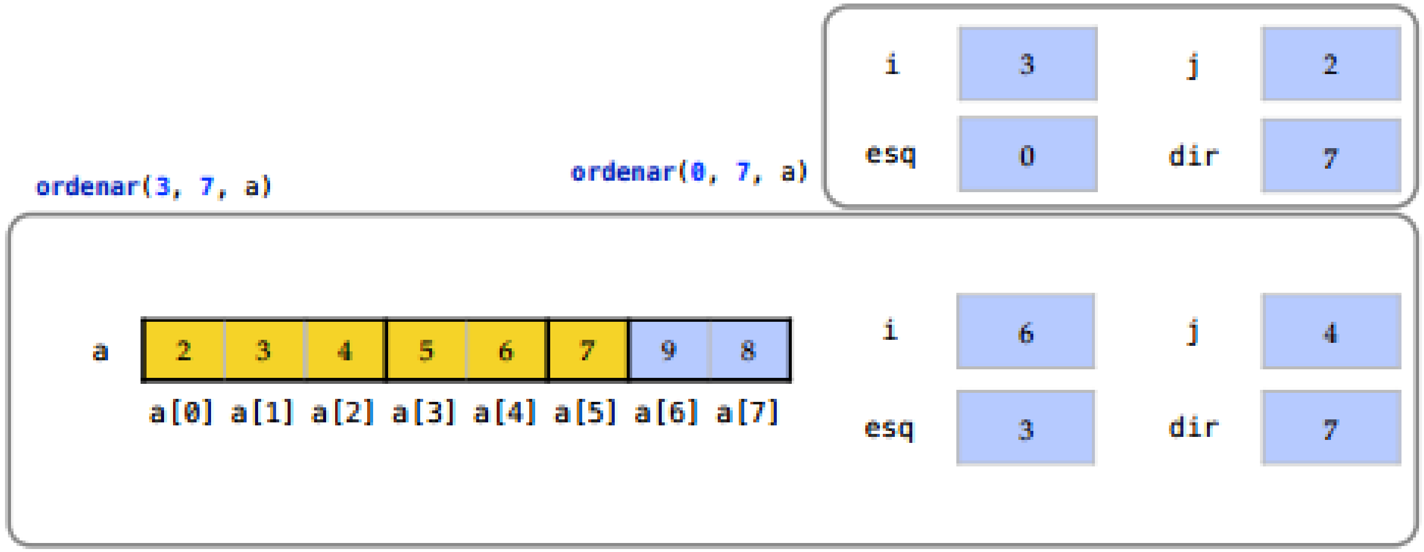
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- A função que sai da pilha ainda precisa ordenar os elementos de a[6] até a a[7]

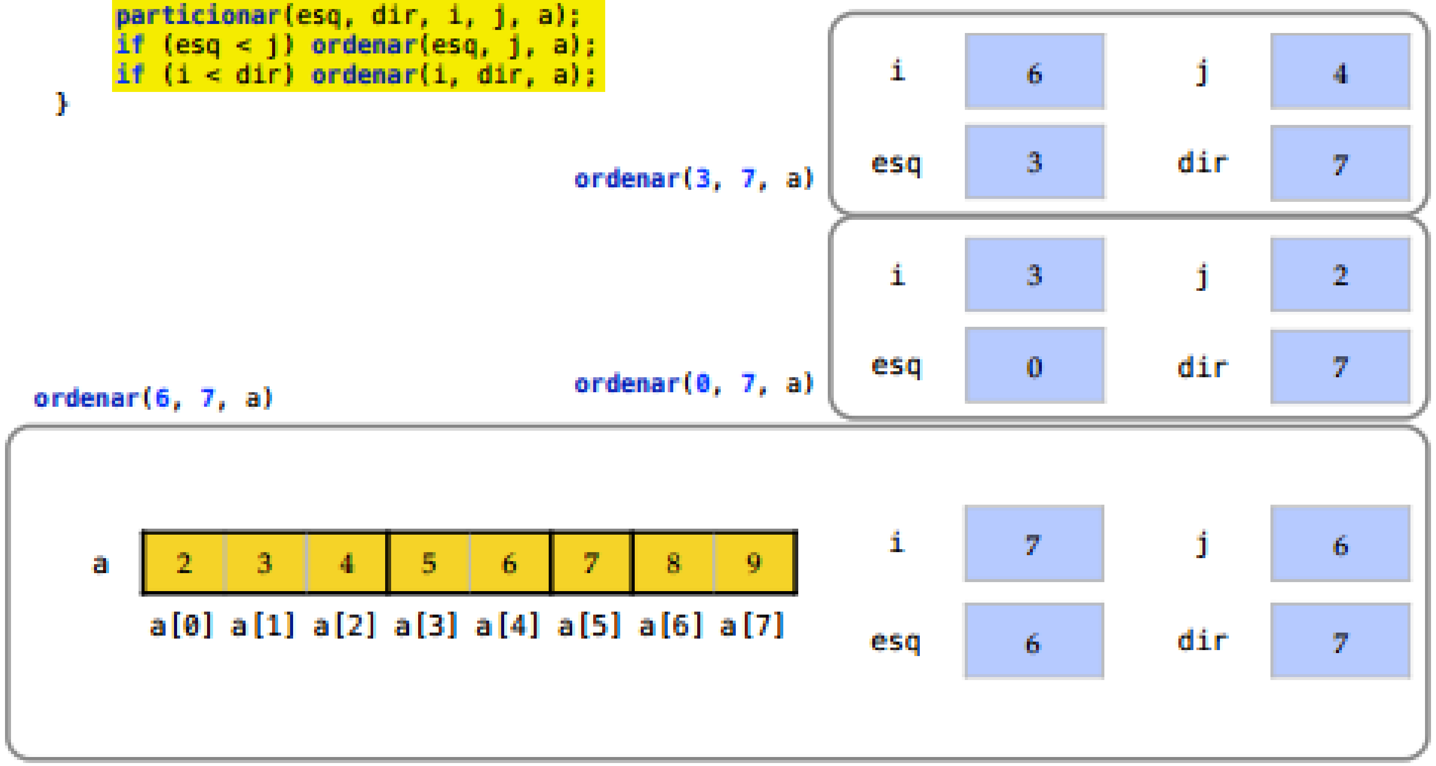
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Esta função também leva a dois subarranjos de tamanho 1, que por isto já estão ordenados

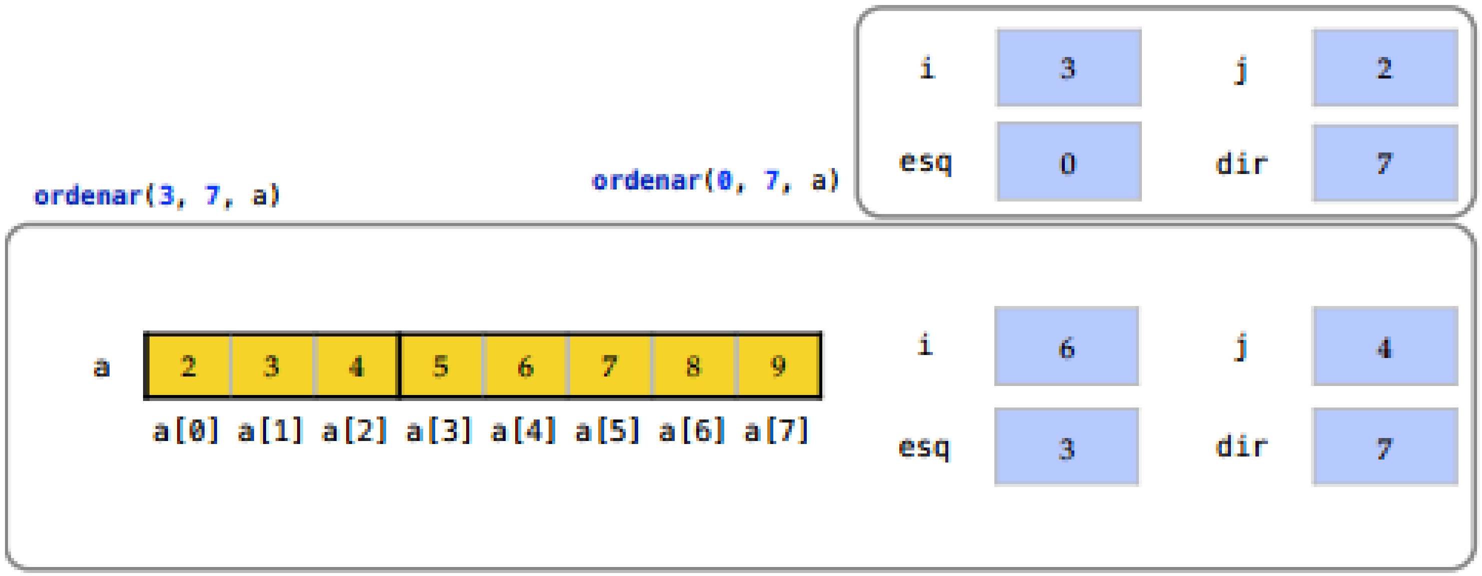
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Assim sendo, a função corrente sai da pilha...

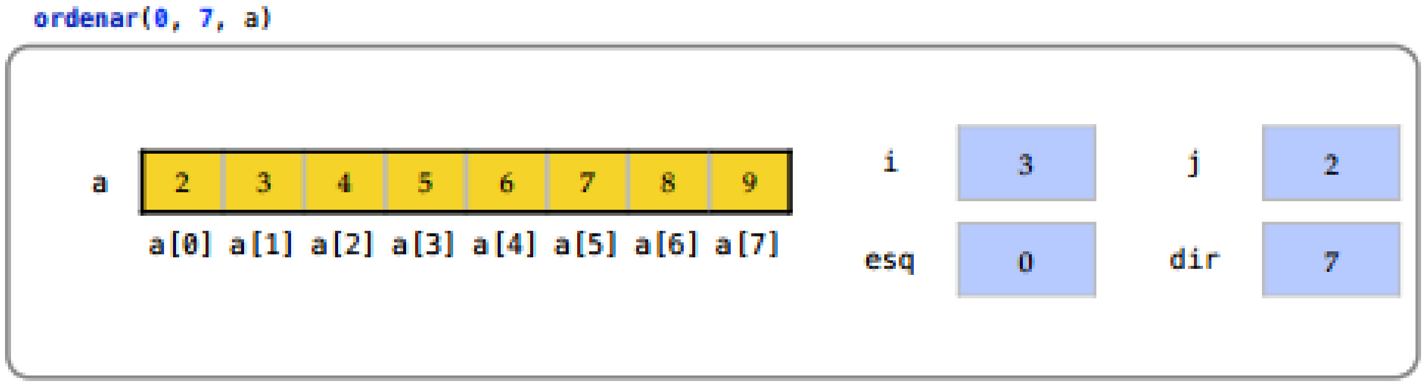
```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- E a função original sai da pilha, finalizando o método com o arranjo original ordenado

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```



QuickSort

- Um inconveniente deste método é que precisamos passar para a função três parâmetros quando queremos ordenar todo o arranjo

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

QuickSort

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

- Para não precisarmos fazer isto, criamos um método auxiliar que é chamado para ordenar todo o arranjo, iniciando de a[0]

```
void quicksort(int a[], int n)
{
    ordenar(0, n-1, a);
}
```

QuickSort

```
void ordenar(int esq, int dir, int a[])
{
    int i, j;
    particionar(esq, dir, i, j, a);
    if (esq < j) ordenar(esq, j, a);
    if (i < dir) ordenar(i, dir, a);
}
```

- Deste modo, mantemos o padrão com os outros métodos de ordenação

```
void quicksort(int a[], int n)
{
    ordenar(0, n-1, a);
}
```

QuickSort - Análise

- O QuickSort é também um típico exemplo de algoritmo O (n log n)

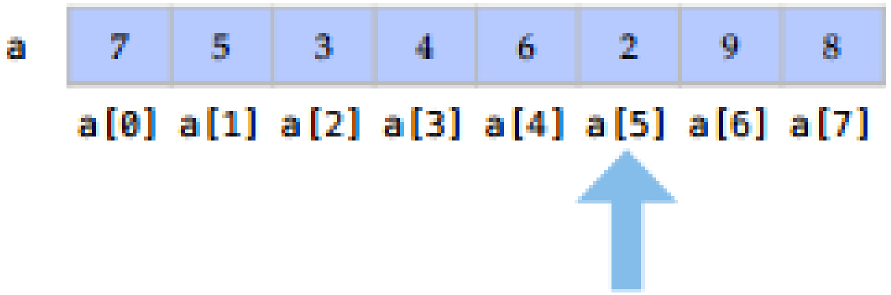
- Algoritmos desta classe são os que dividem um problema em dois problemas menores e depois os juntam fazendo uma operação em cada um dos elementos

QuickSort - Análise

- Assim, a escolha do pivô é um fator determinante no desempenho do algoritmo pois dependemos dele para realmente dividir o arranjo em dois subarranjos de tamanho similar

QuickSort - Análise

- Imagine um algoritmo onde o pivô escolhido é sempre o menor elemento do conjunto de n elementos



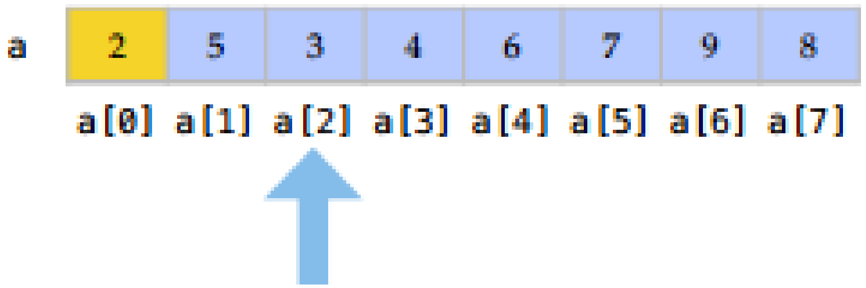
QuickSort - Análise

- Ordenaremos então o subarranjo da direita que ainda tem n-1 elementos

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

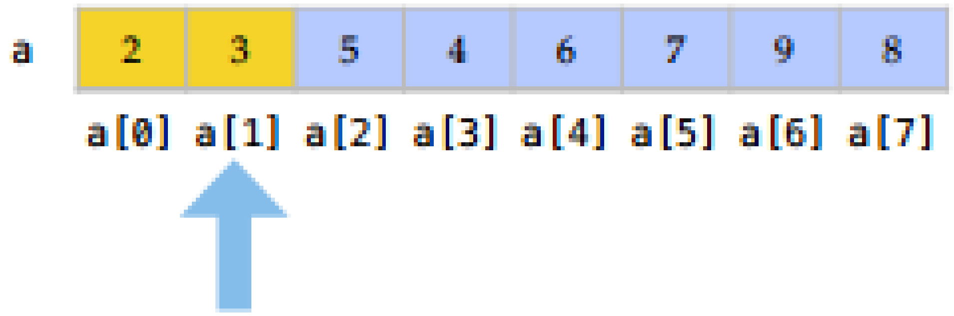
QuickSort - Análise

- Se em todos os passos escolhermos o pior pivô, não conseguiremos dividir o problema pela metade a cada passo



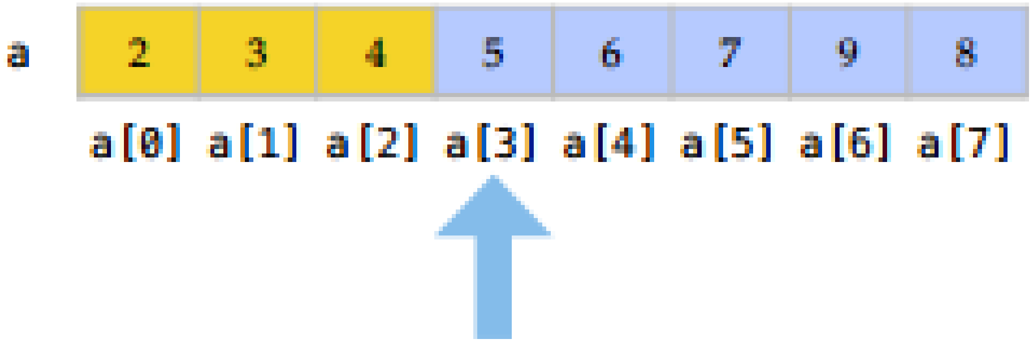
QuickSort - Análise

- Em 2 passos de partição ordenamos apenas 2 elementos. Com a pior escolha de pivô a cada passo, precisaríamos de n passos de partição para ordenar todo o arranjo



QuickSort - Análise

- Como a cada passo estamos achando o menor elemento do arranjo e o colocando em sua posição correta, este pior caso do QuickSort equivale exatamente à ordenação por seleção, que tem custo $O(n^2)$



QuickSort - Análise

- Assim, o algoritmo QuickSort tem os seguintes casos:
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n \log n)$
  - Caso médio: $O(n \log n)$
- O pior caso é muito pouco provável, o que mantém o caso médio em $O(n \log n)$

QuickSort - Análise

- Algumas estratégias podem ser utilizadas para se escolher um melhor pivô, melhorando a eficiência do algoritmo e deixando o pior caso ainda menos provável

QuickSort – Pivô ótimo

- O melhor pivô para um passo de repartição seria mediana dos valores no arranjo
  - Um pivô igual à mediana dividiria o arranjo em dois arranjos de tamanho idêntico
  - Porém, obter a mediana de um arranjo desordenado seria um processo caro

QuickSort – Pivô ótimo

- Estratégias comuns para melhores pivôs são:
  - Escolher a cada passo um elemento aleatório do arranjo
  - Escolher três elementos aleatórios do arranjo e usar a mediana dos três como pivô

QuickSort – Vantagens

- É um método muito eficiente para ordenar dados

- Somente devido à pilha de funções precisa de apenas um espaço extra muito pequeno, que é $O(\log n)$

- Requer apenas $O(n \log n)$ comparações

- Apesar da mesma ordem de complexidade média, é usualmente mais rápido que o MergeSort no caso médio

QuickSort – Desvantagens

- Tem um pior caso $O(n^2)$, o que pode ser um fator importante em aplicações críticas, mesmo que com uma baixa probabilidade

- Não é um algoritmo estável pois as trocas na fase de partição não consideram a ordem relativa dos elementos

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	$\mathbf{O(n \log n)}$	$\mathbf{O(n \log n)}$
Estabilidade	Não	Sim	Sim	Não
Adaptabilidade	Não	Sim	Não	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$
Estabilidade	Não	Sim	Sim	Não

Entre os métodos eficientes, Quicksort é mais rápido na média do que o Mergesort, apesar de terem mesma ordem de grandeza.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$\boxed{O(n)}$	$O(n \log n)$	$O(n \log n)$
Caso médio	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	$\mathbf{O(n \log n)}$	$\mathbf{O(n \log n)}$
Estabilidade	Não	Sim	Sim	Não

Se os elementos já estivem ordenados, a ordenação por Inserção é sempre o método mais rápido. Este melhor caso, porém, é pouco provável na maioria das aplicações.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$\boxed{O(n)}$	$O(n \log n)$	$O(n \log n)$
Caso médio	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Estabilidade	Não	Sim	Sim	Não

A ordenação por inserção é interessante para se adicionar alguns elementos a um arranjo já ordenado. Assim, estaremos próximos de seu melhor caso.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Apesar de ter a mesma ordem de grandeza da ordenação por seleção, a ordenação por inserção é a mais lenta para arranjos em ordem decrescente. Mesmo assim, pode continuar melhor que a ordenação por seleção.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$
Estabilidade	Não	Sim	Sim	Não

Apesar da mesma ordem de grandeza, a ordenação por inserção tende a ser melhor que a ordenação por seleção no caso médio com arranjos aleatórios.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$
Estabilidade	Não	Sim	Sim	Não

Nos casos gerais, a inserção é um método interessante apenas para arranjos bastante pequenos, com menos de 20 elementos.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Adaptabilidade	Não	Sim	Não	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Se os registros são grandes, as movimentações se tornam caras e a ordenação por seleção se torna interessante, caso não haja tantos elementos.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pio				
Melh				
Cas				
Estal				
Adaptabilidade	Não	Sim	Não	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Porém, outra maneira de evitar movimentos é usar ordenação indireta, ou seja, ordenamos ponteiros para elementos e só fazemos $O(n)$ movimentações no final.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesor	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
[illegible]	[illegible]	[illegible]	og n)	$\mathbf{O(n \log n)}$
[illegible]	[illegible]	[illegible]	m	Não
Adaptabilidade	Não	Sim	Não	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

No caso médio, quicksort é a melhor opção para a maioria da situações.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
			og $n)$	$O(n \log n)$
			og $n)$	**$O(n \log n)$**
			m	Não
			ão	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Apesar de no geral ser o melhor método entre os apresentados, ele não garante estabilidade.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
[illegible]	[illegible]	[illegible]	og n)	$O(n \log n)$
[illegible]	[illegible]	[illegible]	m	Não
Adaptabilidade	Não	Sim	Não	Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Suas chamadas recursivas também demandam um pouco de memória extra.

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesort	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Ca[illegible]	[illegible]	[illegible]	[illegible]	$O(n \log n)$
Est[illegible]	[illegible]	[illegible]	[illegible]	Não
Ada[illegible]	[illegible]	[illegible]	[illegible]	Sim
Mov[illegible] na média	[illegible]	[illegible]	[illegible]	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Apesar de haver um pior caso, sua ocorrência é quase impossível para qualquer boa estratégia de seleção de pivôs.

Comparação dos métodos de ordenação

Quando os arranjos já estão quase ordenados, usar o elemento do meio como pivô melhora muito o desempenho do algoritmo. Porém, nem sempre usar o pivô do meio é uma boa estratégia.

Al			...rt	Quicksort
P			...n)	$O(n^2)$
Me			...n)	$O(n \log n)$
Co			...n)	$O(n \log n)$
Est				Não
Ad				Sim
Movimentações na média	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Memória extra	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Comparação dos métodos de ordenação

Algoritmo	Seleção	Inserção	Mergesor	Quicksort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Est				
Adc				
Mo				
Me				

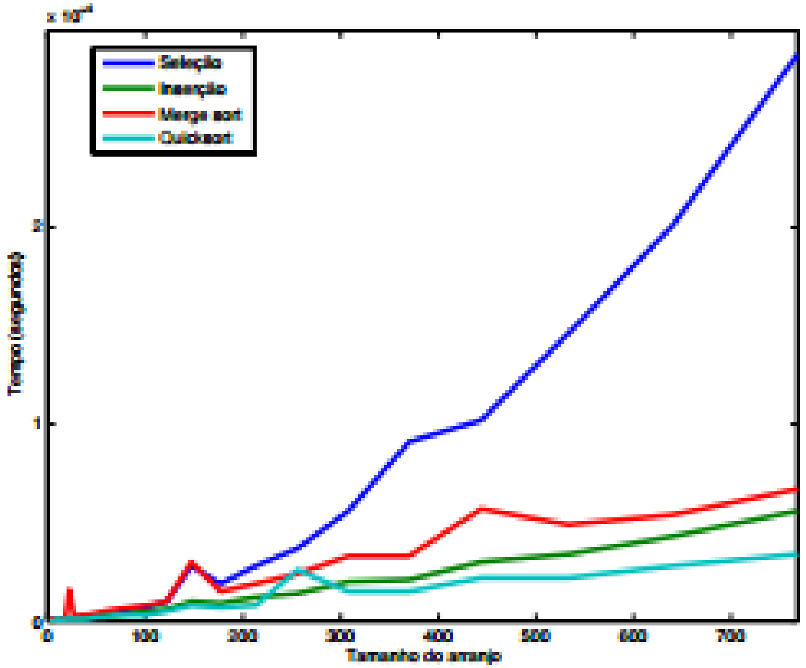
Os métodos de inserção e quicksort podem ser combinados. A inserção pode ordenar subarranjos pequenos do quicksort, o que costuma melhorar muito seu desempenho médio.

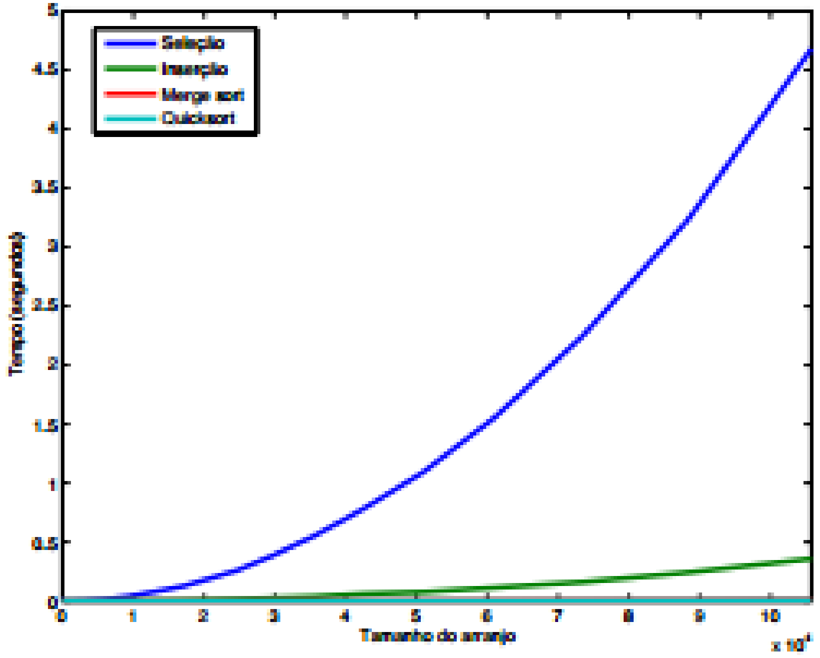
Testes Reais

- Como os MergeSort e QuickSort são O(n log n) no caso médio, sabemos que todos devem ser mais eficientes que os métodos simples
- Entre os dois, podemos fazer comparações reais de tempo para ver a performance em segundos dos algoritmos
- Para as comparações apresentadas a seguir, foram utilizadas versões mais eficientes dos algoritmos

Testes Reais

- Arranjos em ordem inicial aleatória – Comparação com os métodos simples

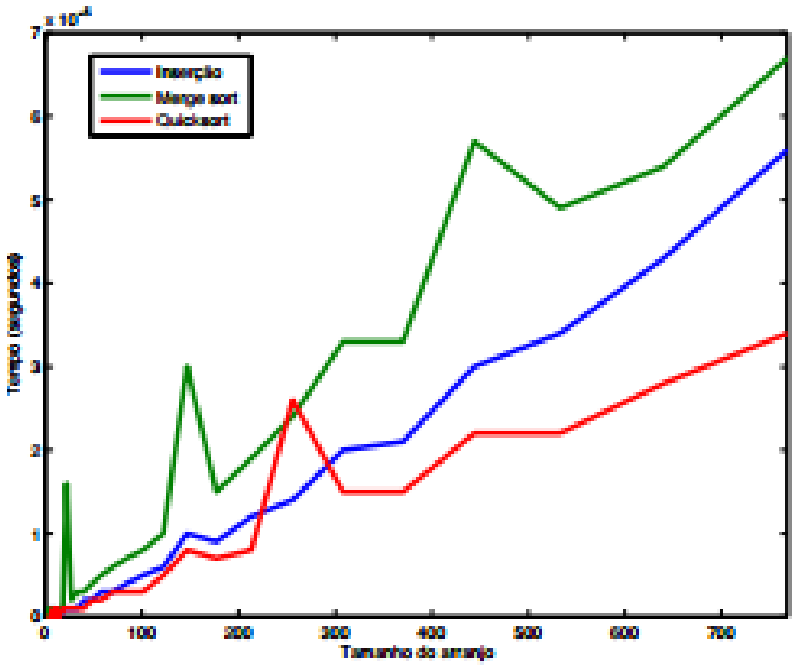


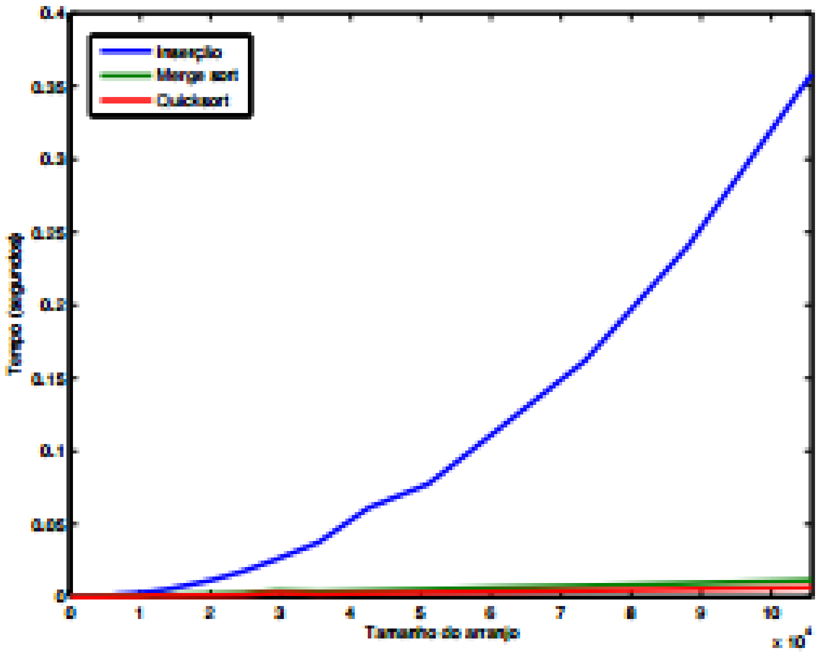


UNIVERSIDADE
DO ESTADO DE MINAS GERAIS
UEMG
UNIDADE DIVINÓPOLIS

Testes Reais

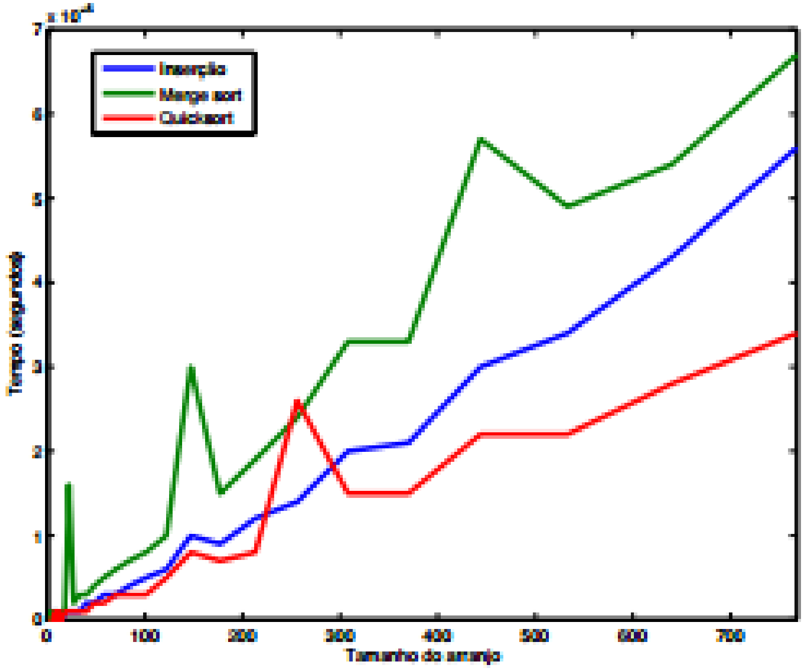
- Mesmo sem a ordenação por seleção, há muita diferença entre os métodos

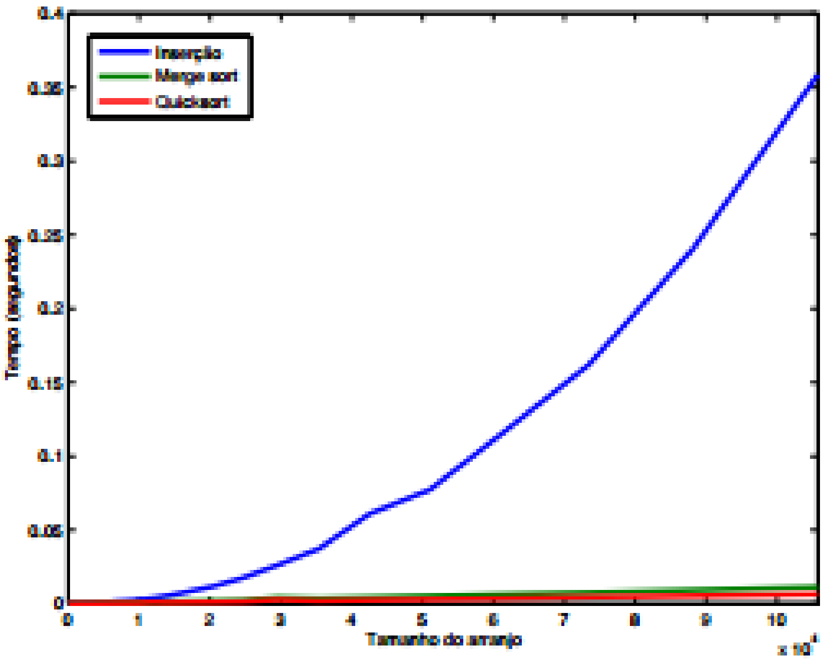




Testes Reais

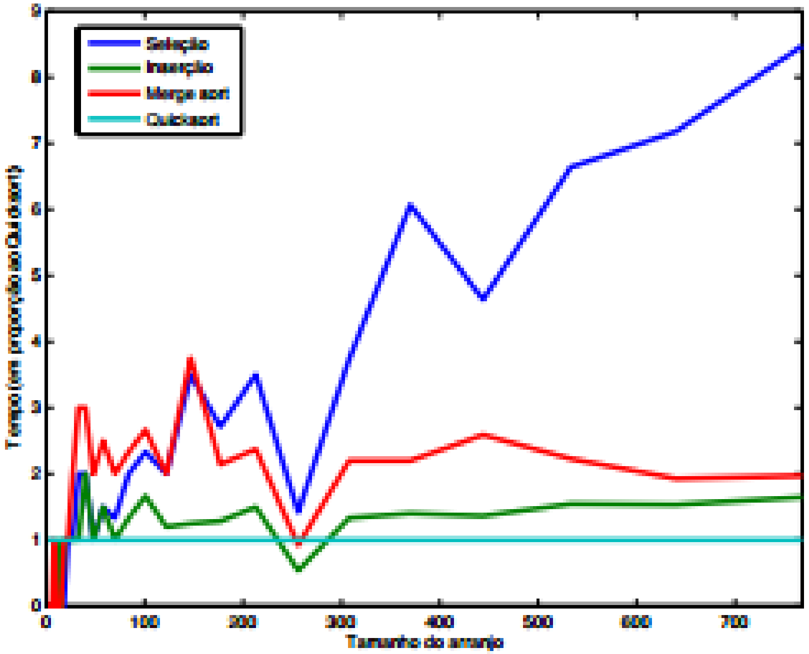
- Para arranjos pequenos, porém, a ordenação por inserção pode ser mais eficiente que o MergeSort

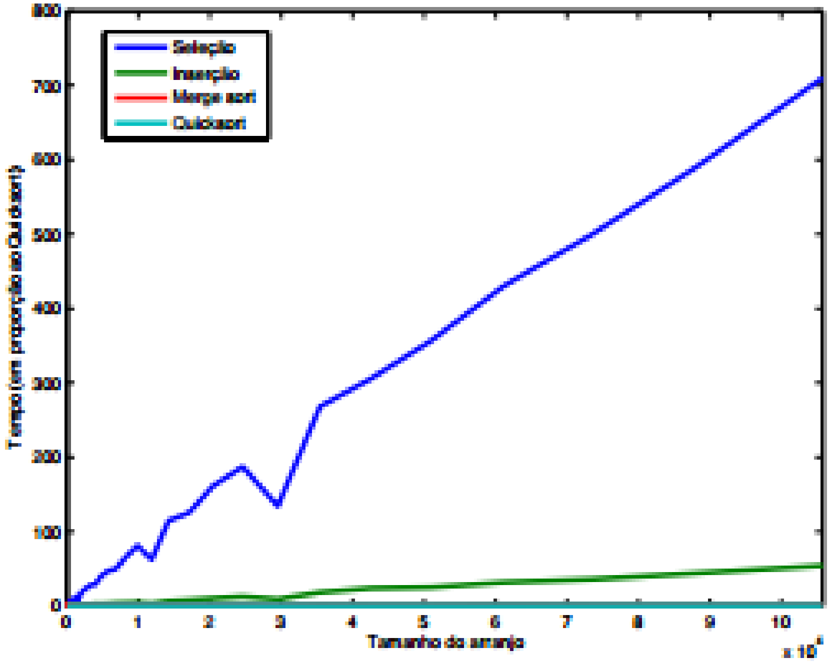




Testes Reais

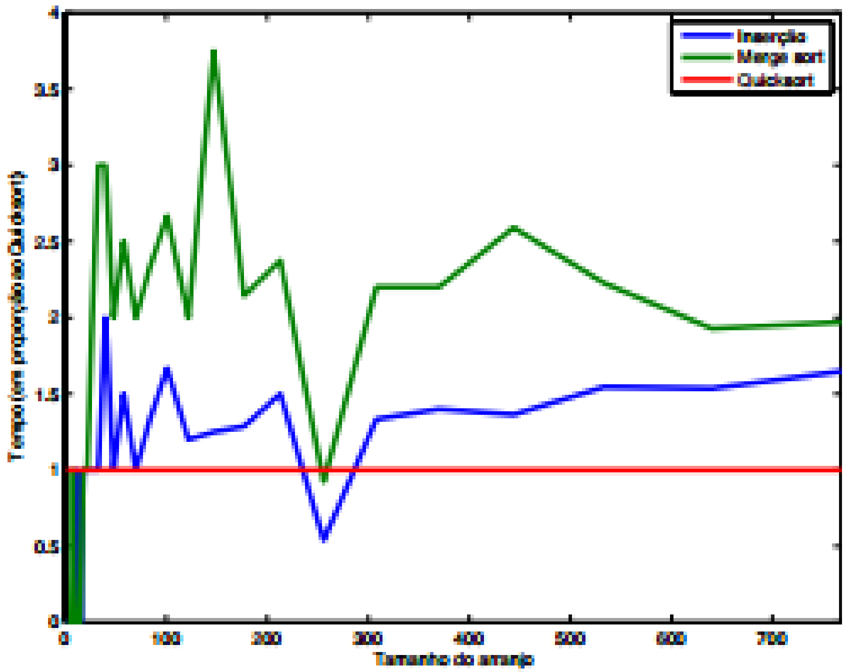
- Como esperado, métodos simples são muito mais lentos. Até 700 vezes mais lentos em arranjos grandes

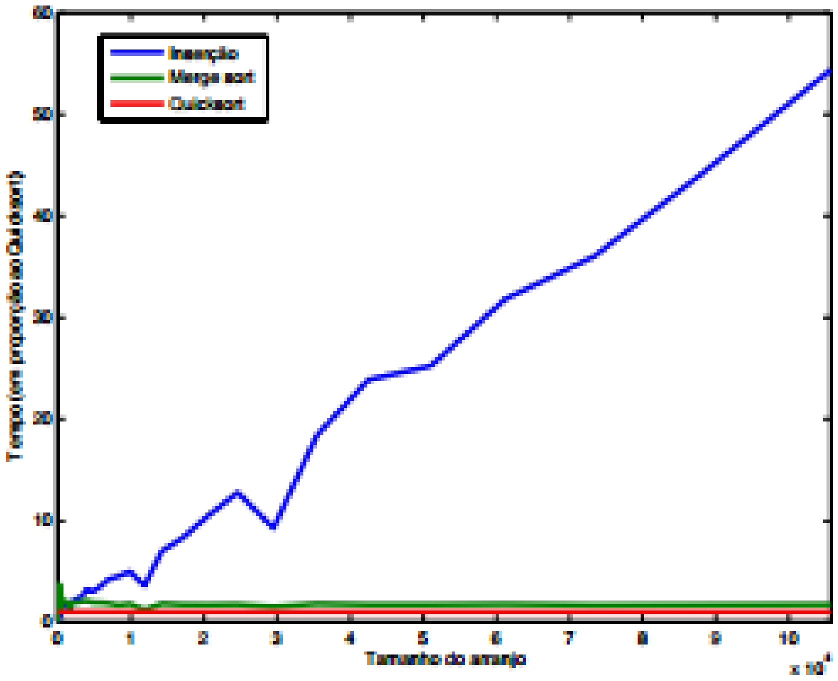




Testes Reais

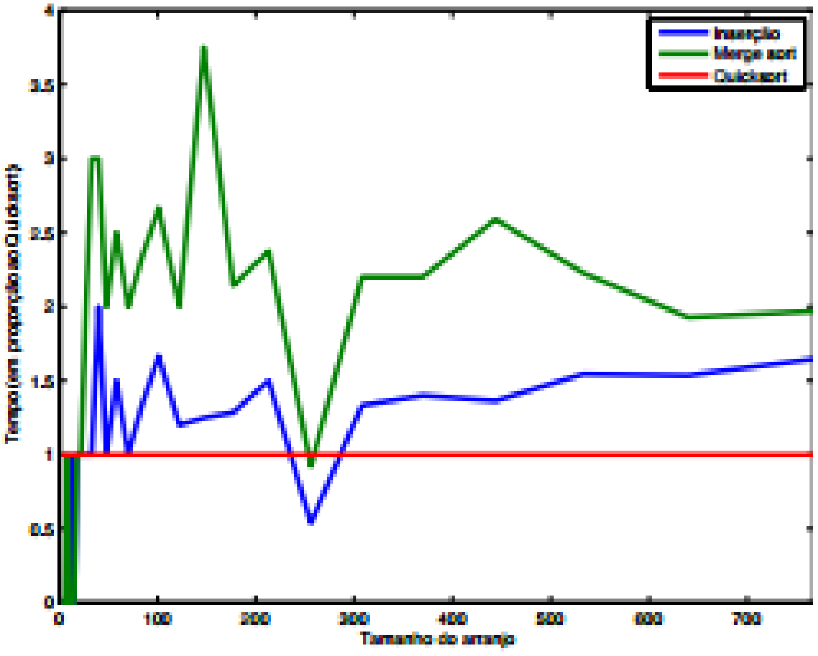
- Mesmo a inserção é cerca de 55 vezes mais lenta para arranjos grandes

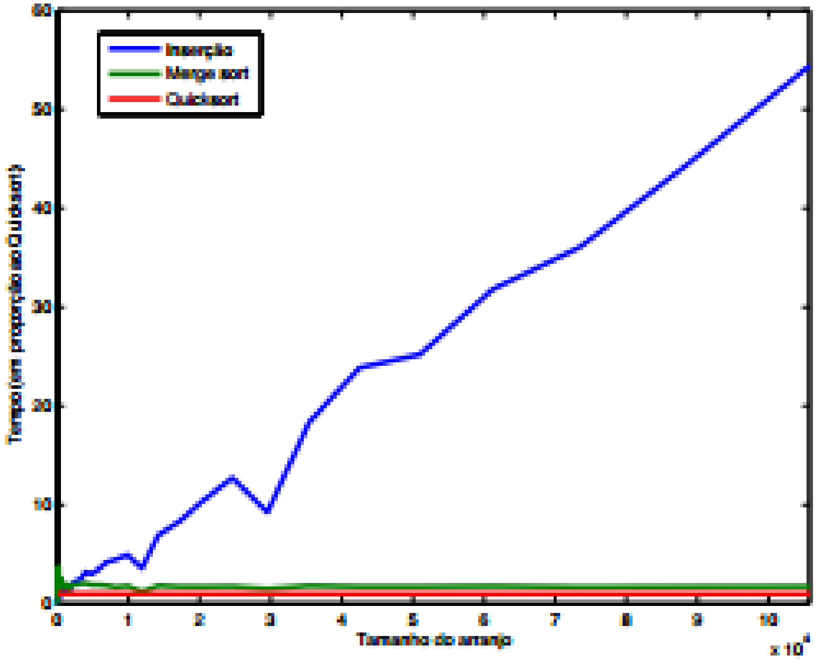




Testes Reais

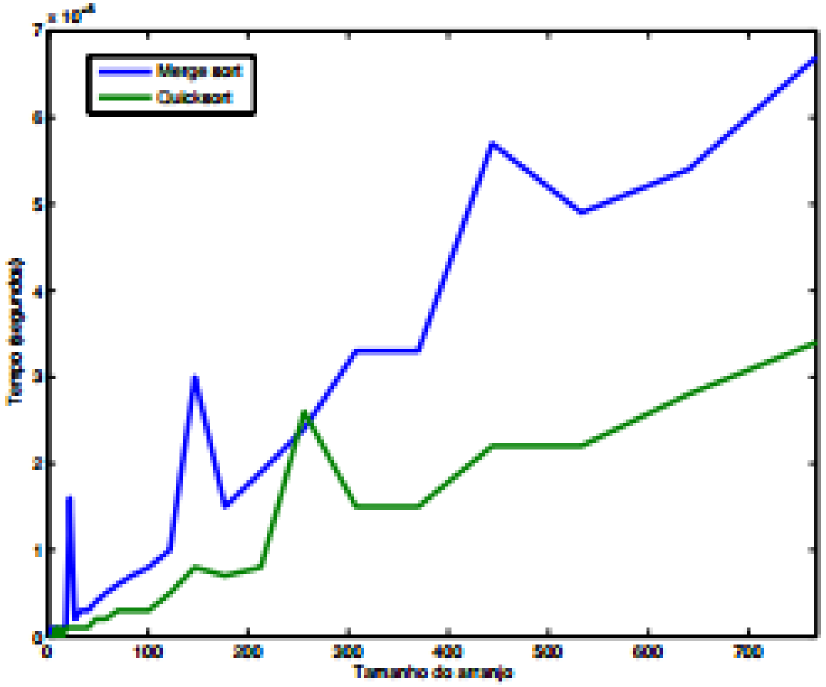
- Quanto maior o valor de n, piores serão os métodos O(n²) em relação aos métodos O(n log n)

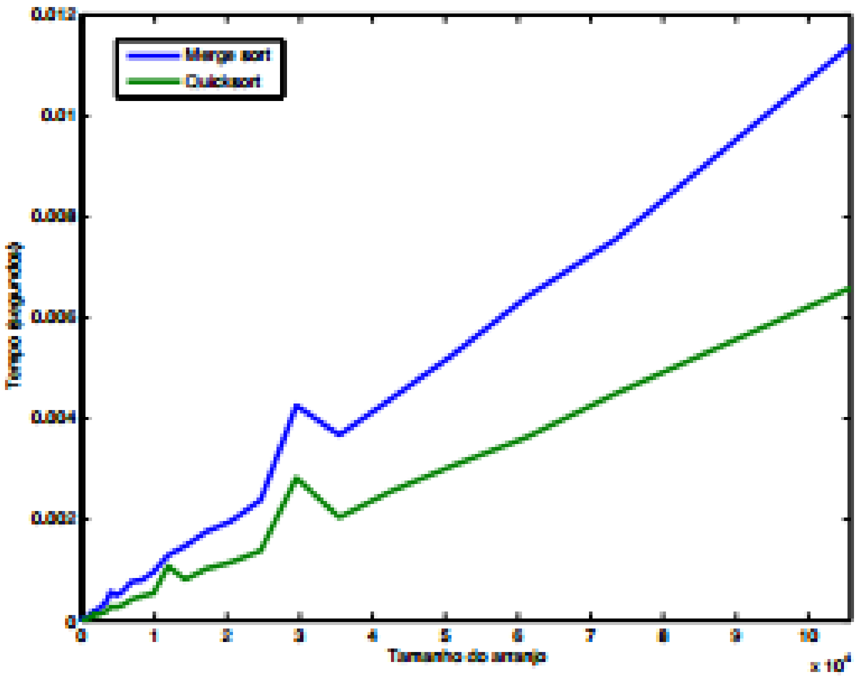




Testes Reais

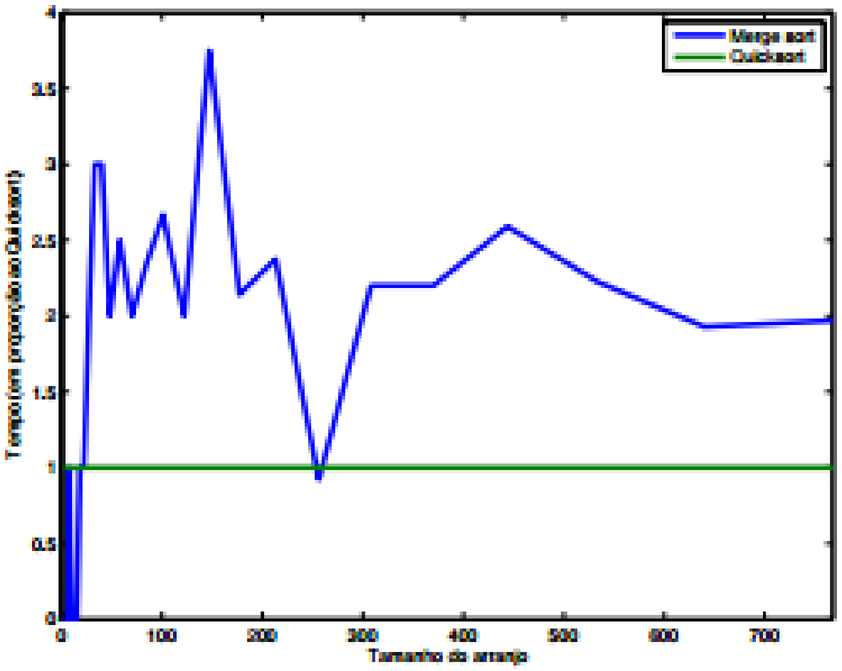
- Em comparação direta entre os métodos, o QuickSort é usualmente mais eficiente que o MergeSort

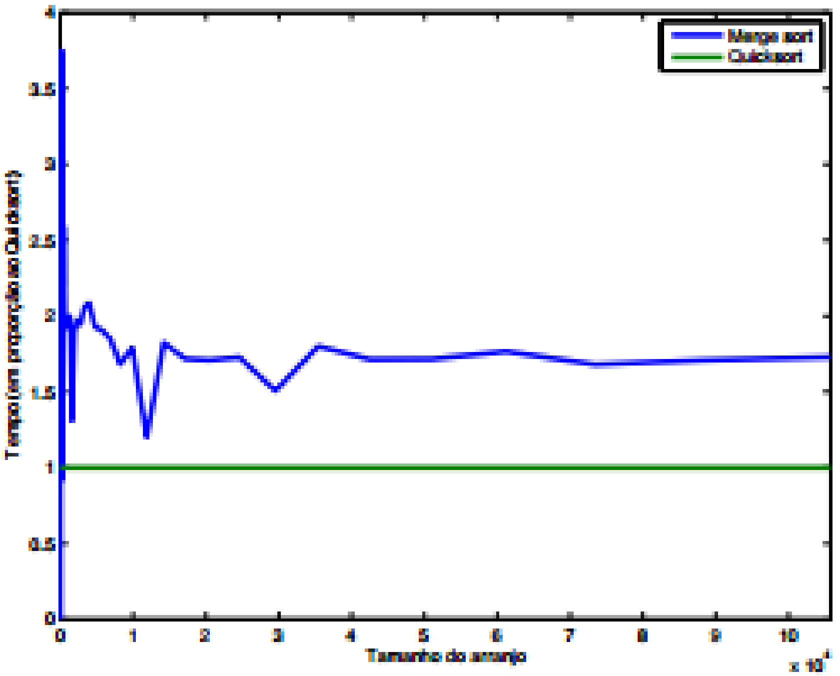




Testes Reais

- A proporção de eficiência entre os dois métodos se mantém constante para diferentes tamanhos de arranjos





Conclusão

- Os métodos de ordenação mostram como é possível obter diversificados tipos de vantagens e desvantagens entre algoritmos para a mesma classe de problemas, mesmo que este problema seja simples

- Mesmo com o limite inferior de O(n log n), há diversos fatores a se considerar em cada algoritmo como pior caso, caso médio, estabilidade, adaptabilidade, memória extra e operações de atribuição

Conclusão

- Apesar desta introdução ao tópico de ordenação ter sido limitada, as comparações entre algoritmos mostram a importância de conceitos como:
  - Notação O
  - Divisão e conquista
  - Estruturas de dados
  - Análise de melhor caso, pior caso e caso médio
  - Conflito entre tempo e memória

Conclusão

- Além dos métodos apresentados aqui, existem vários outros métodos de ordenação, cada um com suas vantagens e desvantagens, alguns dos mais famosos são:

  - Heapsort: melhora a ordenação por seleção com estruturas de dados mais eficientes

  - Introsort: combinação de QuickSort e Heapsort que não tem o pior caso $O(n^2)$

  - In-place MergeSort: MergeSort sem gasto extra de memória $O(n)$

Conclusão

- Buble sort: método simples de ordenação $O(n^2)$
- Shell sort: Generalização da ordenação por inserção
- Ordenação por contagem: o número de ocorrências de cada elemento é contado e recolocado no arranjo original
- Radix sort: ordenação com caso médio $O(nk)$, onde k é o número de dígitos dos elementos

Vídeos de apresentação dos algoritmos de ordenação Eficientes

- MergeSort
  - https://www.youtube.com/watch?v=XaqR3G_NVoo
- QuickSort
  - https://www.youtube.com/watch?v=ywWBy6J5gz8

Exercício

- Criar um arranjo com 10 elementos aleatórios não ordenados entre 1 e 50

- Desenhar o arranjo várias vezes demonstrando os passos de uma ordenação com os métodos:

  - QuickSort (particionar e mover a cada passo)

- Colocar um círculo nos elementos movimentados. Colocar um traço entre os elementos ordenados e desordenados

Algoritmos e Estruturas de Dados II

- Bibliografia:
  - Básica:

    - CORMEN, Thomas, RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
    - EDELWEISS, Nina, GALANTE, Renata. Estruturas de dados. Porto Alegre: Bookman. 2009. (Série livros didáticos informática UFRGS,18).
    - ZIVIANI, Nívio. Projeto de algoritmos com implementação em Pascal e C. São Paulo: Cengage Learning, 2010.

  - Complementar:

    - ASCENCIO, Ana C. G. Estrutura de dados. São Paulo: Pearson, 2011. ISBN: 9788576058816.
    - PINTO, W.S. Introdução ao desenvolvimento de algoritmos e estrutura de dados. São Paulo: Érica, 1990.
    - PREISS, Bruno. Estruturas de dados e algoritmos. Rio de Janeiro: Campus, 2000.
    - TENEMBAUM. Aaron M. Estruturas de dados usando C. São Paulo: Makron Books. 1995. 884 p. ISBN: 8534603480.
    - VELOSO, Paulo A. S. Complexidade de algoritmos: análise, projeto e métodos. Porto Alegre, RS: Sagra Luzzatto, 2001

Algoritmos e Estruturas de Dados II