Aula 13
Ordenação
Quicksort

Algoritmos e Estruturas de Dados

Quicksort

UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Quicksort

- Ideia:
  - Dividir para conquistar

    Dividir problema original em 2 subproblemas mais pequenos

  - Em vez de usar merge (como no mergesort)

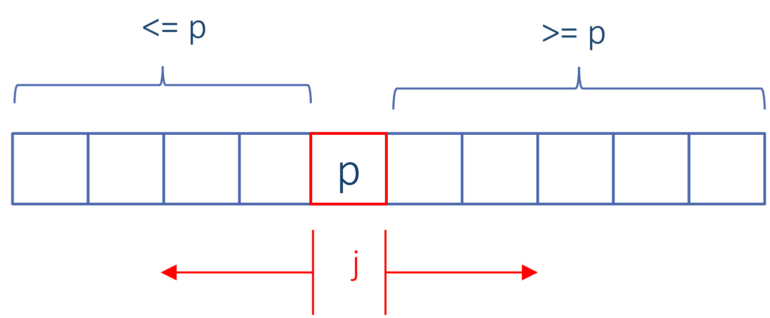
  - Usar partições (partitions)

Quicksort

- Em vez de ordenar de forma completa um *array* (ou *subarray*)
- Vai ordenar parcialmente um *array* (*subarray*) em relação a um pivô
  - À esquerda do pivô

    Elementos <= pivô
  - À direita do pivô

    Elementos >= pivô



método *partition*

Intuição para eficiência do algoritmo

- Estamos a converter o problema original
  - $O(n^2)$

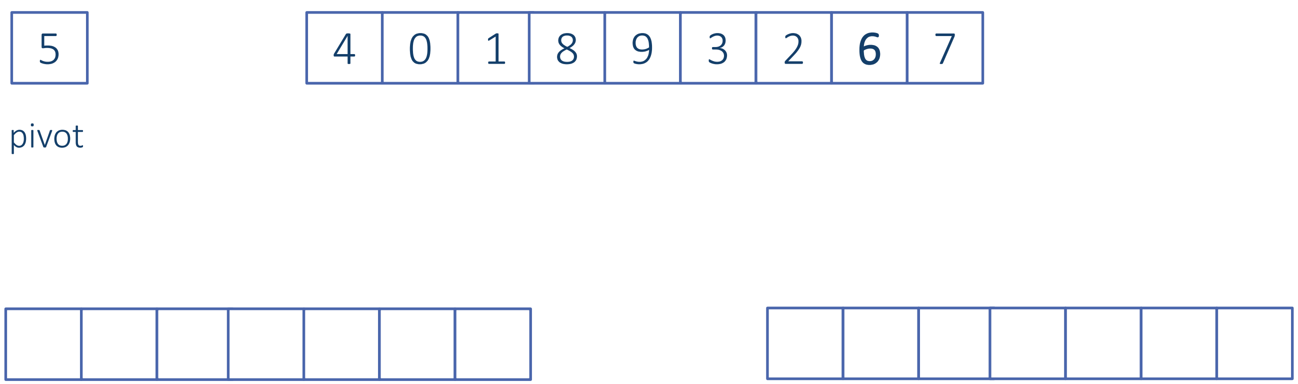
- Num problema equivalente que corresponde a fazer
  - $\log_2 n$ partições

Intuição para eficiência do algoritmo

- Estamos a converter o problema original
  - $O(n^2)$
- Num problema equivalente que corresponde a fazer
  - $\log_2 n$ partições
- Isto é bom desde que o custo de fazer uma partição seja
  - $O(n)$

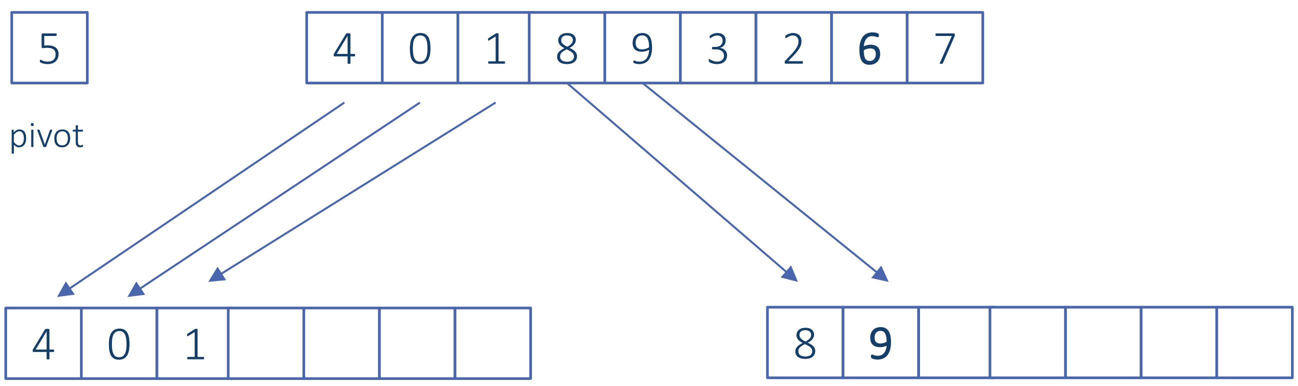
Qual o custo de fazer uma partição?

- Iremos ver em detalhe mais à frente, mas
  - Considerem que o espaço não é um problema, qual o custo de colocar os elementos < pivot num array esquerdo, e o colocar os elementos > pivot num array direito?



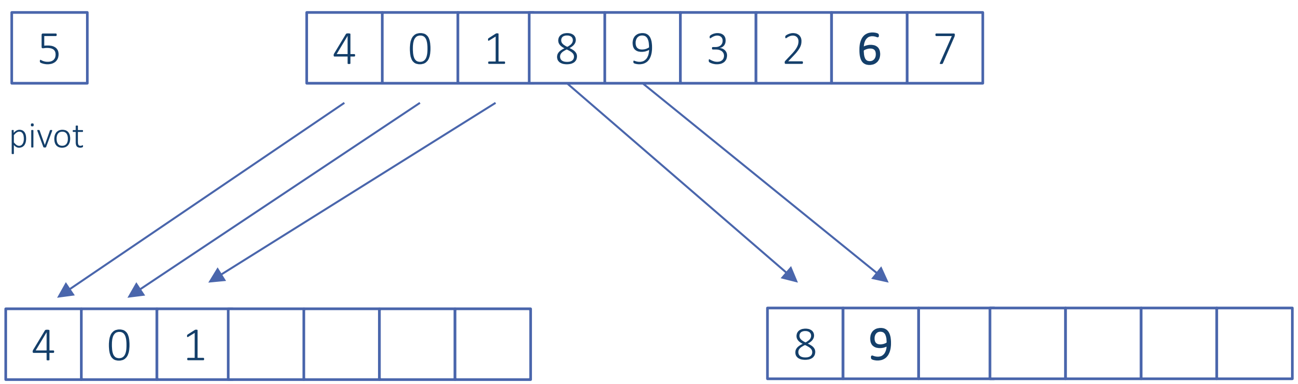
Qual o custo de fazer uma partição?

- Iremos ver em detalhe mais à frente, mas
  - Considerem que o espaço não é um problema, qual o custo de colocar os elementos < pivot num array esquerdo, e o colocar os elementos > pivot num array direito?



Qual o custo de fazer uma partição?

- Espero que não seja dificil de perceber que a complexidade temporal deste processo seja dada por
  - O(n)



Abordagem simplista de partições

- A abordagem vista no slide anterior não é aplicada pq
  - Requer memória a mais
  - Conseguimos fazer isto usando menos trocas
  - Mas funcionaria perfeitamente

    e seria teóricamente tão eficiente quanto o quicksort

- O algoritmo de partition que vamos ver é um pouco mais complexo

Propriedades método *partition*

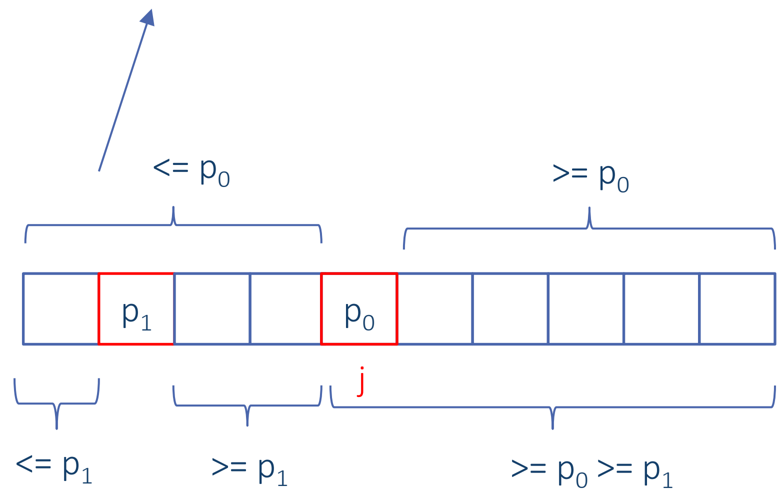
- No fim do método partition
  - O elemento pivô vai estar na posição correcta!
  - Já está ordenado.

- Portanto se fizermos *n partitions* num array de tamanho *n, o array* vai ficar completamente ordenado

- "Truque" do Quicksort
  - Partitions vão sendo cada vez mais simples
  - Ex: caso óptimo
  - 1º *partition*: compara n elementos
  - 2º, 3º *partition*: compara n/2 elementos
  - 4º,5º,6º,8º *partition*: compara n/4 elementos…

Para efectuar o método partition para p_1 apenas é necessário olhar para os elementos até j-1

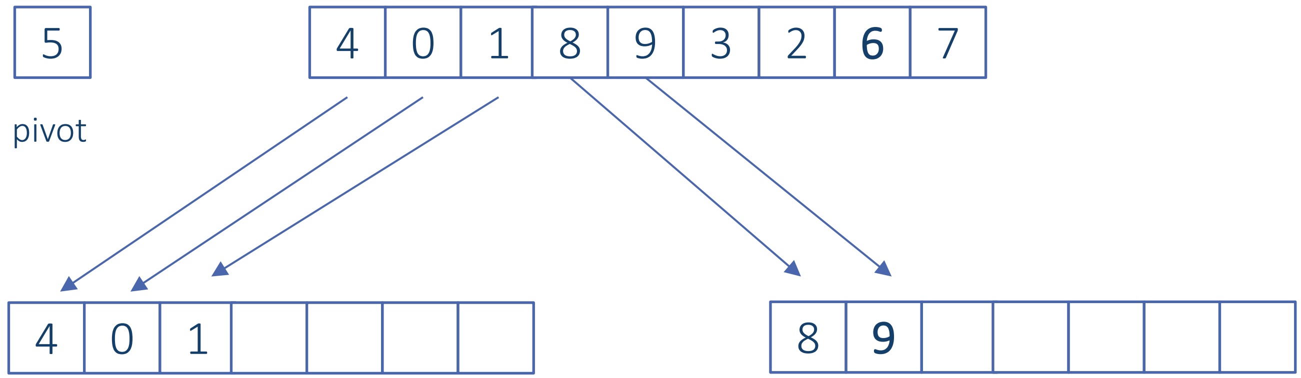
Como sabemos que p_1 <= p_0, logo todos os elementos a partir de j (inclusive) >= p_1, e satisfazem a ordenação parcial pretendida



método *partition*

Partition 1ª abordagem

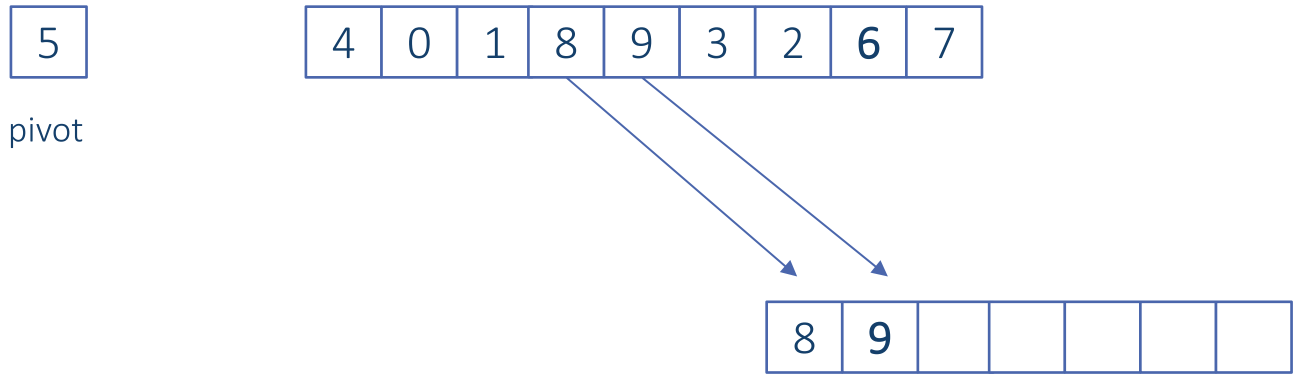
- Consideramos a existência de 2 subarrays onde podemos colocar os elementos < pivô e > pivô



- É possível implementar com um for simples da esq. para a dir.

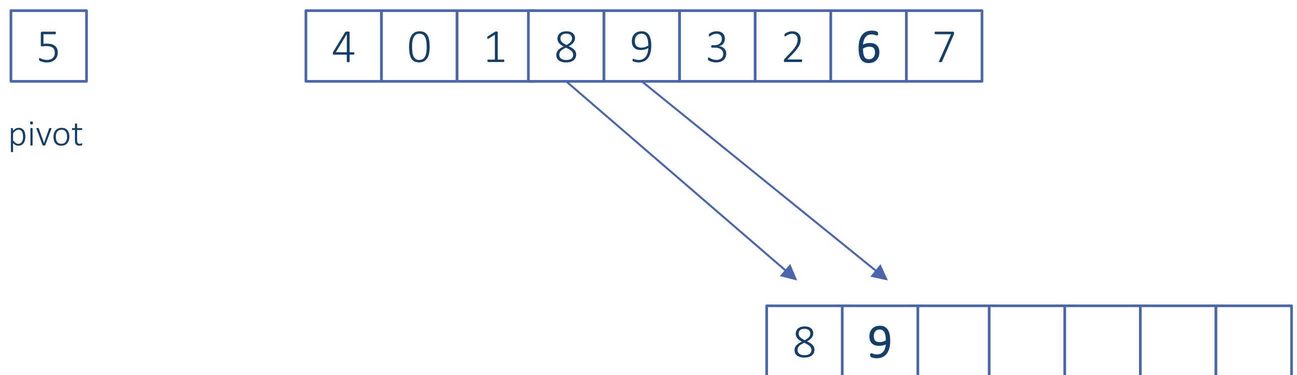
Partition 2ª abordagem

- Consideramos a existência de 1 subarray onde podemos colocar os elementos > pivô
- E **não nos importamos** de deixar espaços em branco



Partition 2ª abordagem

- Consideramos a existência de 1 subarray onde podemos colocar os elementos > pivô
- E **não nos importamos** de deixar espaços em branco



- for simples da esq. para a dir.

  if(a[i] > pivô) aux[j++] = a[i];

Partition 3ª abordagem

- Não podemos usar subarrays
- Usamos um ciclo apenas
  - Se a[i] < pivô, ignoramos e passamos para o próximo

5

pivot

4	0	1	8	?	?	?	**?**	?

i

?

- E se a[i] > pivô, o que fazer?

Partition 3ª abordagem

- Não podemos usar subarrays
- Usamos um ciclo apenas
  - Se a[i] < pivô, ignoramos e passamos para o próximo
  - Caso contrário, exchange(a,i--,j--)

5

pivot

4	0	1	8	?	?	?	**?**	7
			i					j

Partition 3ª abordagem

- Não podemos usar subarrays
- Usamos um ciclo apenas
  - Se a[i] < pivô, ignoramos e passamos para o próximo
  - Caso contrário, exchange(a,i--,j--)

5

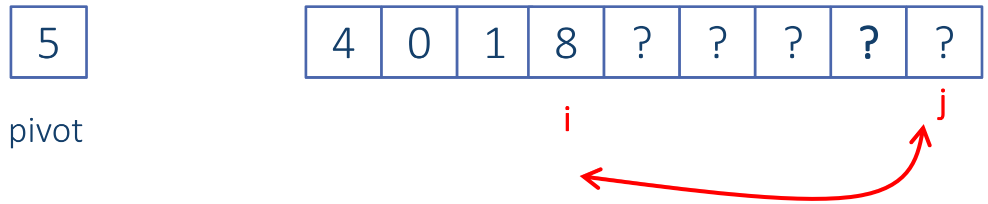
pivot

4	0	1	8	?	?	?	**?**	7
			i					j

- Esta abordagem já tem uma eficiência muito boa, mas ainda fazemos trocas a mais
  - O elemento 7 vai ser trocado de posição 2 vezes

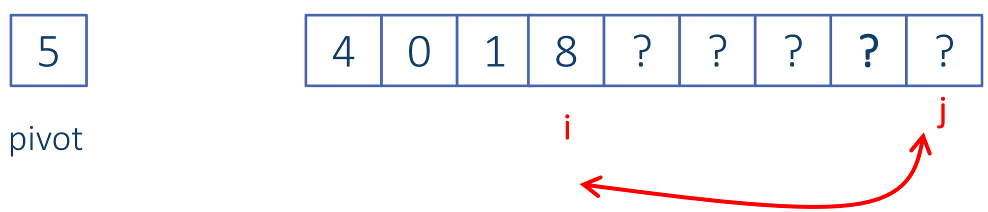
Partition 4ª abordagem

- Como otimizar as trocas de elementos?



Partition 4ª abordagem

- Como otimizar as trocas de elementos?



- Ideia: quando faço uma troca quero colocar 2 elementos no "lado" certo do array.

Partition 4ª abordagem

- Como otimizar as trocas de elementos?

5

pivot

4	0	1	8	?	?	2	**6**	7
			i			j		

- Consigo otimizar as trocas se tiver 2 ciclos que estão a tentar fazer algo semelhante
  - Um vai da esquerda para a direita à procura de elementos > pivo
  - O outro vai da direita para a esquerda à procura de elementos > pivo

Partition 4ª abordagem

- Como otimizar as trocas de elementos?

5

pivot

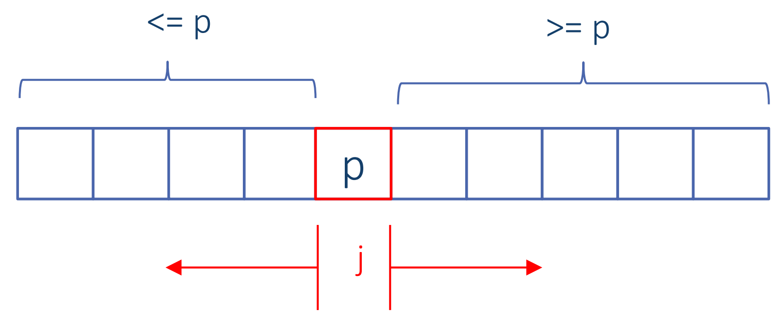
4	0	1	8	?	?	?	**?**	?

i

- Consigo otimizar as trocas se tiver 2 ciclos que estão a tentar fazer algo semelhante
  - Um vai da esquerda para a direita à procura de elementos > pivo
  - O outro vai da direita para a esquerda à procura de elementos > pivo

Quicksort partition

- A compreensão do método partition é crucial para percebermos o quicksort



método *partition*

- Metáfora útil
  - Separação de elementos (ex: ovelhas) de cores ou tamanhos distintos

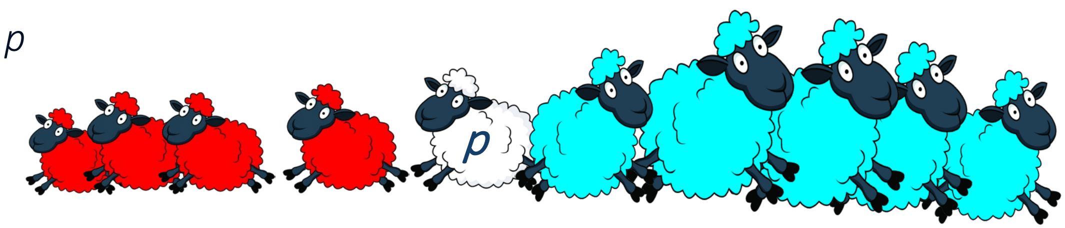
    Pequenas, cor vermelha: ovelhas <= p

    Grandes, cor azul: ovelhas >= p

  - Objectivo

    Colocar todas as ovelhas menores (vermelhas) do lado esquerdo

    Colocar todas as ovelhas maiores (azuis) do lado direito



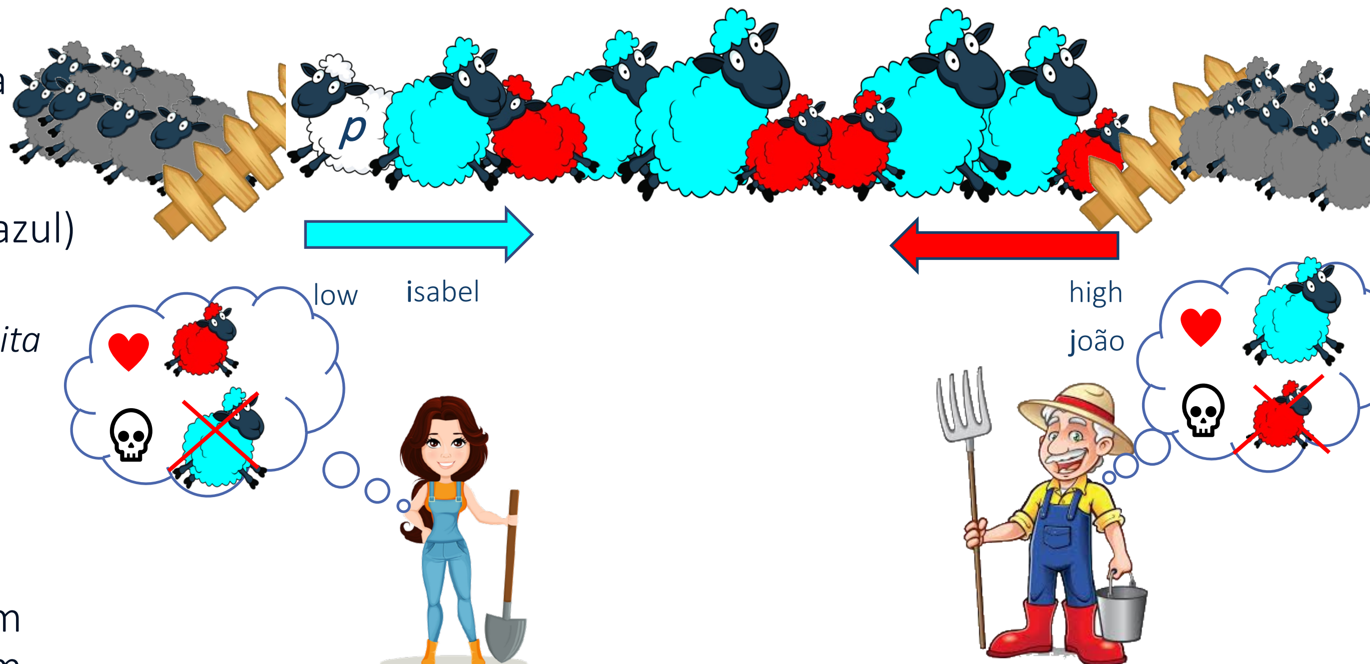
metáfora método *partition*

Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)



A **i**sabel só gosta de ovelhas vermelhas pequenas e fofinhas e não gosta das grandes azuis. Quer ficar apenas com ovelhas vermelhas do lado esquerdo e vai avançando da esquerda para a direita. Sempre que encontrar uma **azul** vai tentar trocá-la por uma vermelha com o **j**oão

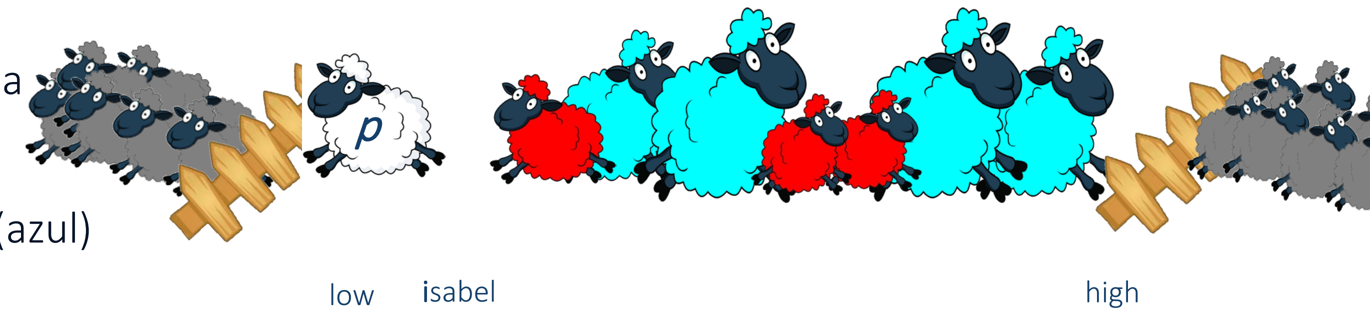
O **j**oão só gosta de ovelhas grandes e azuis, pois dão muita carne para o Jantar. Quer ficar apenas com ovelhas azuis do lado direito, e vai avançando da direita para a esquerda. Sempre que encontrar uma **vermelha** vai tentar trocá-la por uma azul com a **i**sabel

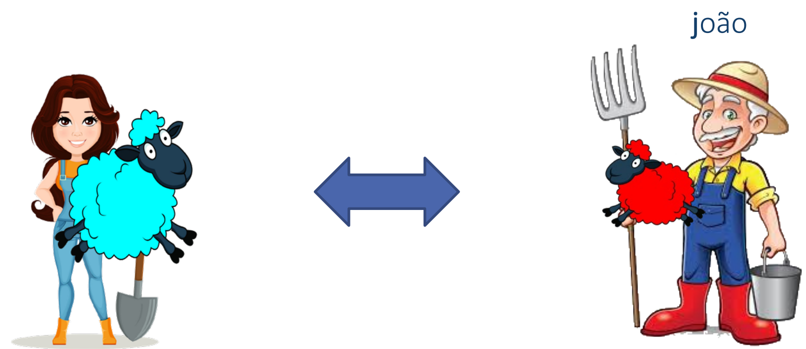
Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)





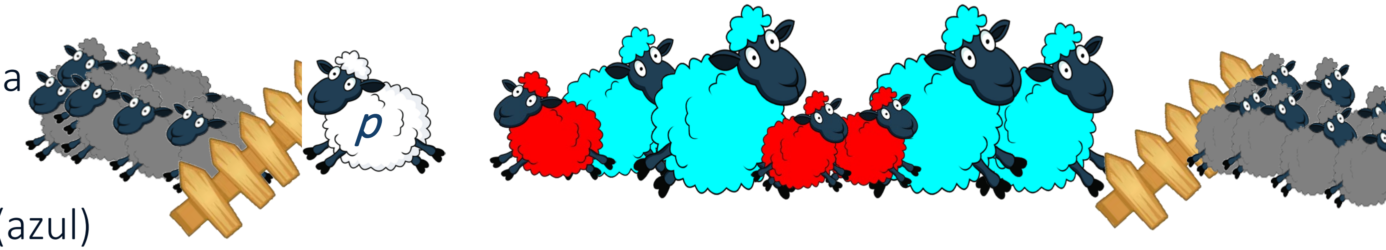
UALg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

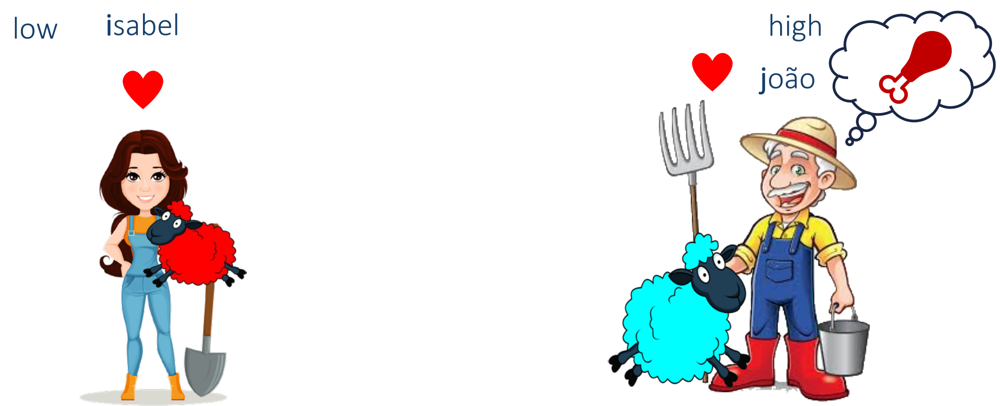
Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - i procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - j procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)





Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)

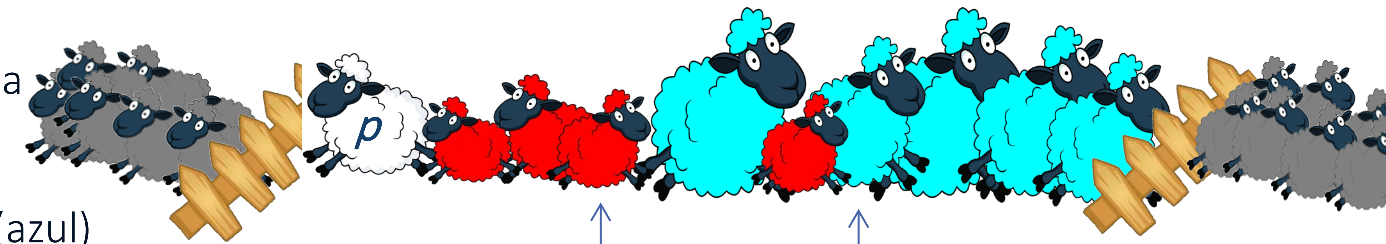
UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Método *partition*

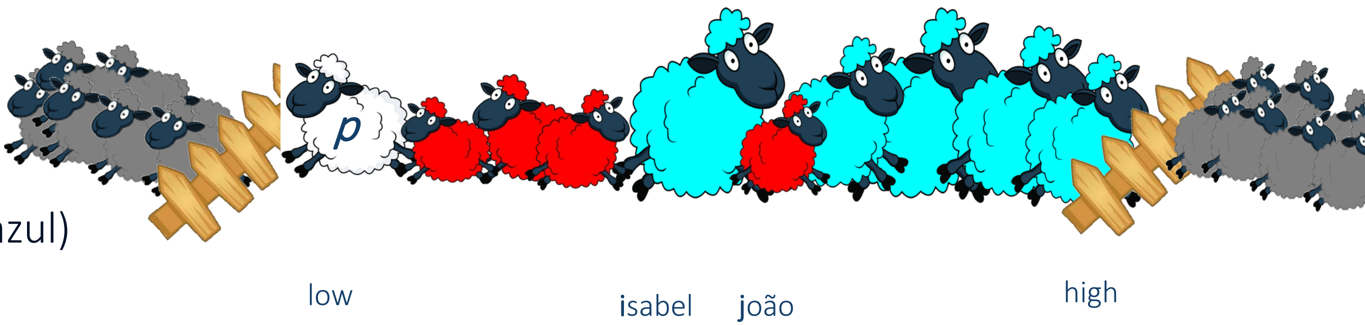
- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)



Método *partition*



- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

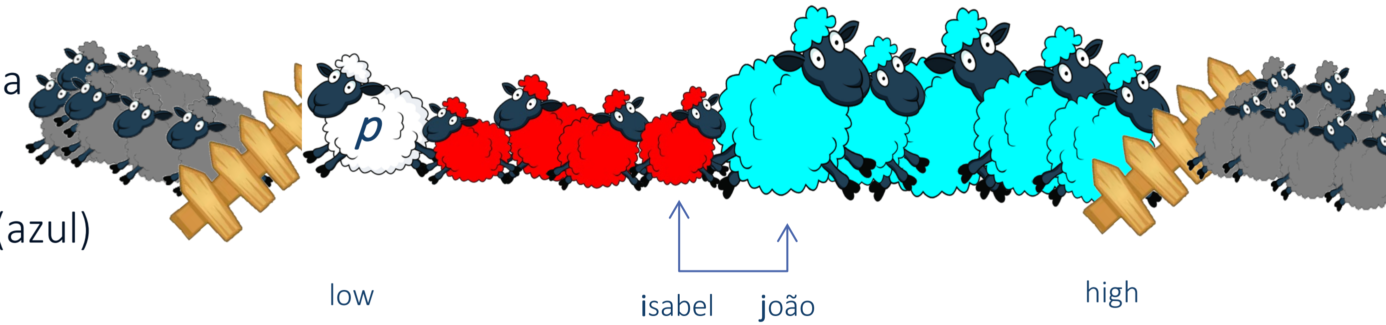
    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)

Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)

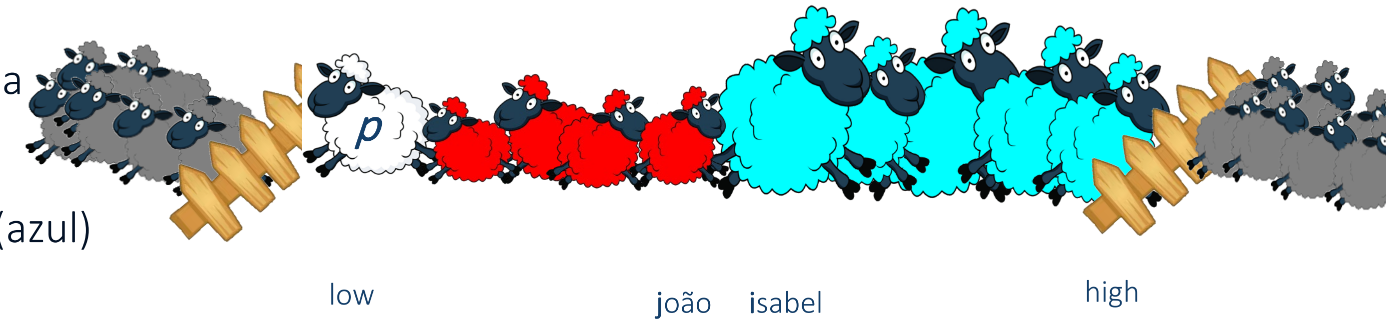


Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)



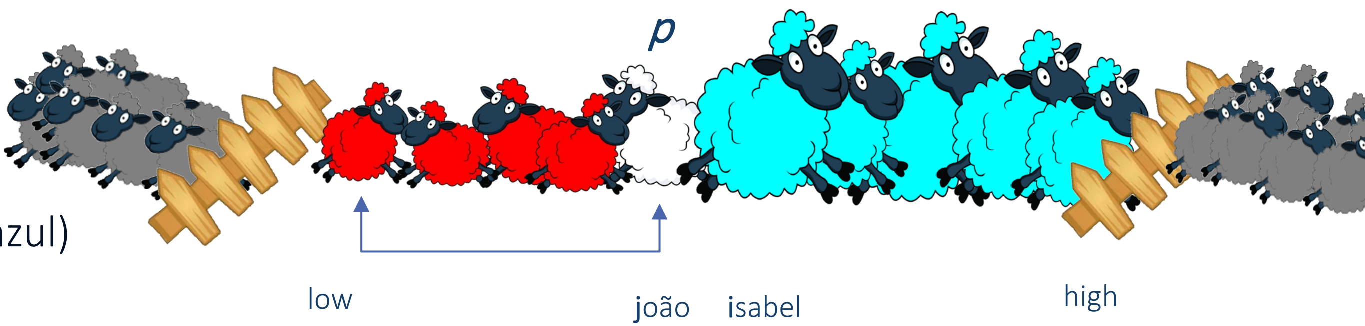
O processo pára quando:
O **j**oão e a **i**sabel se cruzam (o João passou para o lado da Isabel e vice-versa).

Método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha >p (azul) para trocar

    Pára se chegar à cerca direita
  - **j** procura uma ovelha <p (vermelha) para trocar

    Pára se chegar à cerca esquerda
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, pára o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)



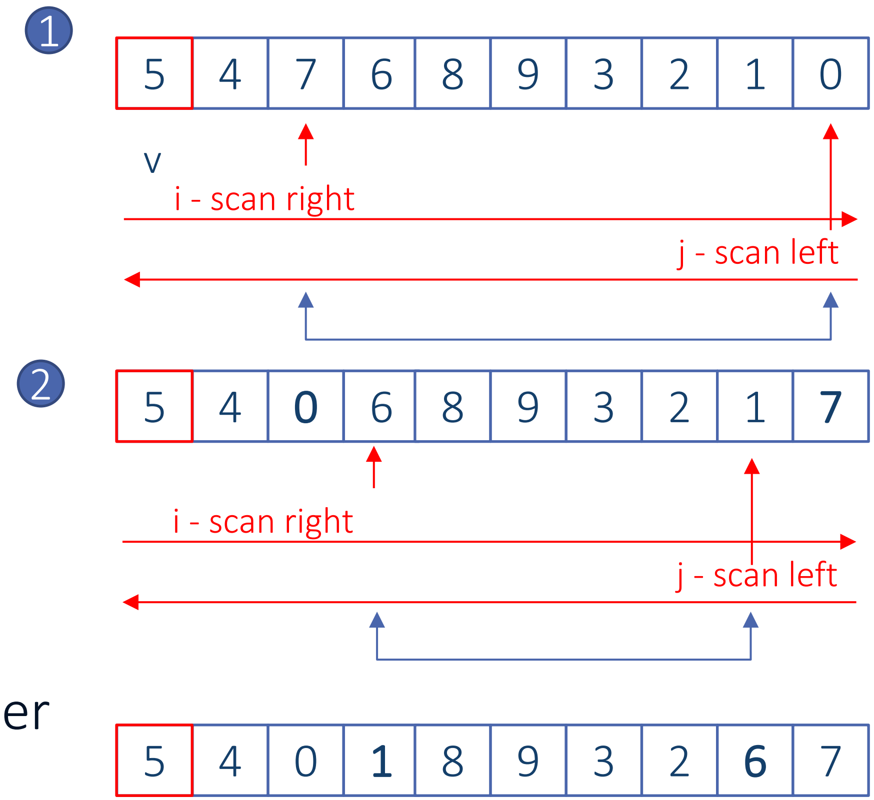
No final
Trocamos o pivô com a posição de j (joão)

Partir (partitioning)

- Escolher um elemento v - Pivô
- v = 1.º elemento array
- Repete:
  - Scan right (i)

    procura o 1.º elemento ≥ v
  - Scan left (j)

    procura o 1.º elemento ≤ v
  - Troca a[i] com a[j]
- Até não haver mais trocas a fazer
- No fim troca v com a[j]

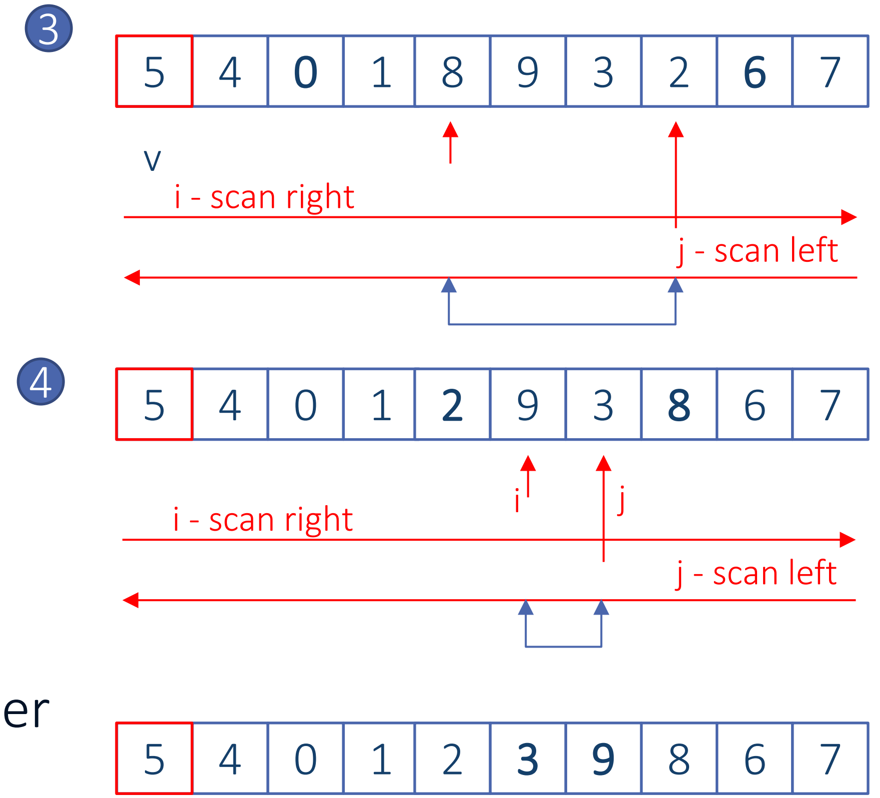


Partir (partitioning)

- Escolher um elemento v - Pivô
- v = 1.º elemento array
- Repete:
  - Scan right (i)

    procura o 1.º elemento ≥ v
  - Scan left (j)

    procura o 1.º elemento ≤ v
  - Troca a[i] com a[j]
- Até não haver mais trocas a fazer
- No fim troca v com a[j]

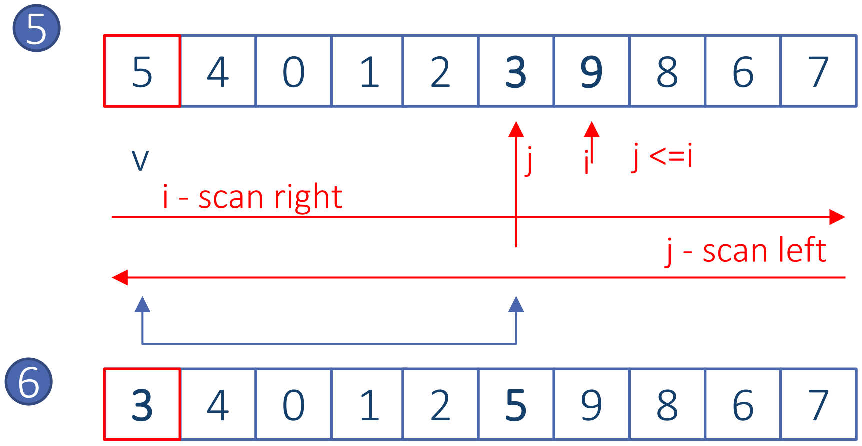


Partir (partitioning)

- Escolher um elemento v - Pivô
- v = 1.º elemento array
- Repete:
  - Scan right (i)

    procura o 1.º elemento ≥ v
  - Scan left (j)

    procura o 1.º elemento ≤ v
  - Troca a[i] com a[j]
- Até não haver mais trocas a fazer
  - Ex: j <= i
- No fim troca v com a[j]



Quicksort

```
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];
}
```

define os índices iniciais para scan-right (isabel) e scan-left (joão)

escolhe o pivô como sendo o valor em low

Quicksort

```
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];

    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;




    }
}
```

Scan right
Vai avançando i enquanto a[i] < v, ou seja para quando encontrar a[i] >= v

também para se i chegar ao fim do array

Quicksort

```
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];

    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;
        while(less(v,a[--j])) if(j == low) break;




}
```

também para se j chegar ao fim do array

Scan left

Vai avançando j enquanto a[j] > v, ou seja para quando encontrar a[j] <= v

Quicksort

```
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];

    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;
        while(less(v,a[--j])) if(j == low) break;

        if(i >= j) break;
```

Fim do scan

Se o i e o j se encontrarem, ou se se cruzarem já não há mais nada a trocar. Podemos terminar os scans e sair do *while* principal

```
}
```

Quicksort

```
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];

    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;
        while(less(v,a[--j])) if(j == low) break;

        if(i >= j) break;
        else exchange(a , i, j);
    }

}
```

Caso contrário, temos trocas a fazer. Troca o elemento na posição i com o elemento da posição j.
i tem um elemento que o scan-right não gosta, j tem um elemento que scan-left não gosta.
Voltamos a repetir o ciclo.

Quicksort

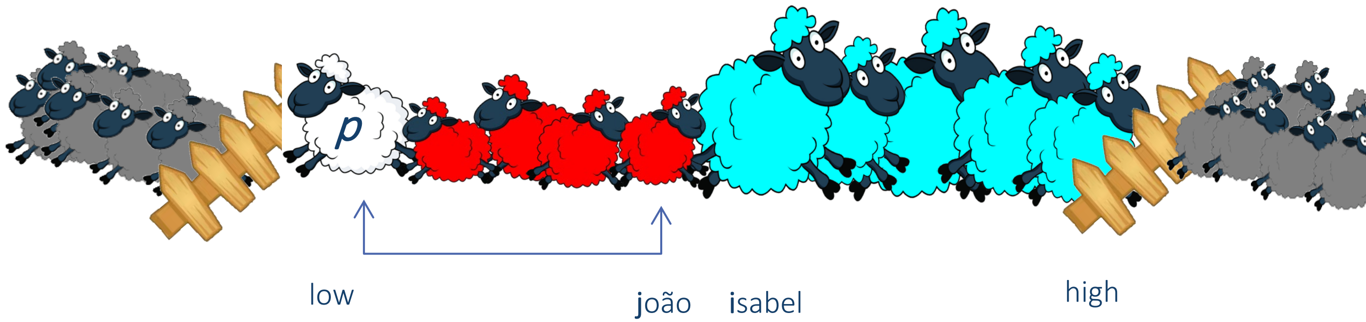
```java
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];

    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;
        while(less(v,a[--j])) if(j == low) break;

        if(i >= j) break;
        else exchange(a , i, j);
    }

    exchange(a, low, j);

    return j;
}
```

Depois de todas as trocas, colocamos o pivô (posição low) com o elemento que está na posição j.

E retornamos a posição onde ficou o pivô (j).

Pormenores método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha azul para trocar
  - **j** procura uma ovelha vermelha para trocar
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, para o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)



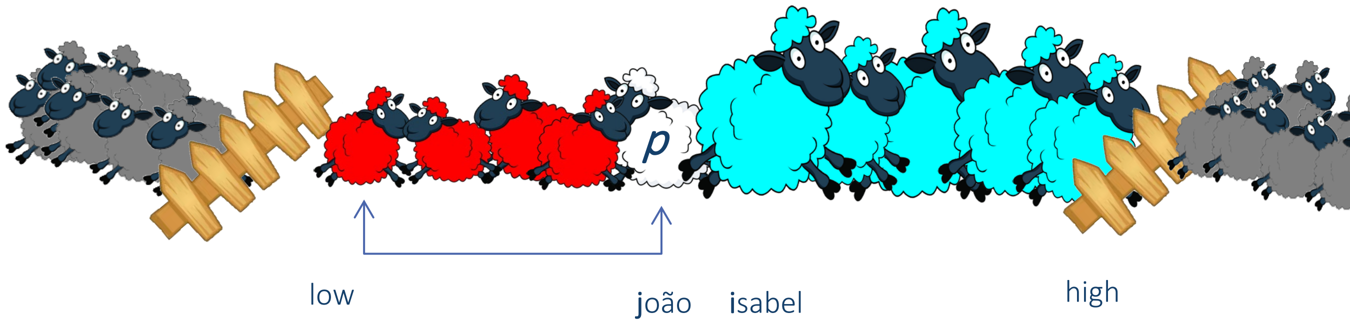
Pq é que usamos a posição do **j**oão como posição do pivô, e não a posição da **i**sabel?

R: Pq o pivô está do lado esquerdo, e portanto temos de o trocar com uma ovelha vermelha. Quando o **j**oão e a **i**sabel se cruzam, o **j**oão acabou de passar para a zona das ovelhas vermelhas (a **i**sabel acabou de passar para a zona das ovelhas azuis).

Mas reparem, se tivessemos escolhido (ou colocado) como pivô o mais à direita, teríamos de escolher a posição da **i**sabel como a posição final do pivô.

Pormenores método *partition*

- 1) escolher o pivô
  - Elemento mais à esquerda
- 2) Repetir
  - **i** procura uma ovelha azul para trocar
  - **j** procura uma ovelha vermelha para trocar
  - Se ambos i e j encontrarem ovelhas para trocar, trocam
  - Se i e j se cruzarem, para o ciclo
- 3) mover o pivô para a posição j (vai ser a posição final)

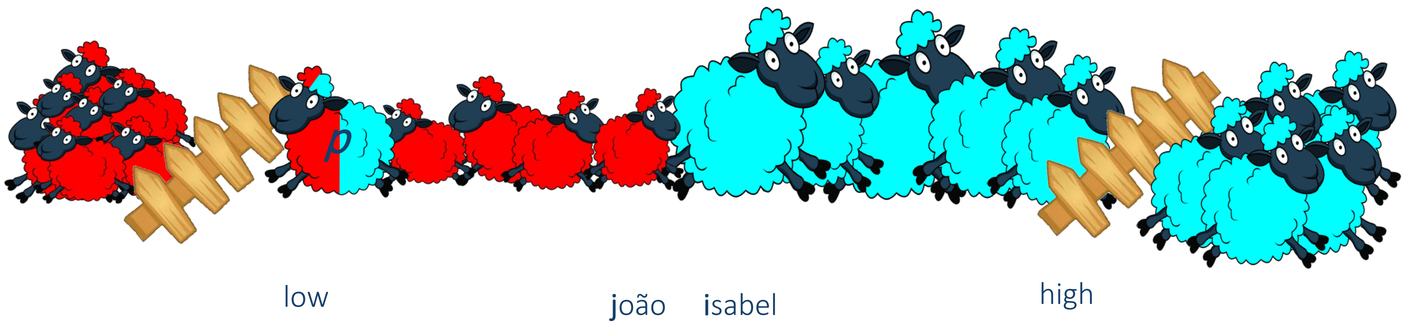


Pq é que usamos a posição do **j**oão como posição do pivô, e não a posição da **i**sabel?

R: Pq o pivô está do lado esquerdo, e portanto temos de o trocar com uma ovelha vermelha. Quando o **j**oão e a **i**sabel se cruzam, o **j**oão acabou de passar para a zona das ovelhas vermelhas (a **i**sabel acabou de passar para a zona das ovelhas azuis).

Mas reparem, se tivessemos escolhido (ou colocado) como pivô o mais à direita, teríamos de escolher a posição da **i**sabel como a posição final do pivô.

Pormenores método *partition*



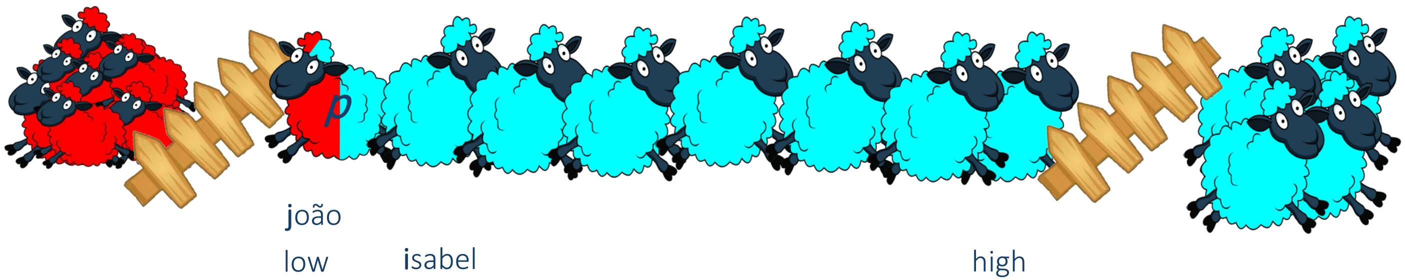
O pivô na realidade pode ser considerado quer como azul quer como vermelho.

A **i**sabel vê-o como azul (pois pode ser trocado com o joão), o **j**oão vê-o como vermelho (pois pode ser trocado com a Isabel).

Isto aplica-se a qualquer elemento = pivô

Elementos à esquerda da cerca esquerda (low) são todos vermelhos (e são ignorados)
Elementos à direita da cerca direita (high) são todos azuis (e são ignorados)

Pormenores método *partition*

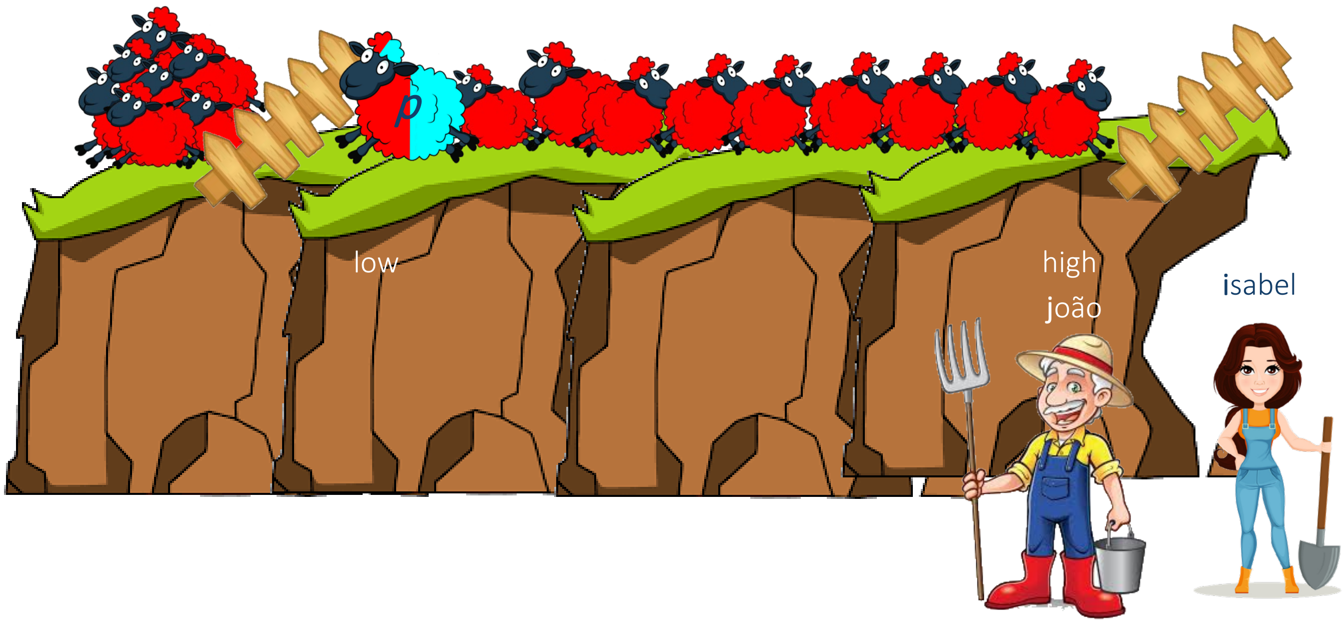


```
while(true)
 {
      while(less(a[++i],v)) if(i == high) break;
      while(less(v,a[--j])) if(j == low) break;

      if(i >= j) break;
      …
 }
```

De acordo com o livro, o teste à sentinela do scan left (joão) pode ser removido. Porquê?

Mesmo que não existam elementos vermelhos, o joão irá sempre parar obrigatóriamente no pivô, e ao fazê-lo vai aperceber-se que cruzou a Isabel, terminando o ciclo.

Pormenores método *partition*



Já a Isabel necessita do teste à sentinela. Porquê?

No caso acima, sem o teste da sentinela, a Isabel iria continuar a avançar para a direita, ultrapassando a cerca direita. E podemos ter o azar de que à direita esteja um precipicio pois o array acaba na cerca direita.

Quicksort - recursão

- Estratégia de recusão
- Partir o array
  - Vai dividir o array em 2

    Esquerda de j

    Direita de j
- Partir recursivamente
  - Esquerda de j
  - Direita de j

Quicksort - recursão

```
public static void sort(Comparable[] a)
{
    sort(a, 0, a.length-1);
}

private static void sort(Comparable[] a, int low, int high)
{
    if (high <= low) return;
    int j = partition(a, low, high);
    sort(a, low, j-1);
    sort(a, j+1, high);
}
```

Complexidade temporal

```java
private static int partition(Comparable[] a, int low, int high)
{
    //partition into a[low...j-1],a[j],[aj+1...high]
    //and return j
    int i = low, j = high +1;
    Comparable v = a[low];
    while(true)
    {
        while(less(a[++i],v)) if(i == high) break;
        while(less(v,a[--j])) if(j == low) break;

        if(i >= j) break;
        exchange(a , i, j);
    }
    exchange(a, low, j);

    return j;
}
```

Método partition

$T_{partition}(n)$=

- n+1 comparações no pior caso
- n comparações no melhor caso
- n/6 trocas em média

Complexidade temporal

- Melhor caso
  - Partir divide sempre array ao meio
  - T_{qsort} (n)

$= T_{partition}(n) + T_{qsort}(n/2) + T_{qsort}(n/2)$

$= 2\ T_{qsort}(n/2) + n$

$= 2(2\ T_{qsort}(n/4) + T_{partition}(n/2)) + n$

$= 4T_{qsort}(n/4) + n + n$

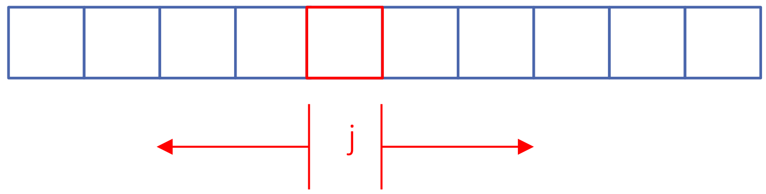
$= 8T_{qsort}(n/8) + n + n + n$

…

$= \underbrace{n + \ldots + n}_{\log 2\ n}$

$$= n \log_2 n$$

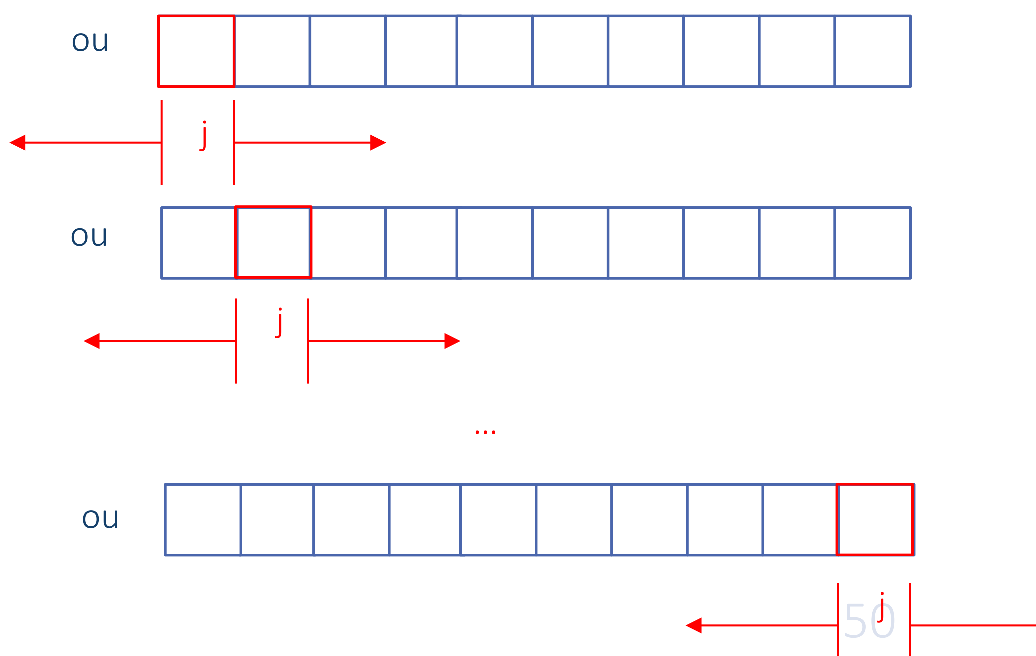
```java
private static void sort(Comparable[] a, int low, int high)
{
    if (high <= low) return;
    int j = partition(a, low, high);
    sort(a, low, j-1);
    sort(a, j+1, high);
}
```



Complexidade temporal

- Caso médio
  - Análise Quicksort é + complexa
  - $T_{qsort}(n) = T_{partition}(n) + T_{esq} + T_{dir}$
  - Existem n formas distintas de separar o array
  - Assumindo igual probabilidade 1/n
  - $T_{sort}(n) = (n+1) + \left[\frac{T_{sort}(0)+T_{sort}(n-1)}{n} + \frac{T_{sort}(1)+T_{sort}(n-2)}{n} + \cdots + \frac{T_{sort}(n-1)+T_{sort}(0)}{n}\right]$

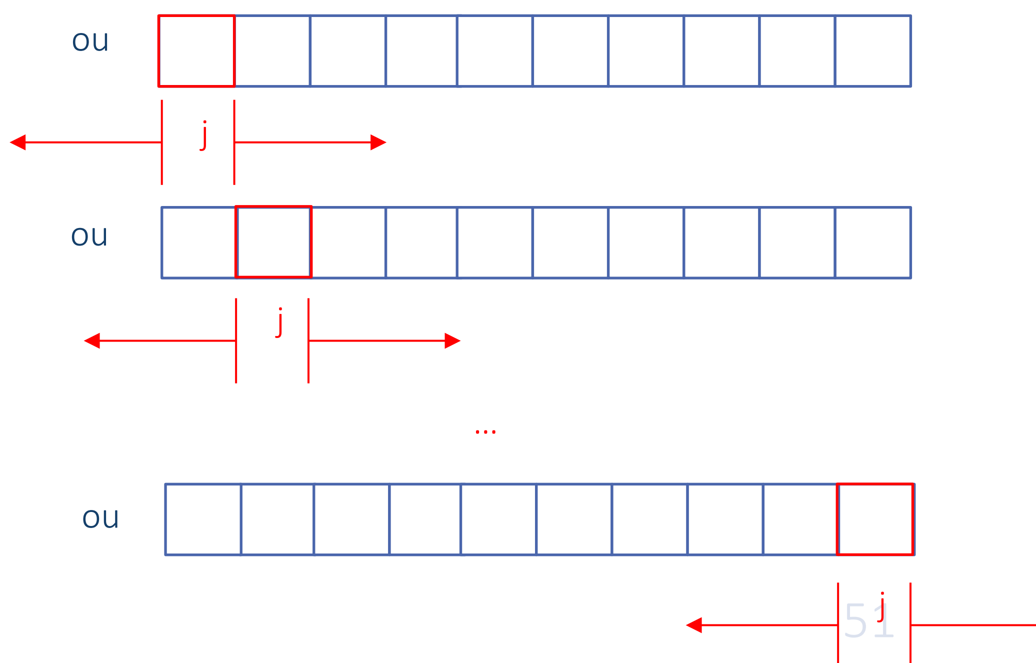
```java
private static void sort(Comparable[] a, int
low, int high)
{
    if (high <= low) return;
    int j = partition(a, low, high);
    sort(a, low, j-1);
    sort(a, j+1, high);
}
```



Complexidade temporal

```java
private static void sort(Comparable[] a, int
low, int high)
{
    if (high <= low) return;
    int j = partition(a, low, high);
    sort(a, low, j-1);
    sort(a, j+1, high);
}
```

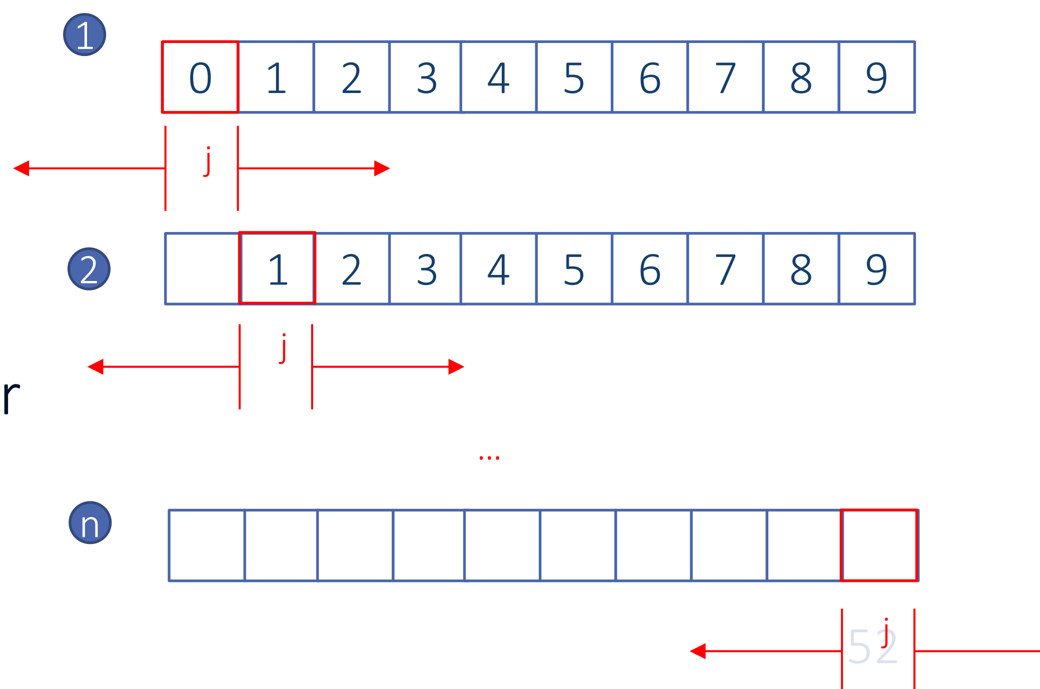
- Caso médio
  - T_{sort} (n) = $(n + 1) + 2\left[\frac{T_{sort}(0)+T_{sort}(1)+\cdots+T_{sort}(n-2)+T_{sort}(n-1)}{n}\right]$
  - …
  - $T_{sort}(n) = 2 + (1 + \frac{1}{n})T_{sort}(n-1)$
  - $T_0 = T_1 = 0$
  - $T_n \sim 2n \ln n$
  - $T_n \sim 1.39\ n \log_2 n$



Complexidade temporal

```
while(true)
{
    while(less(a[++i],v)) if(i == high) break;
    while(less(v,a[--j])) if(j == low) break;

    if(i >= j) break;
    exchange(a , i, j);
}
exchange(a, low, j);
return j;
```

- Pior caso
  - A partição é feita sempre numa das pontas do array
  - Ex: O array já se encontra ordenado
  - n partições
  - $T_{sort}(n) = n + (n-1) + (n-2) + \ldots + 1$
  - $T_{sort(n)} = \dfrac{n(n+1)}{2} \sim \dfrac{n^2}{2}$
  - Felizmente é possível evitar o pior caso



Complexidade temporal

```java
private static void sort(Comparable[] a, int
low, int high)
{
    if (high <= low) return;
    int j = partition(a, low, high);
    sort(a, low, j-1);
    sort(a, j+1, high);
}
```

```java
private static int partition …
…
while(true)
{
    while(less(a[++i],v)) if(i == high) break;
    while(less(v,a[--j])) if(j == low) break;

    if(i >= j) break;
    exchange(a , i, j);
}
exchange(a, low, j);
```

Quicksort	Melhor caso	Pior caso	Aleatório	O
less/compare	$n \log_2 n$	$\sim n^2/2$	$\sim 2\, n \ln n$ $\sim 1.39\, n \log_2 n$	$O(n \log_2 n)$
exchange	$\sim 1/6\, n \log_2 n$	n	$\sim 1/3\, n \ln n$ $\sim 0.21\, n \log_2 n$	

é muito fácil resolver este problema, por isso vamos ignorar este pior caso

Quicksort – Pior caso

- Felizmente é muito fácil evitar que o pior caso aconteça
  - A) reordenar o array de forma aleatória, antes de começar o algoritmo
  - Ou
  - B) no início de cada partição escolher o pivô v de forma aleatória
  - A ordem de crescimento do pior caso torna-se igual ao caso aleatório.

Truques para melhorar *Quicksort*

- 1) usar insertion sort para arrays pequenos
  - A partir de um valor de cutoff (ex: 10) usar *insertionsort* em vez de *quicksort*

- *2) partição com mediana de 3*
  - *Quando se começa uma nova partição*

    Escolher 3 elementos

    Usar o valor do meio como pivô

Características Quicksort

- Tem um calcanhar de aquiles
- A performance depende da qualidade do pivô escolhido
  - Se o array estiver ordenado, eficiência é mt má
  - Easy fix: baralhar o array (ou escolher pivô aleatóriamente)

- Embora assimptóticamente seja equivalente ao mergesort
  - Na prática é mais eficiente
  - Caso aleatório

    40% mais comparações

    10x menos trocas

Quicksort	Melhor caso	Pior caso	Aleatório	O
less/compare	$n \log_2 n$	$\sim n^2/2$	$\sim 1.39\, n \log_2 n$	$O(n \log_2 n)$
exchange	$\sim 1/6\, n \log_2 n$	n	$\sim 0.21\, n \log_2 n$	