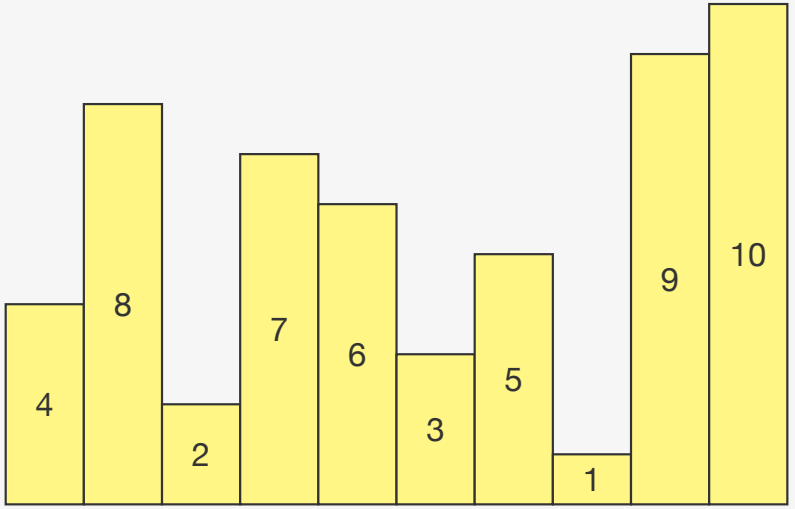
MC-202 — Unidade 12
Quicksort e Particionamento

Rafael C. S. Schouery
rafael@ic.unicamp.br

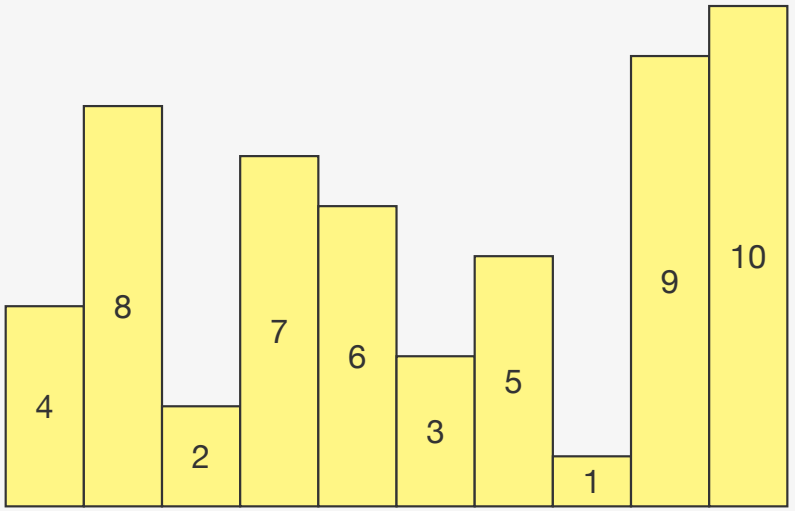
Universidade Estadual de Campinas

2º semestre/2017

Quicksort - Ideia

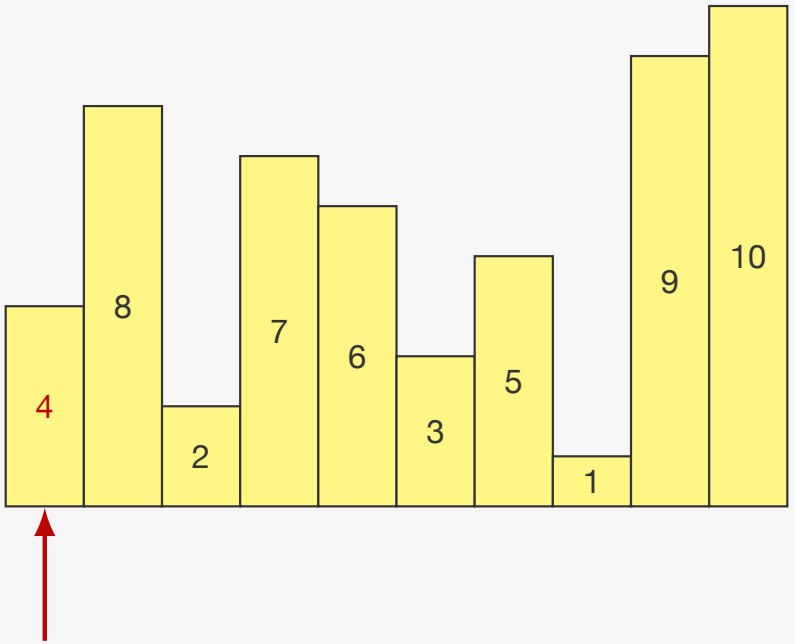


Quicksort - Ideia



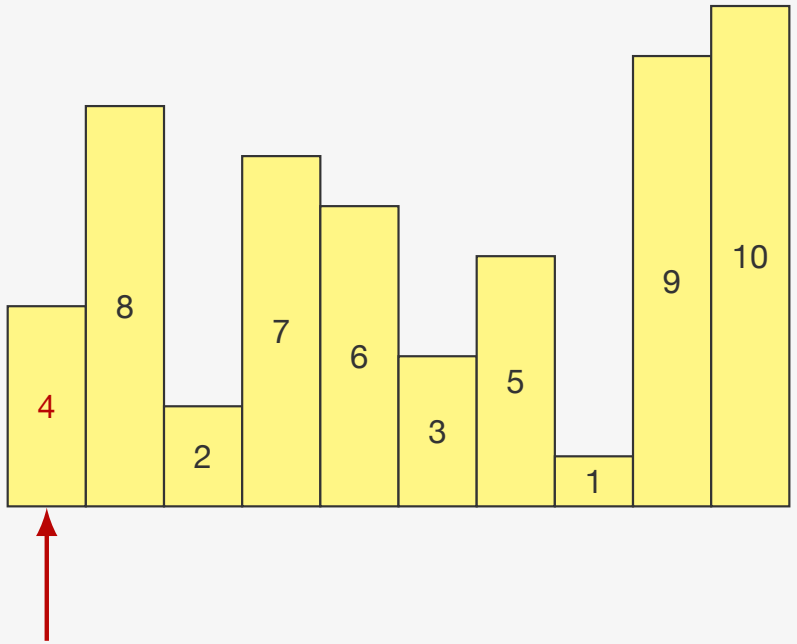
- Escolhemos um pivô (ex: 4)

Quicksort - Ideia



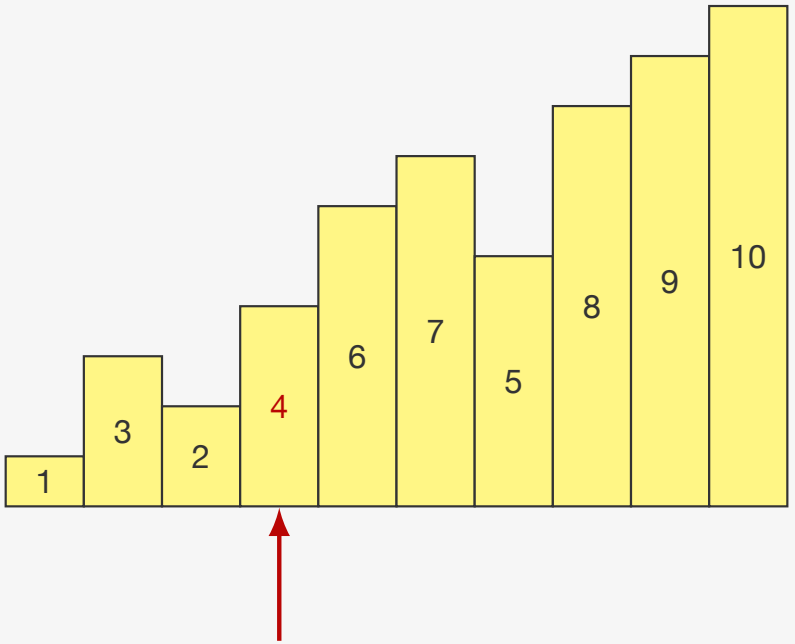
- Escolhemos um pivô (ex: 4)

Quicksort - Ideia



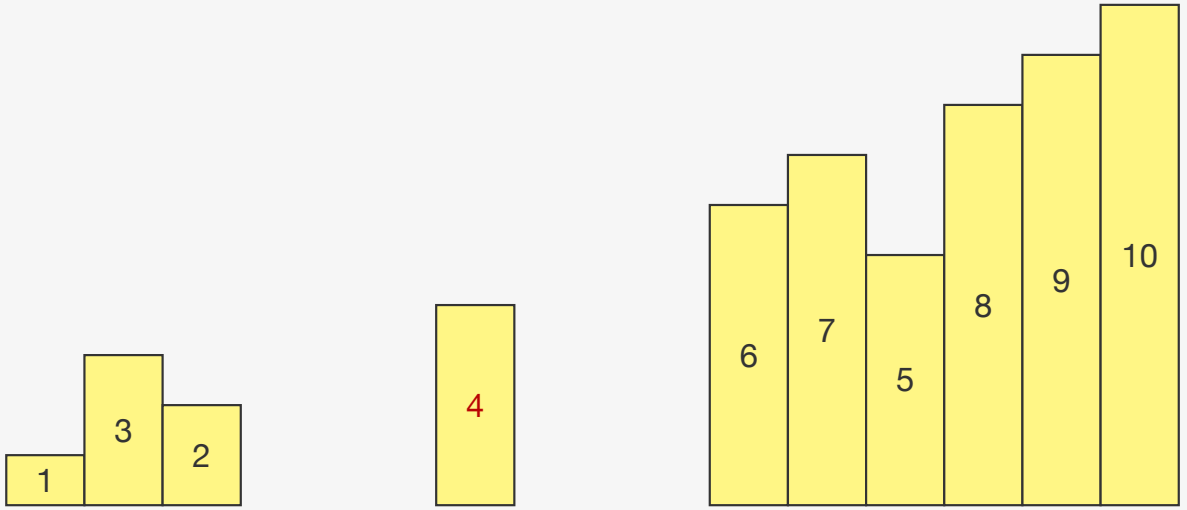
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô na esquerda
- e os elementos maiores que o pivô na direita

Quicksort - Ideia



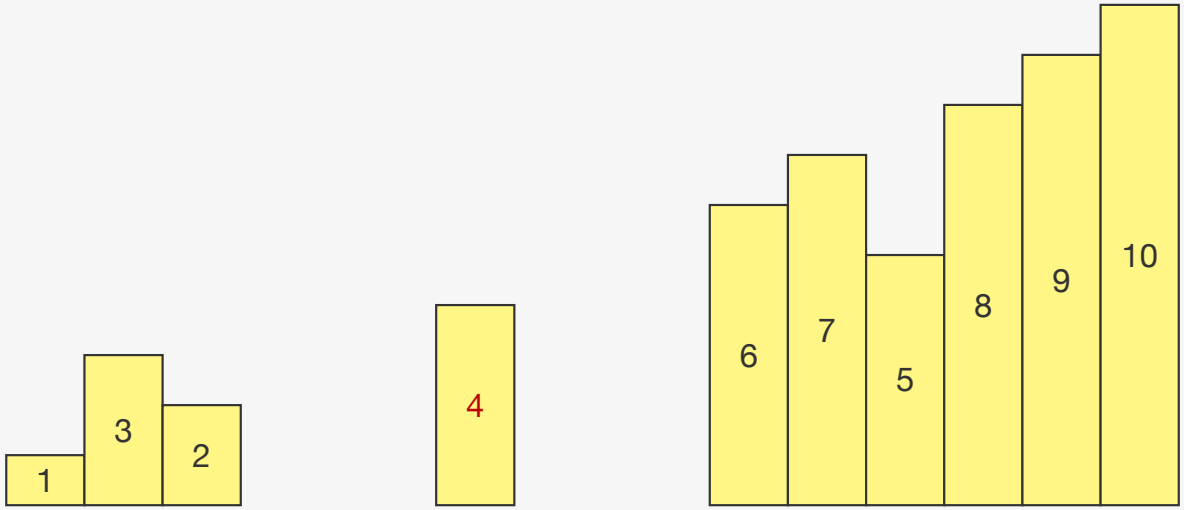
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô na esquerda
- e os elementos maiores que o pivô na direita

Quicksort - Ideia



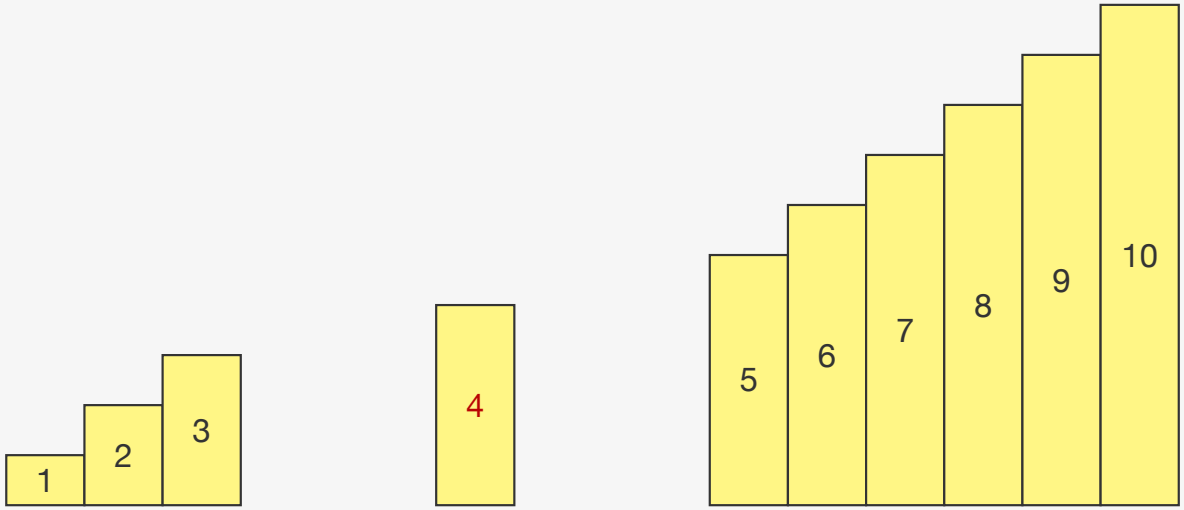
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô na esquerda
- e os elementos maiores que o pivô na direita
- O pivô está na posição correta

Quicksort - Ideia



- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô na esquerda
- e os elementos maiores que o pivô na direita
- O pivô está na posição correta
- O lado esquerdo e o direito podem ser ordenados independentemente

Quicksort - Ideia



- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô na esquerda
- e os elementos maiores que o pivô na direita
- O pivô está na posição correta
- O lado esquerdo e o direito podem ser ordenados independentemente

Quicksort

```
1 int partition(int *v, int l, int r);
```

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```
1 void quicksort(int *v, int l, int r) {
2   int i;
3   if(r <= l) return;
4   i = partition(v, l, r);
5   quicksort(v, l, i-1);
6   quicksort(v, i+1, r);
7 }
```

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```
1 void quicksort(int *v, int l, int r) {
2   int i;
3   if(r <= l) return;
4   i = partition(v, l, r);
5   quicksort(v, l, i-1);
6   quicksort(v, i+1, r);
7 }
```

- Basta particionar o vetor em dois

Quicksort

```
1 int partition(int *v, int l, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```
1 void quicksort(int *v, int l, int r) {
2   int i;
3   if(r <= l) return;
4   i = partition(v, l, r);
5   quicksort(v, l, i-1);
6   quicksort(v, i+1, r);
7 }
```

- Basta particionar o vetor em dois
- e ordenar o lado esquerdo e o direito

Como particionar um vetor?

Ideia:

Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:

Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do primeiro elemento

Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do primeiro elemento
2. Procuramos elementos fora de ordem:

Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do primeiro elemento
2. Procuramos elementos fora de ordem:
   - do fim ao início: em busca de valores menores que o pivô

Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do primeiro elemento
2. Procuramos elementos fora de ordem:
   - do fim ao início: em busca de valores menores que o pivô
   - do início ao fim: em busca de valores maiores que o pivô

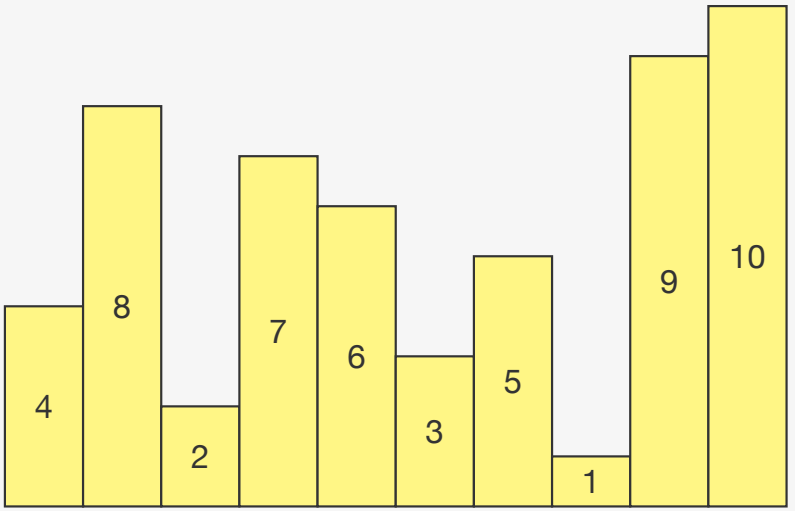
Como particionar um vetor?

Ideia:

1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do primeiro elemento
2. Procuramos elementos fora de ordem:
   - do fim ao início: em busca de valores menores que o pivô
   - do início ao fim: em busca de valores maiores que o pivô
3. Corrigimos os elementos que estão em posições erradas

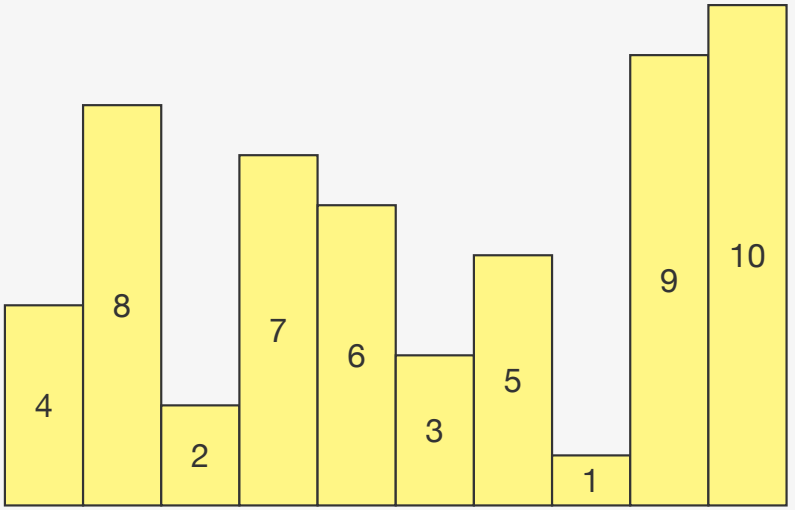
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



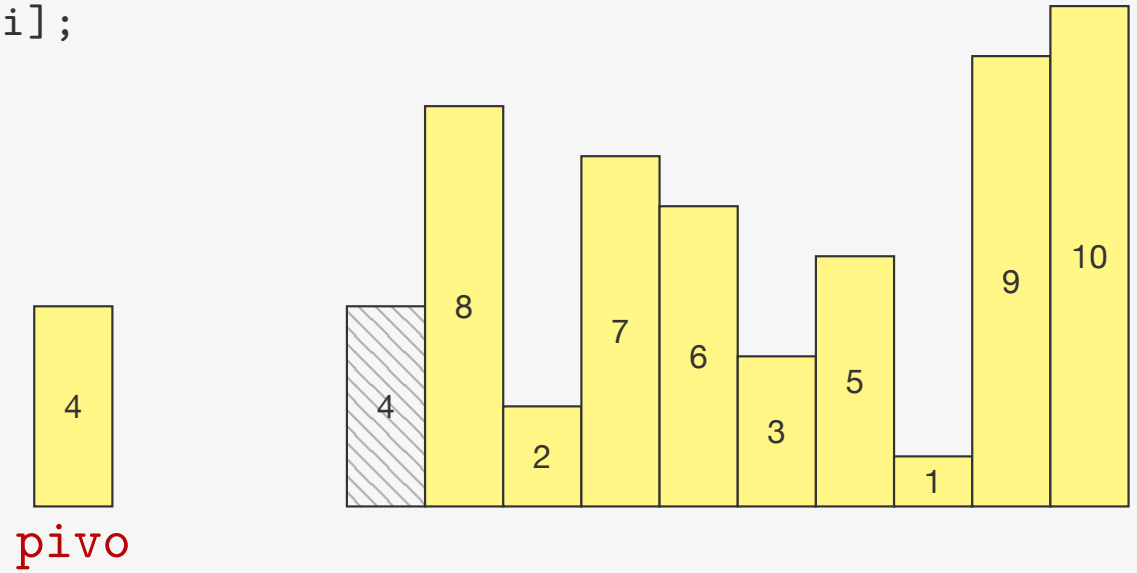
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



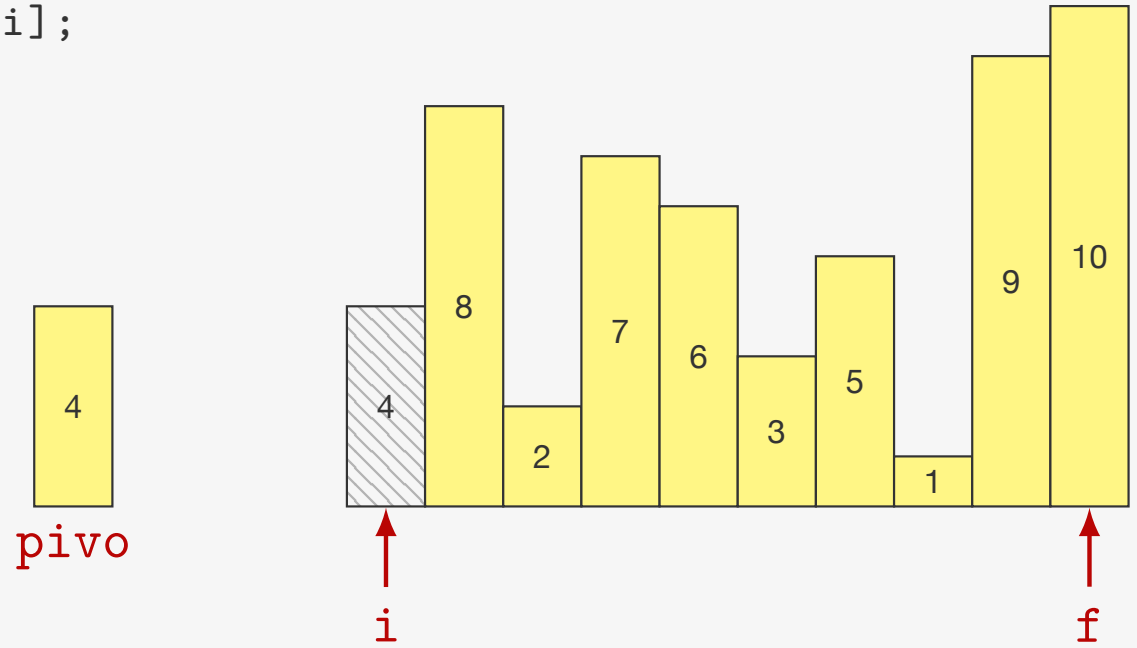
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



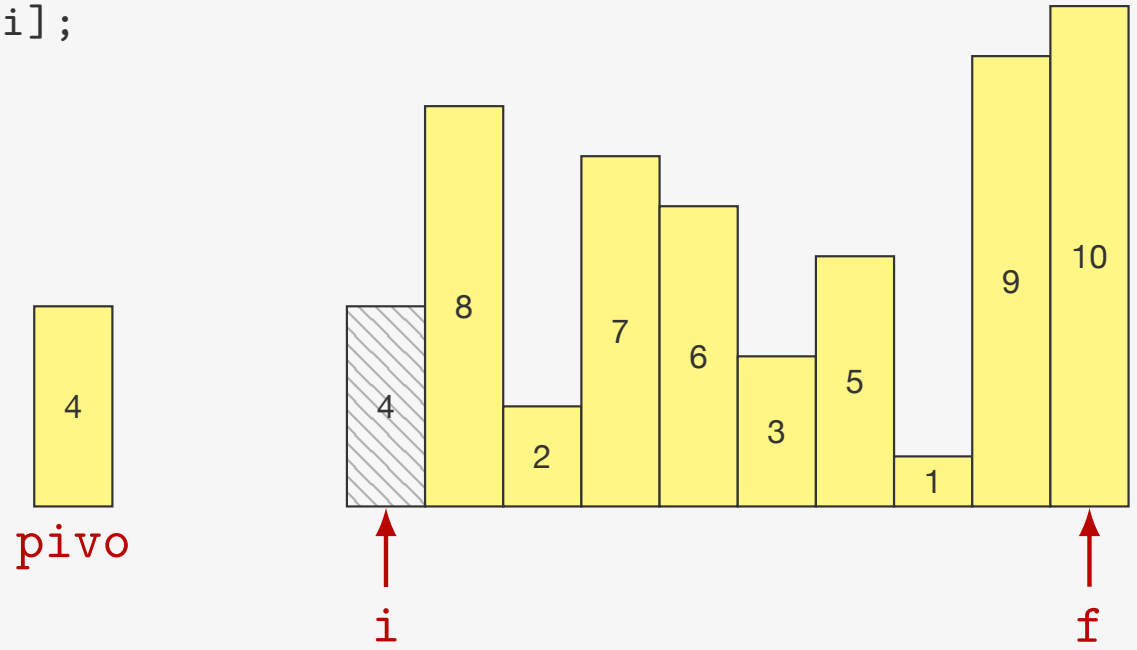
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



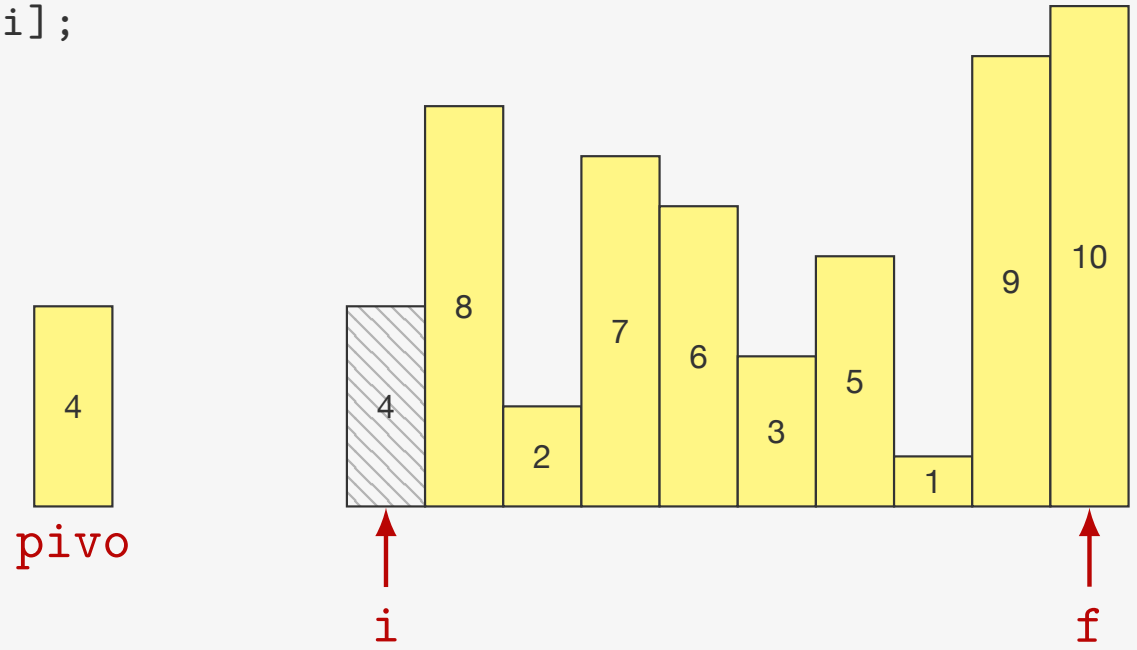
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



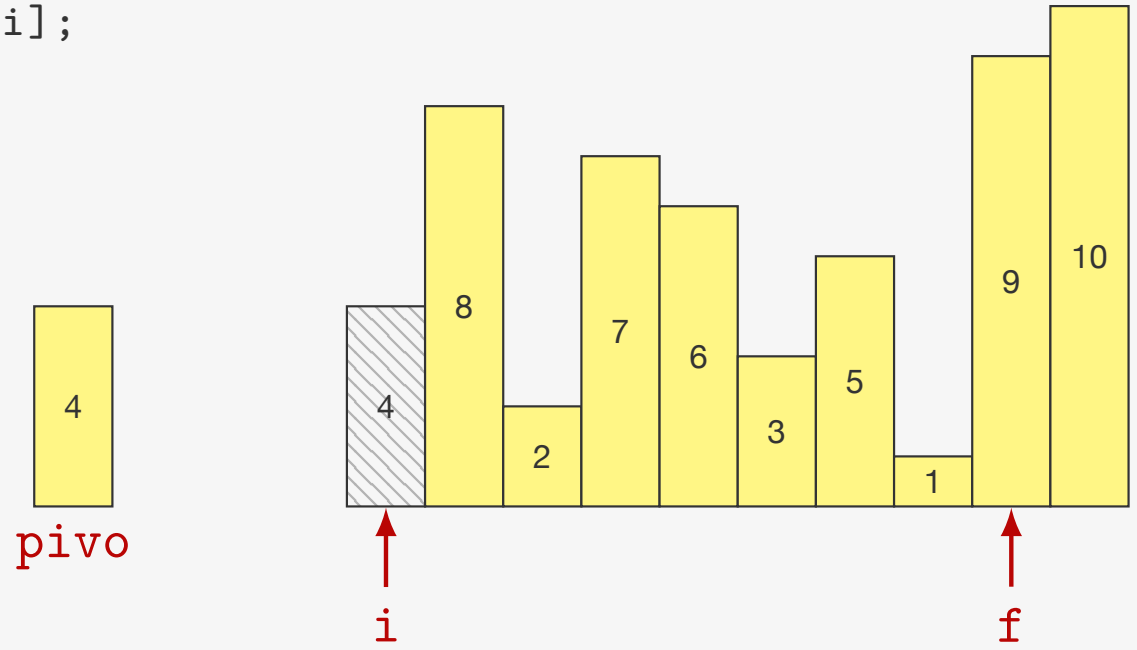
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



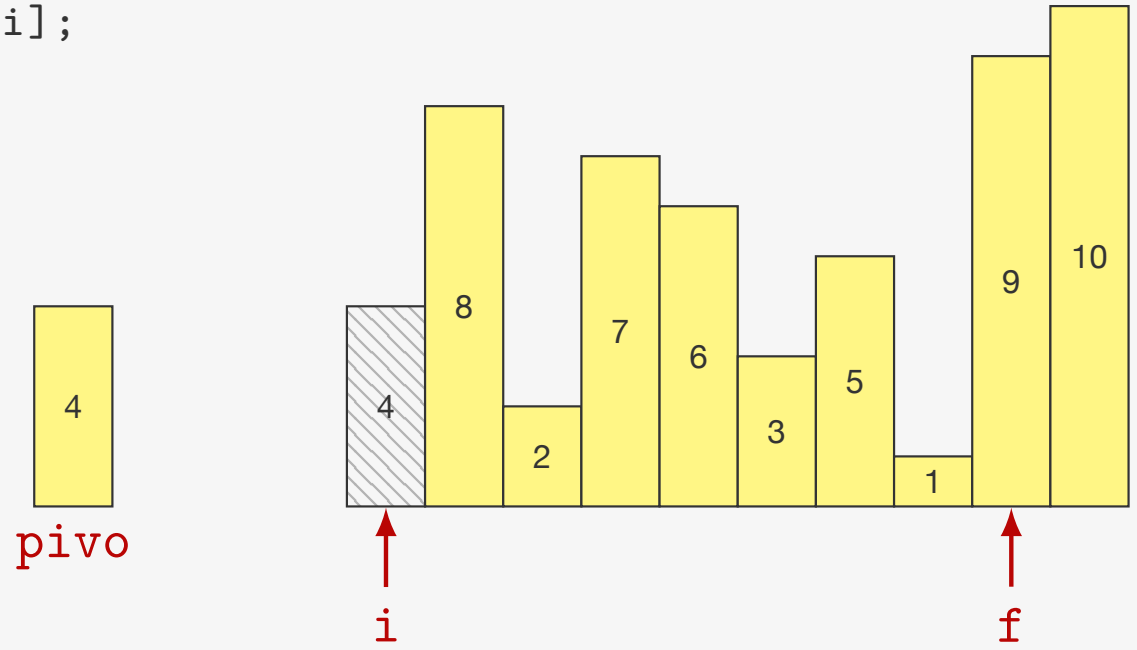
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



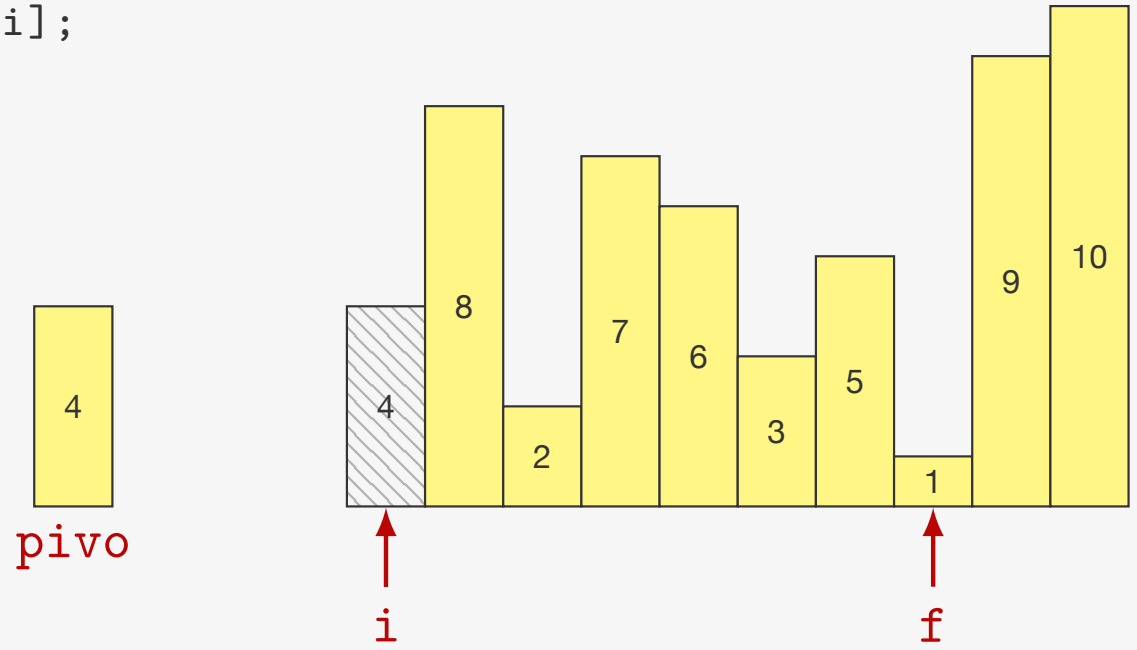
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



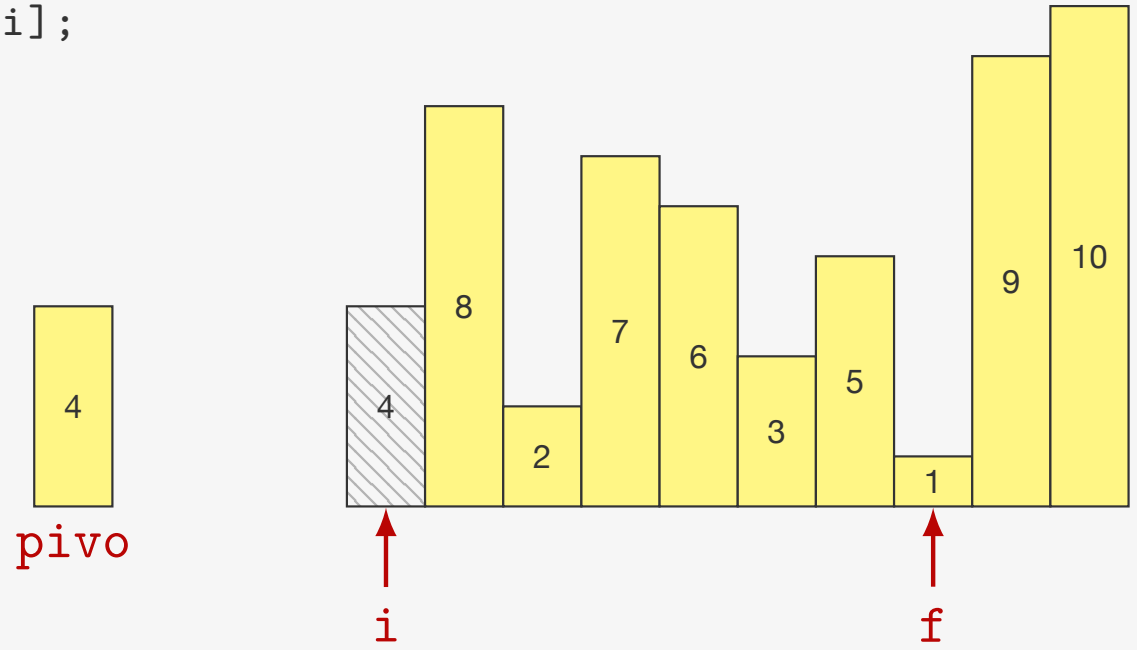
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



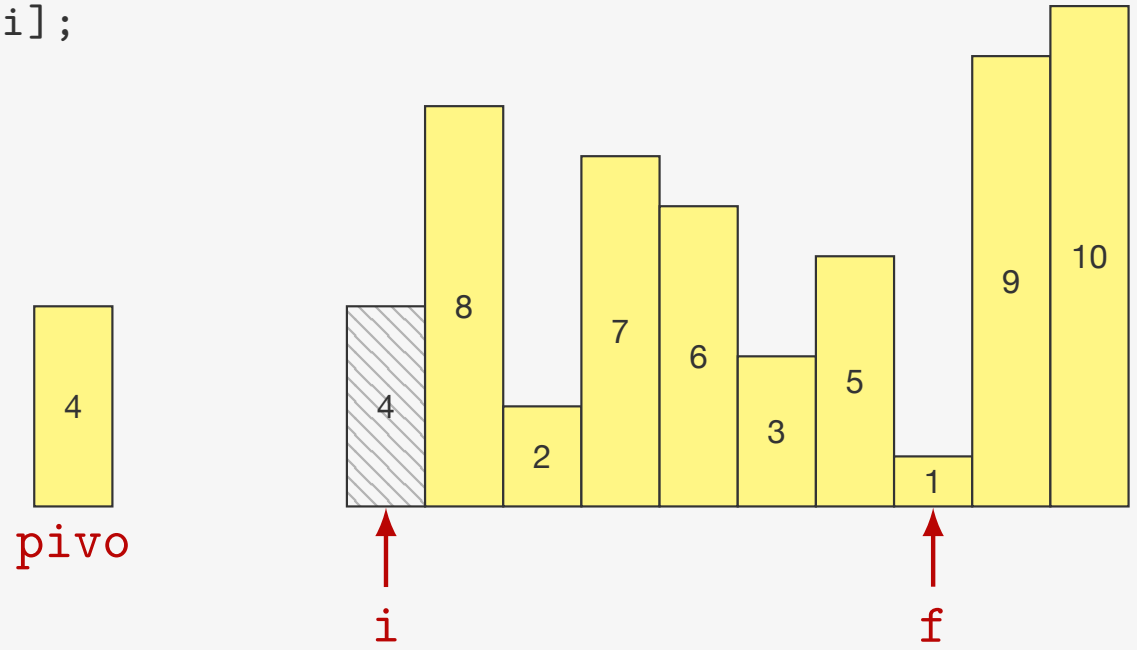
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



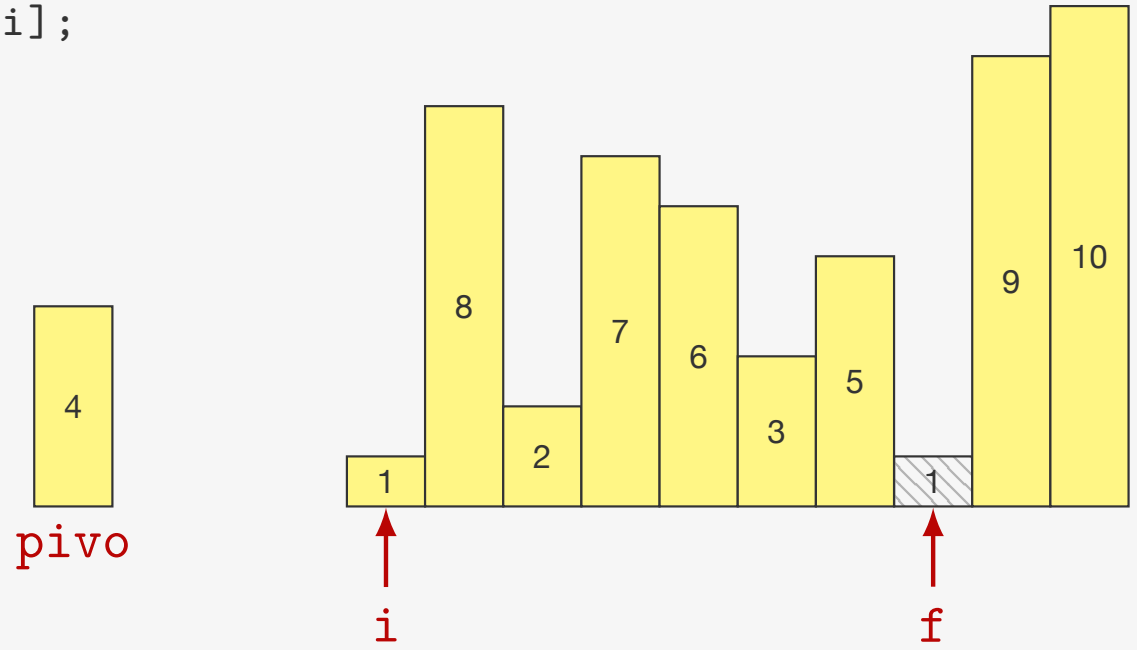
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



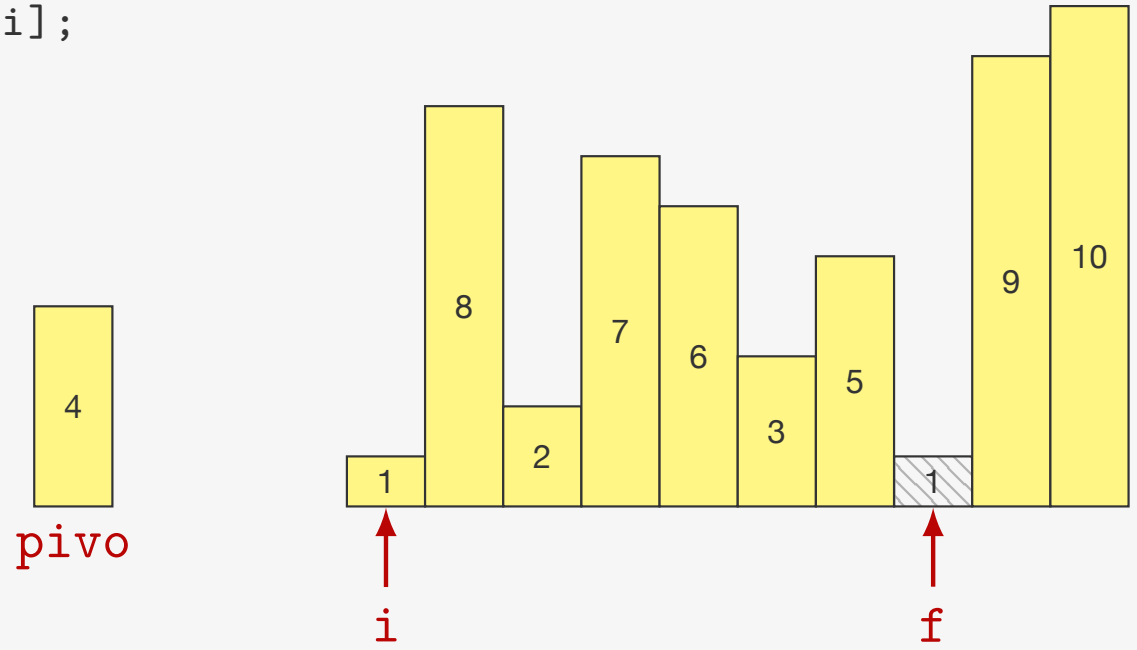
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



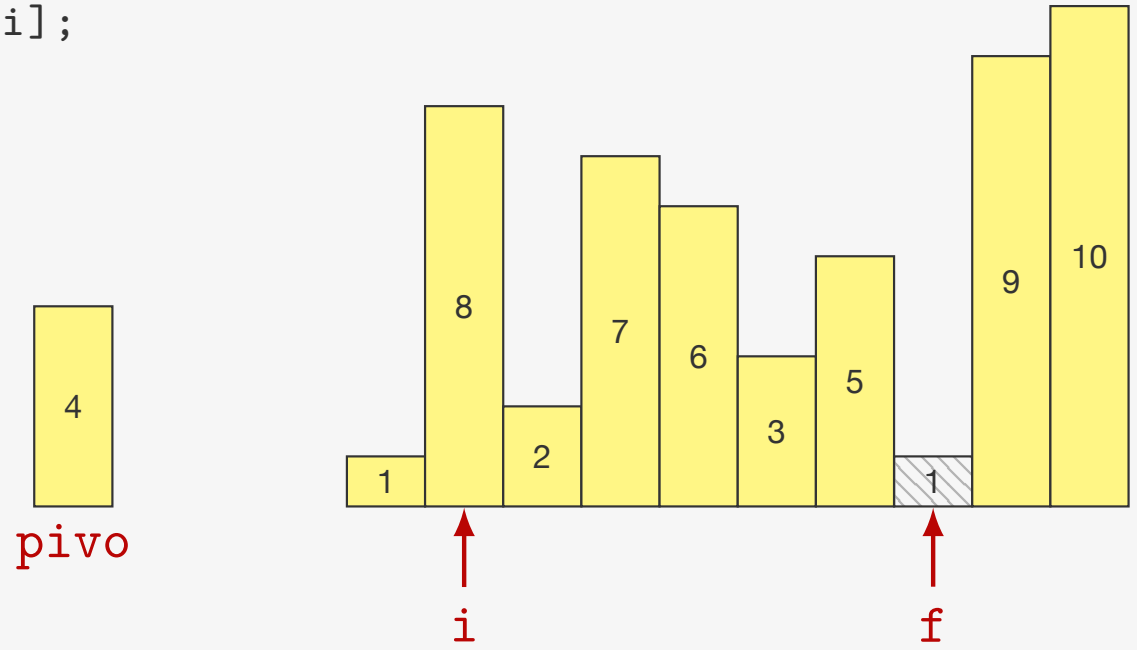
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



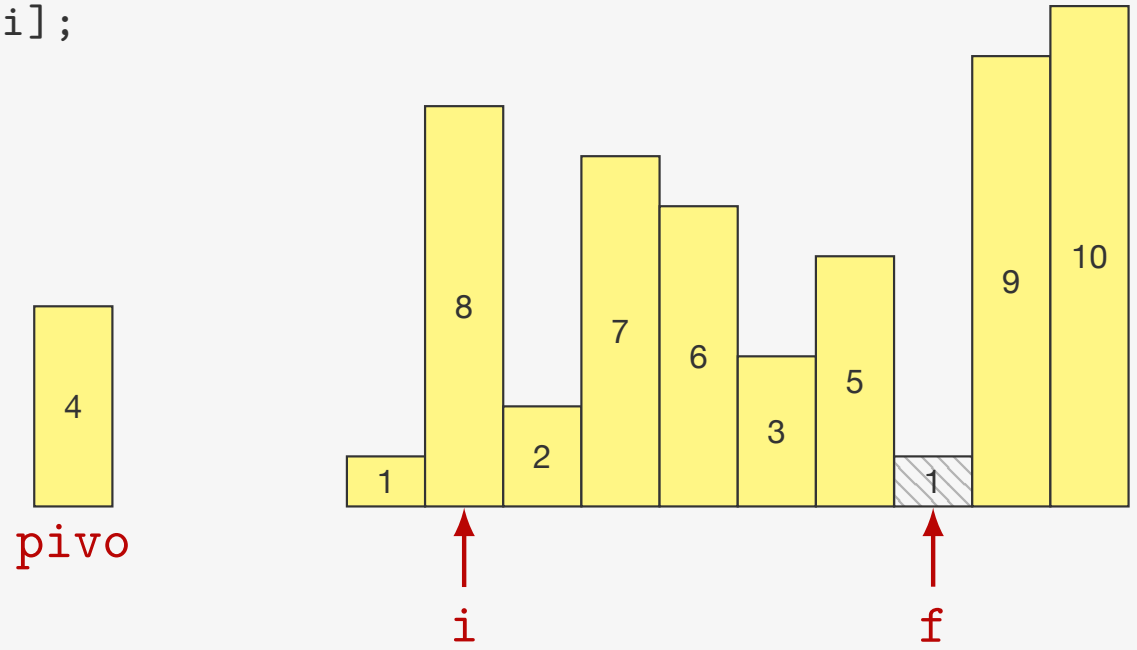
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



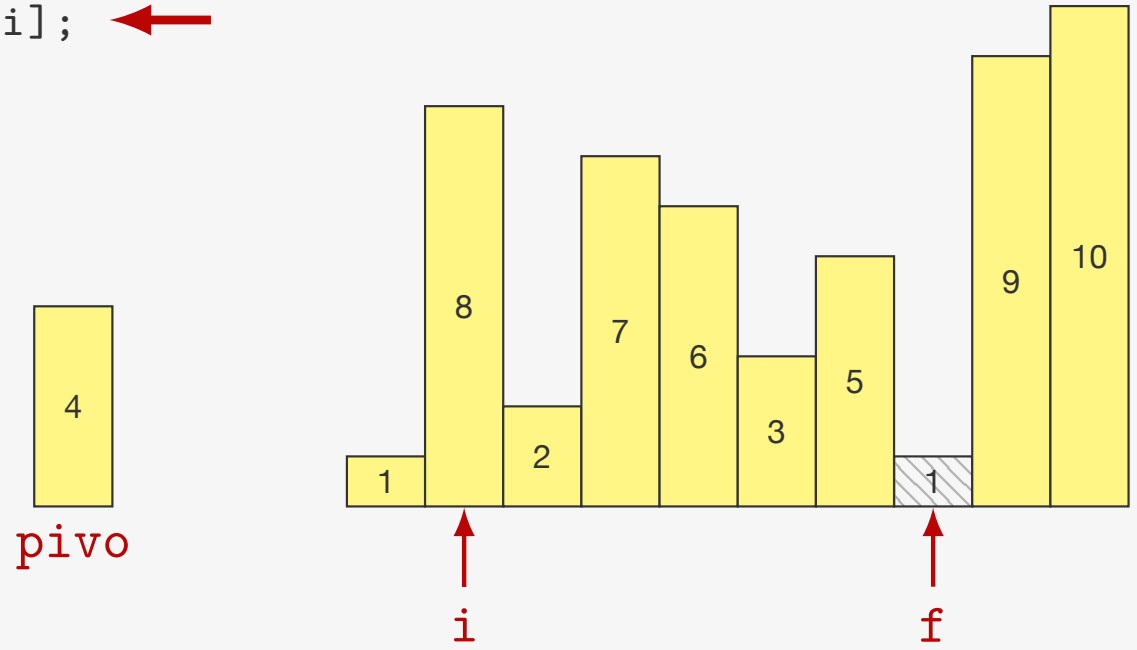
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



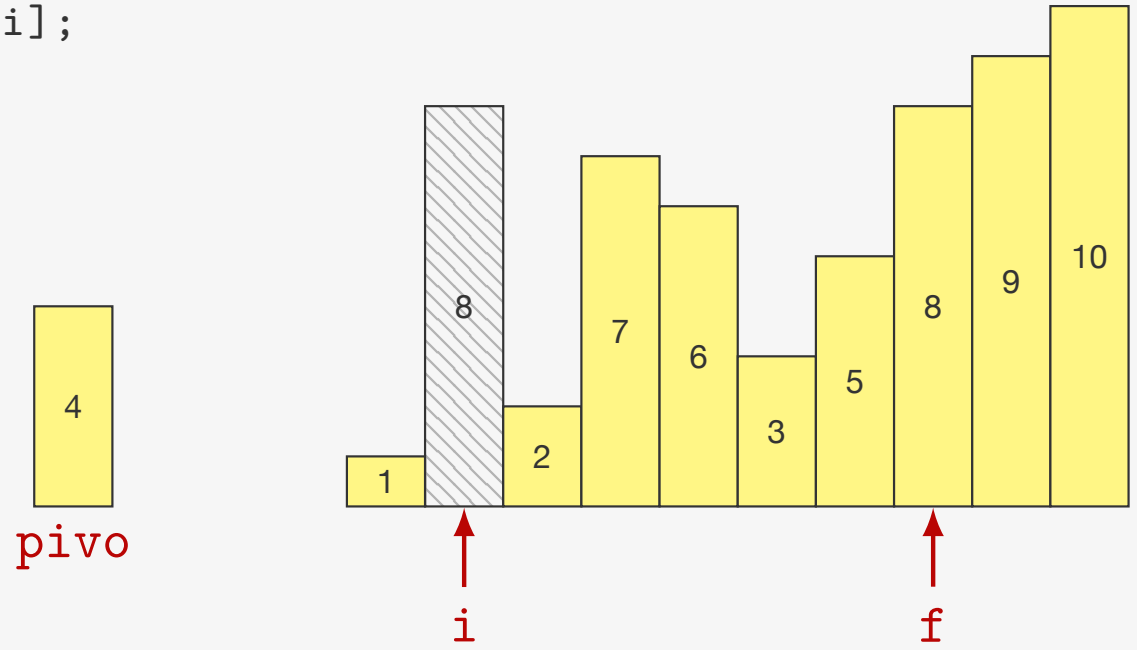
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



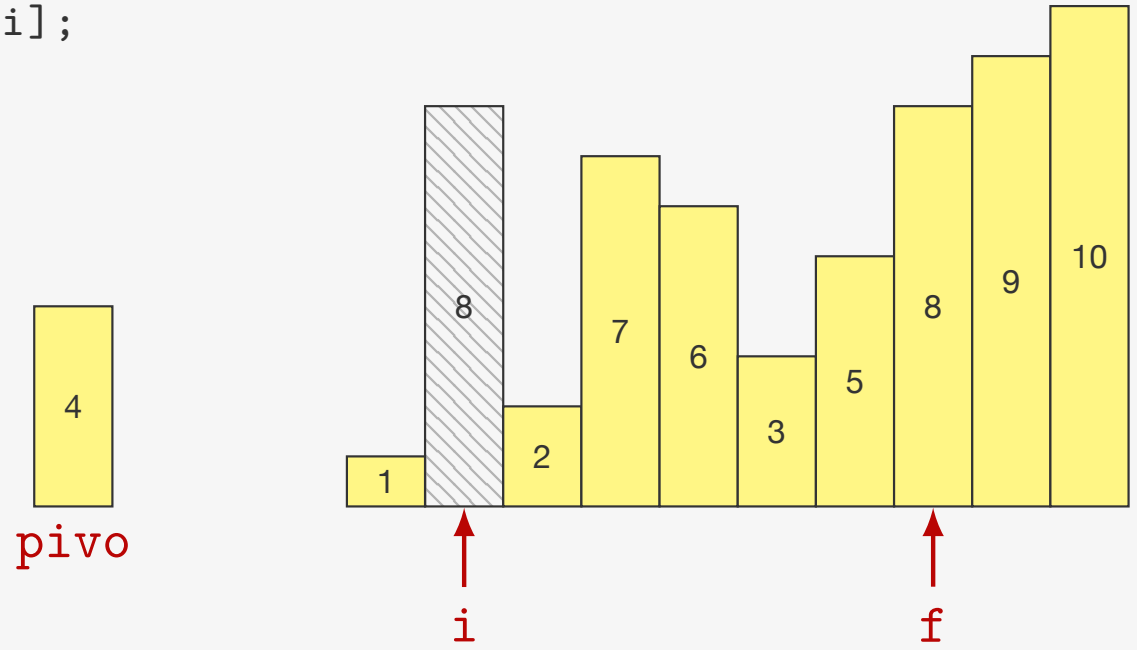
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



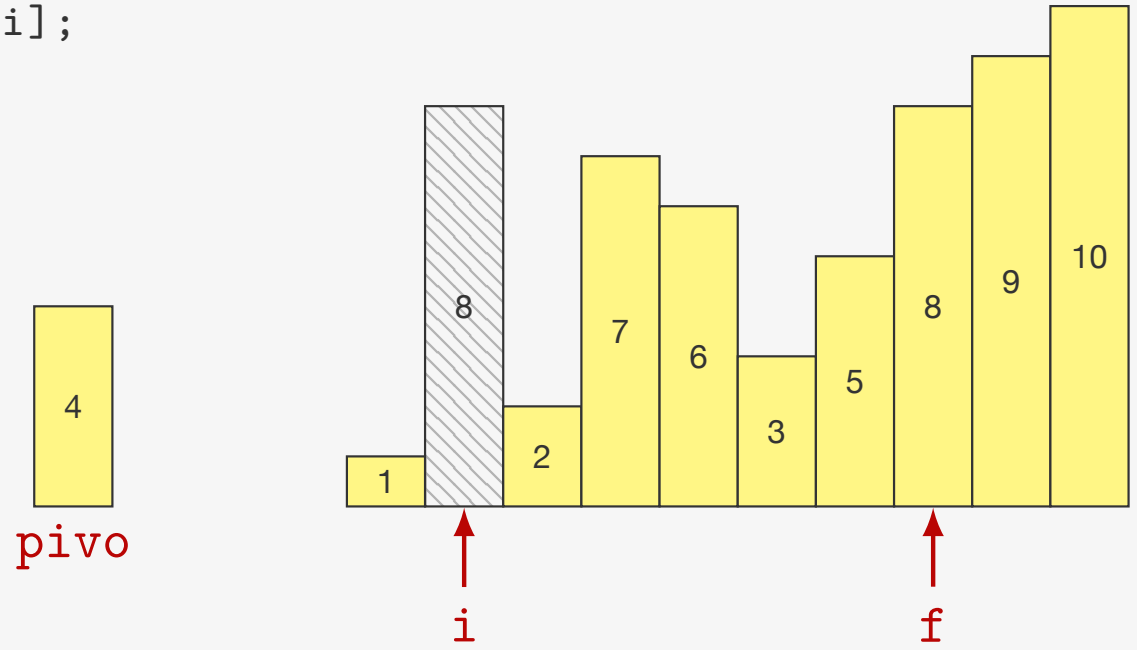
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



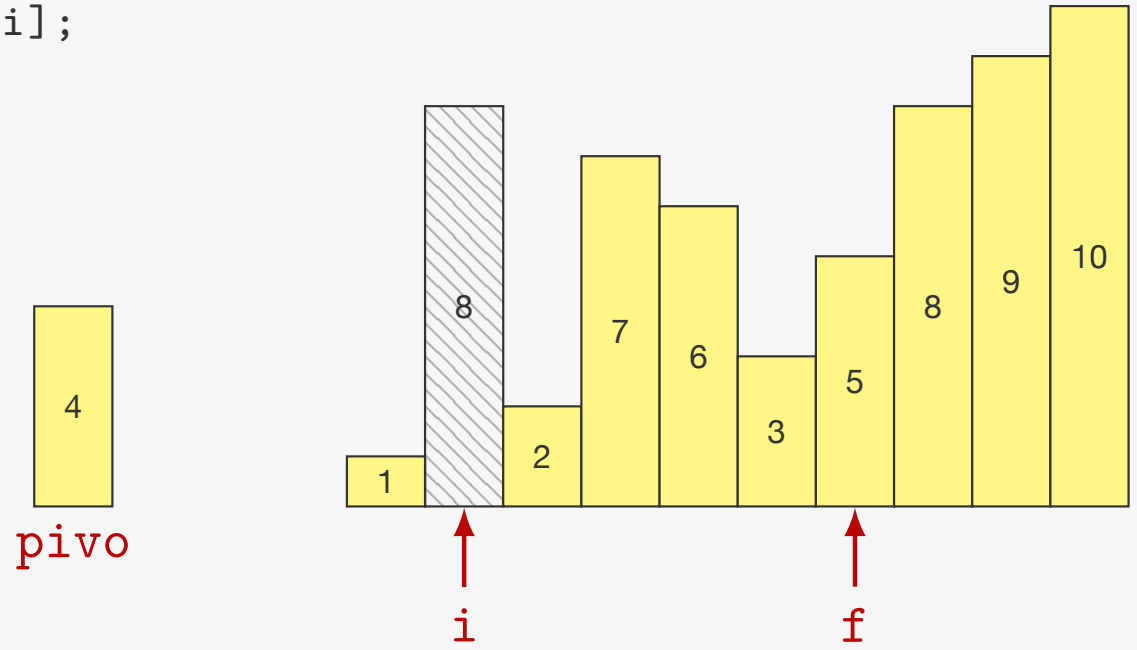
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



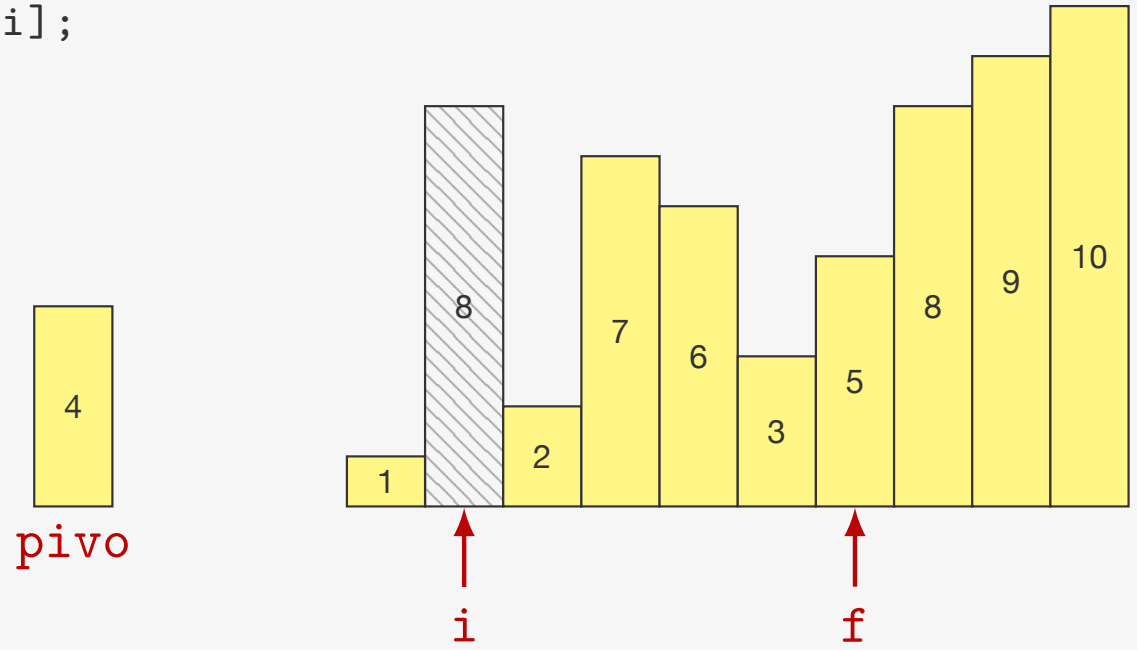
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



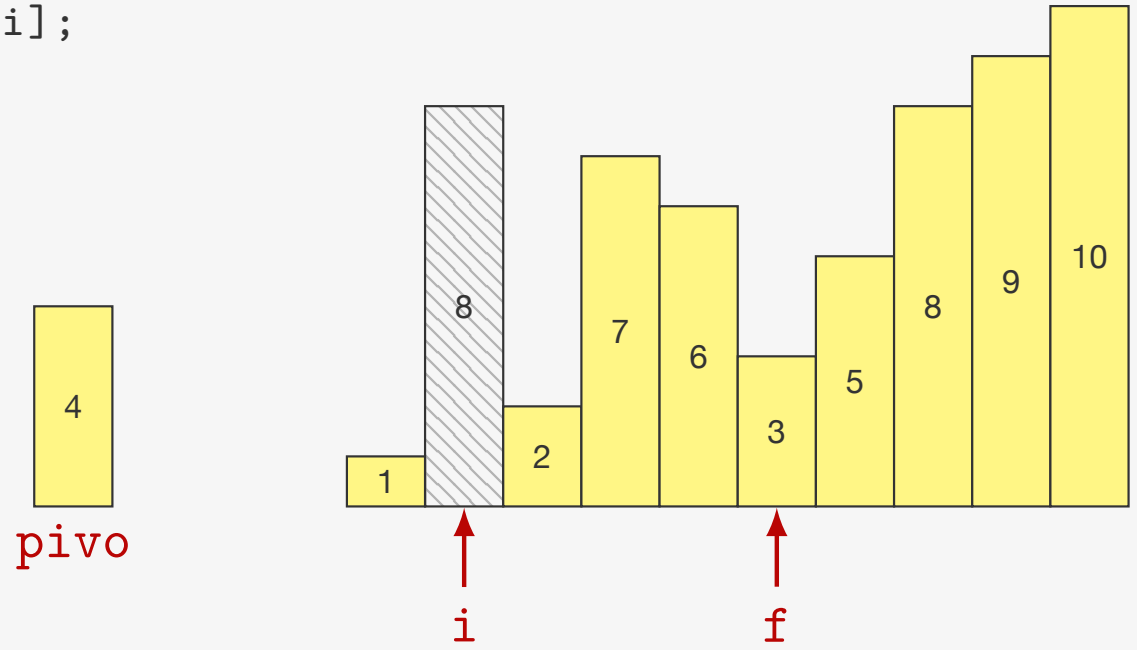
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



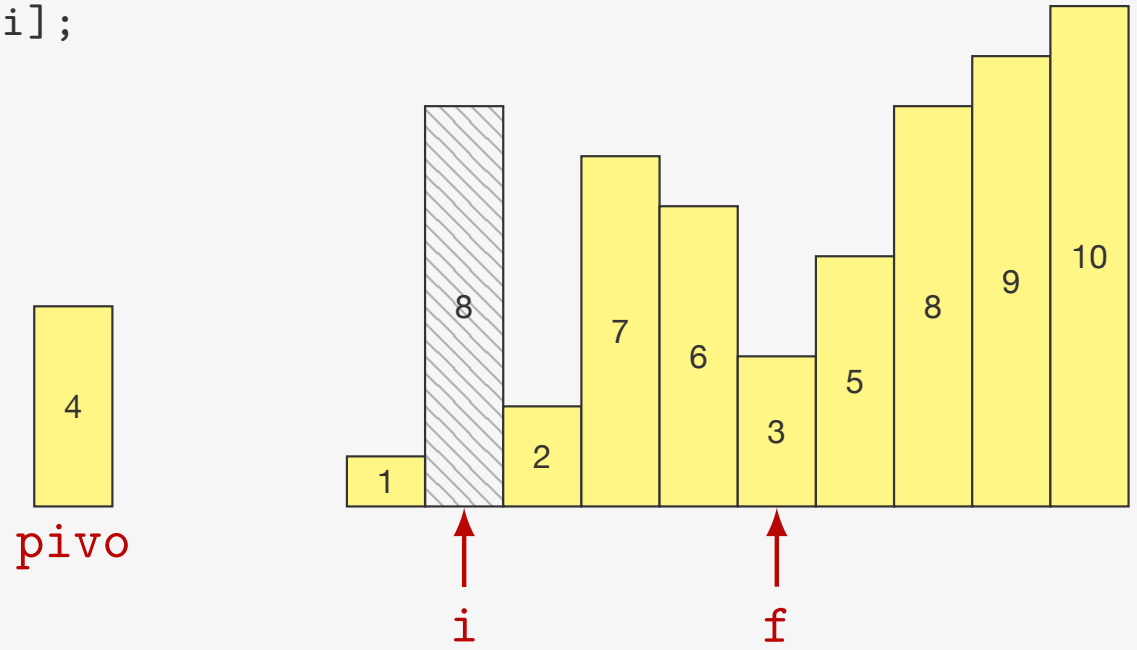
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



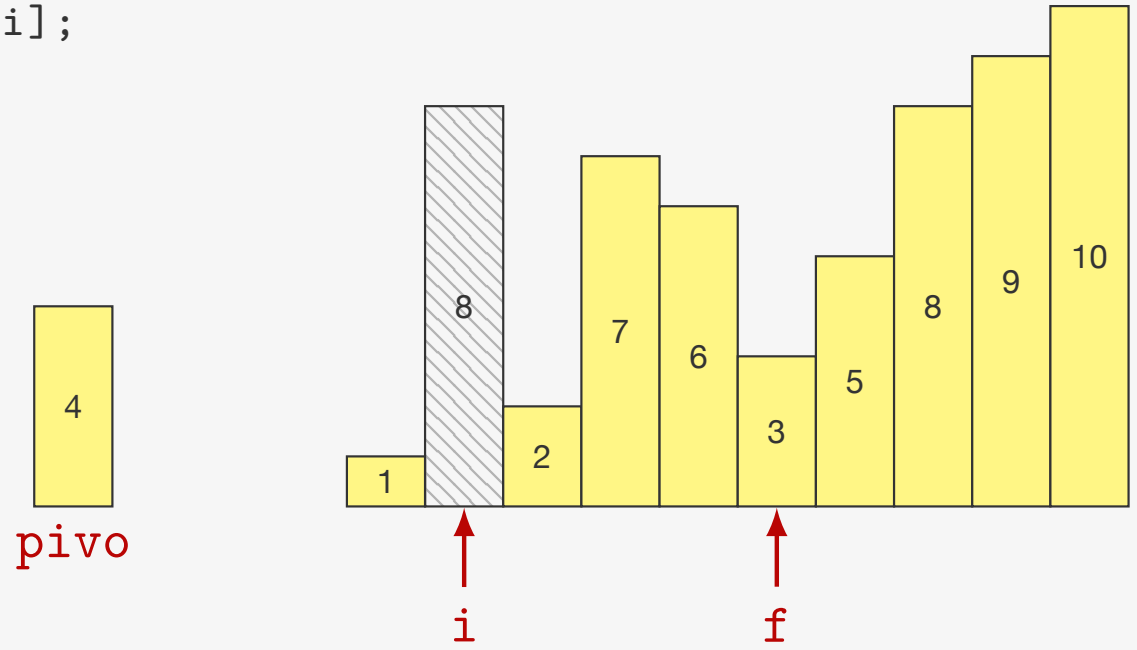
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



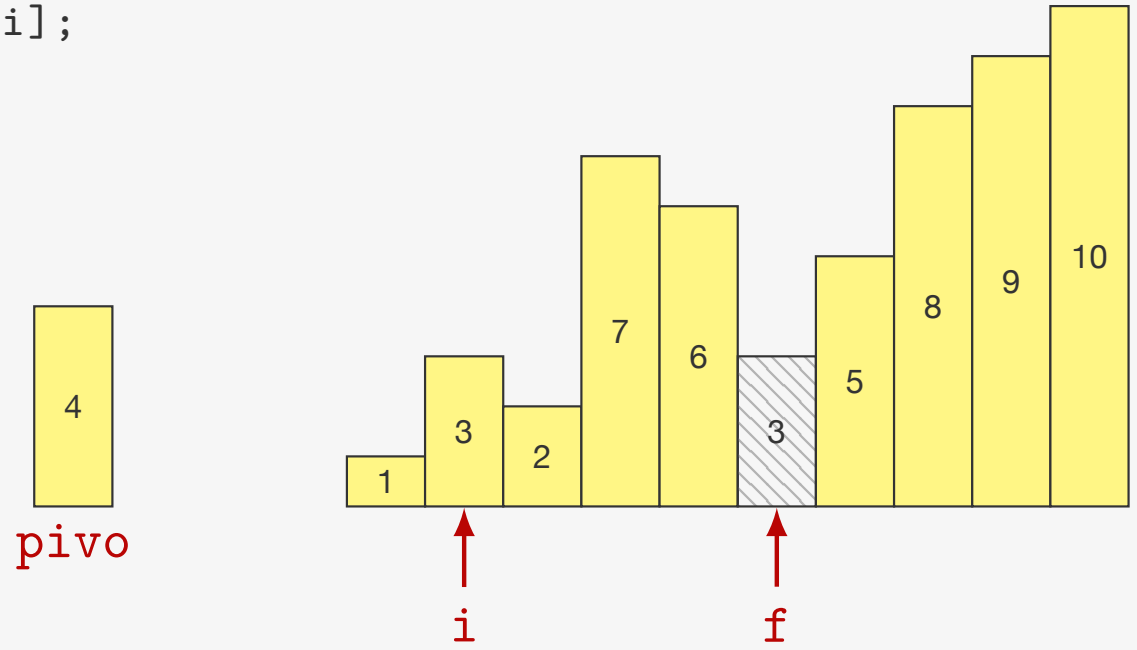
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



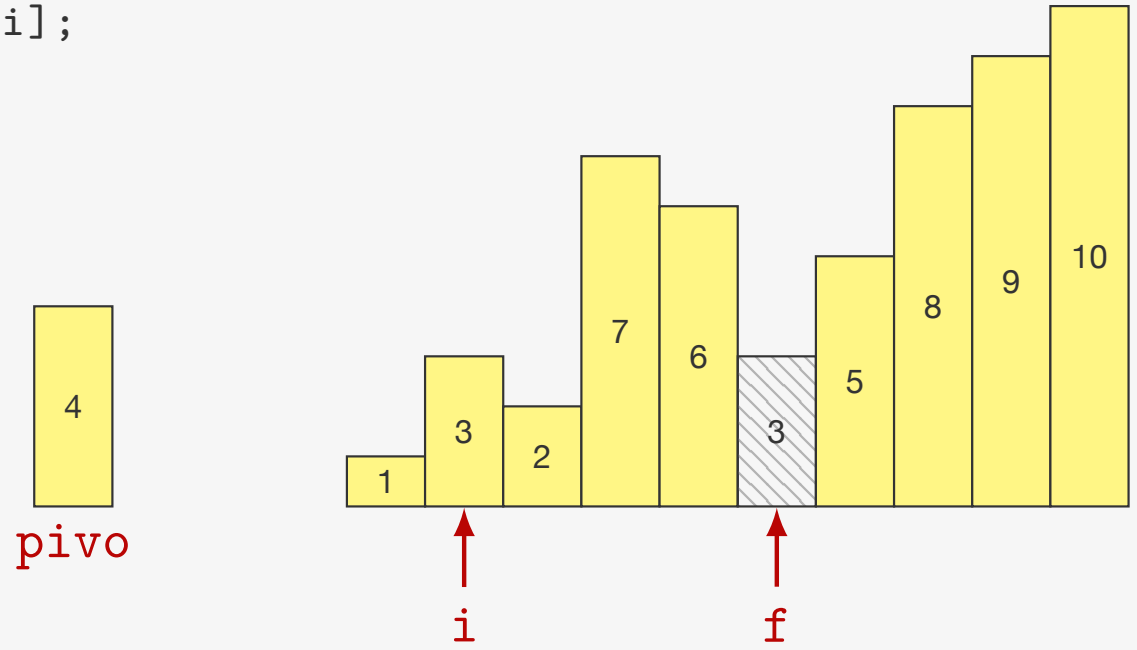
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



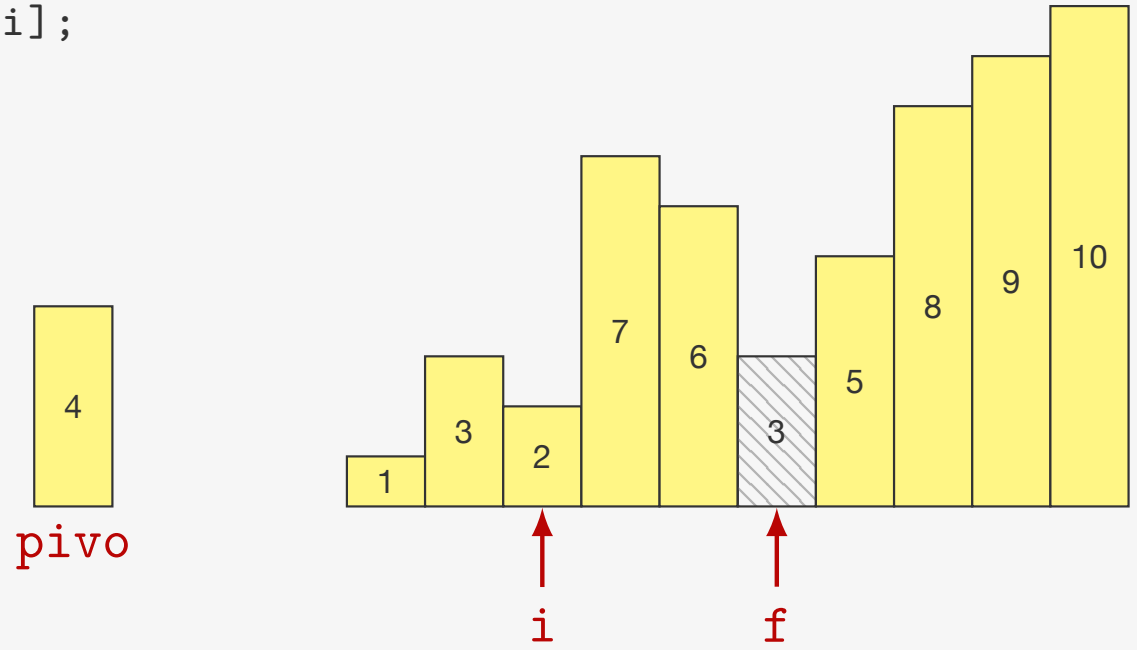
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



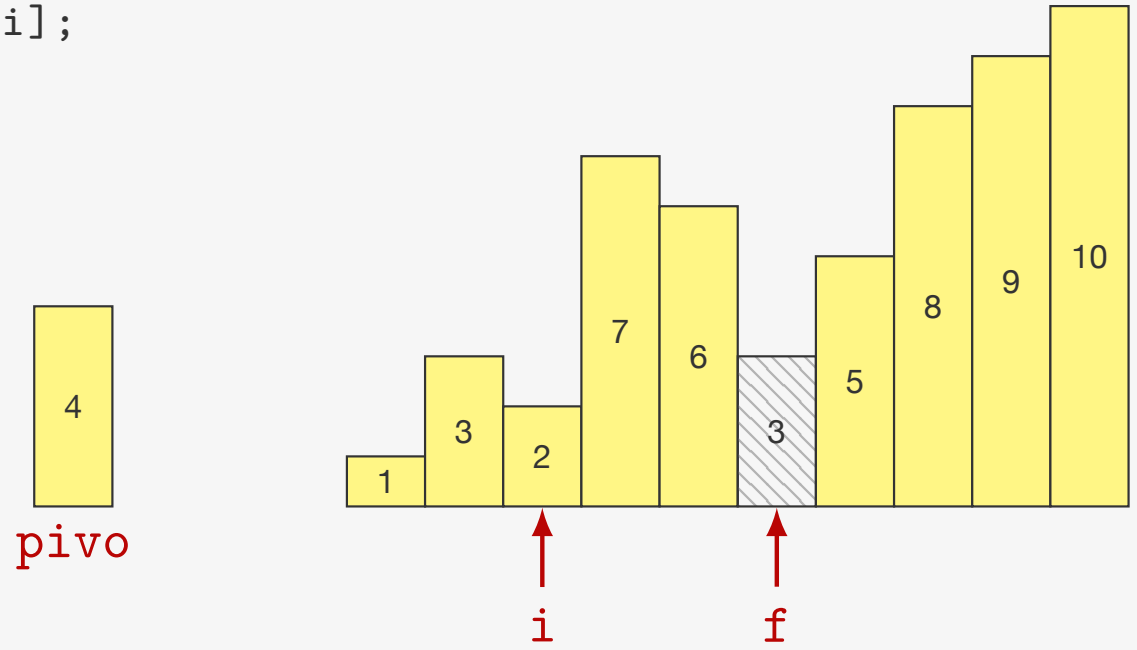
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



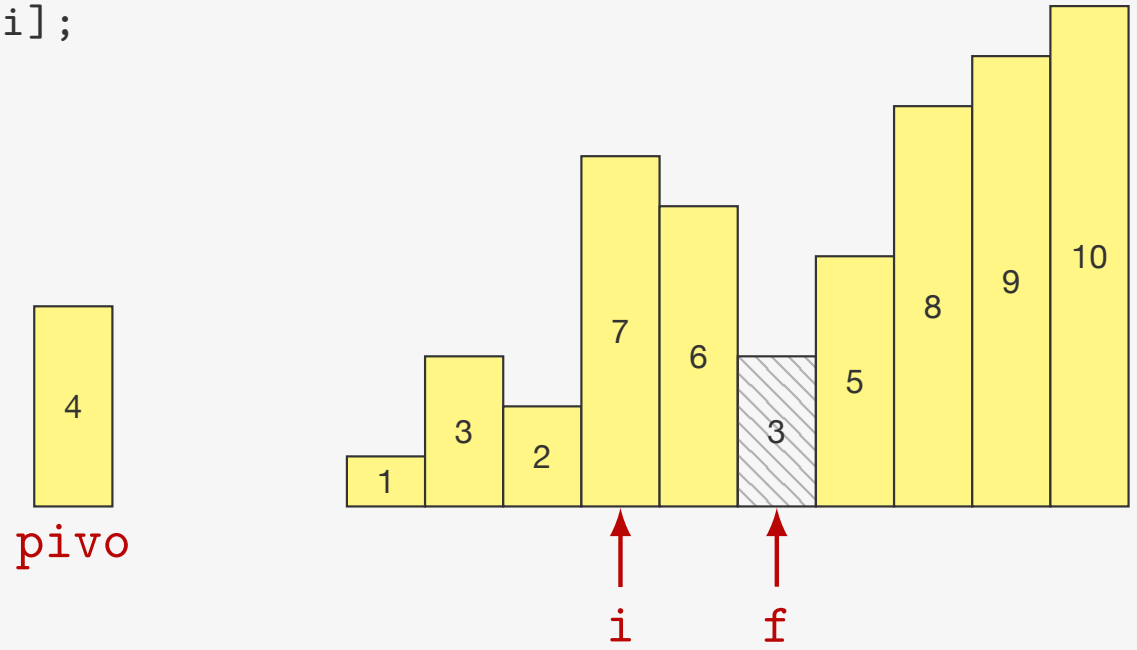
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



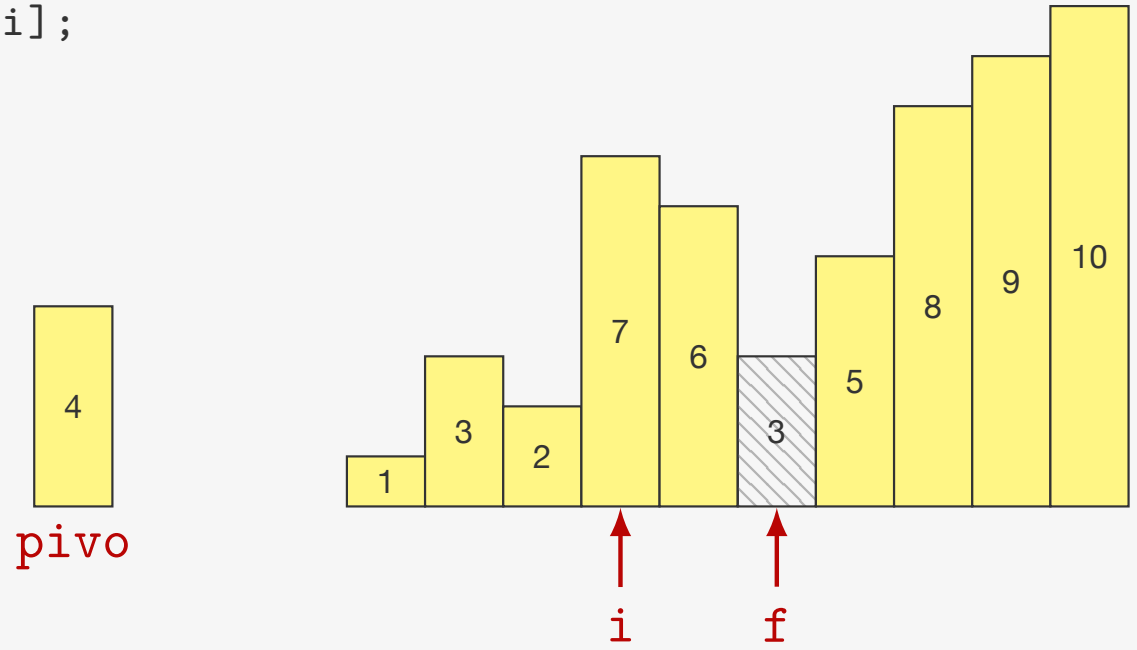
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



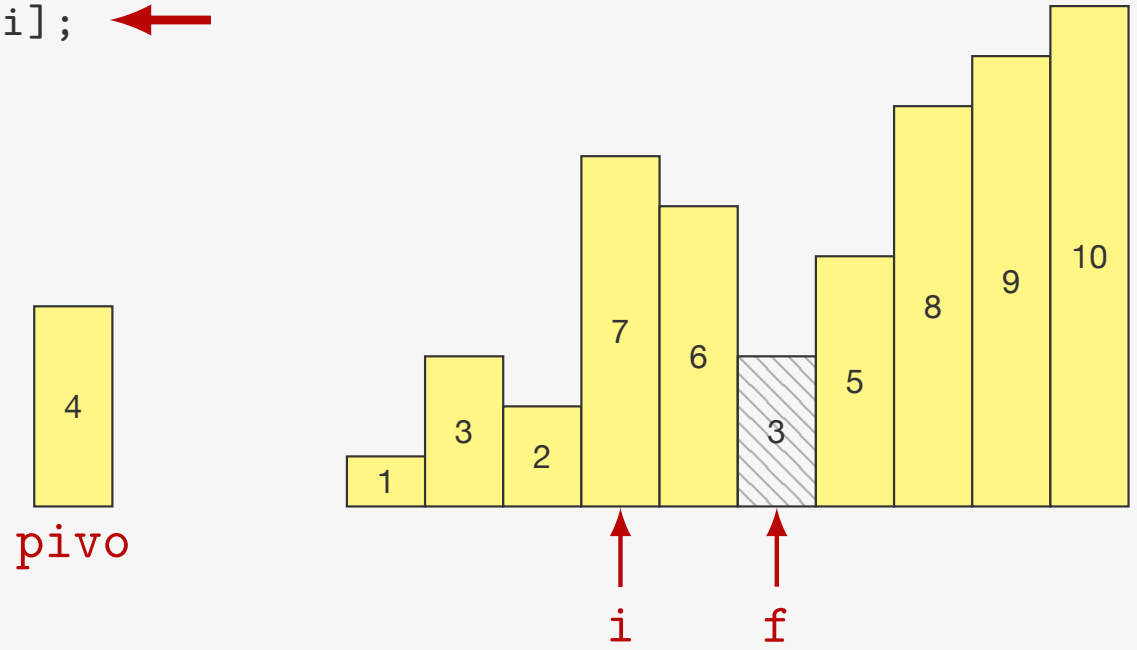
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



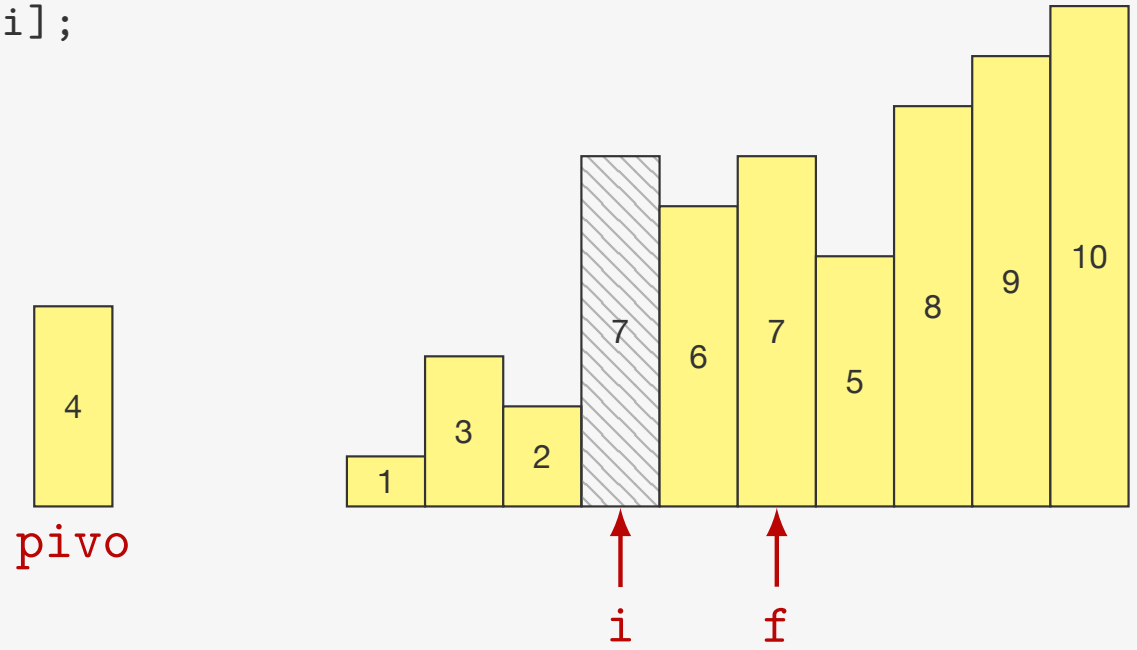
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



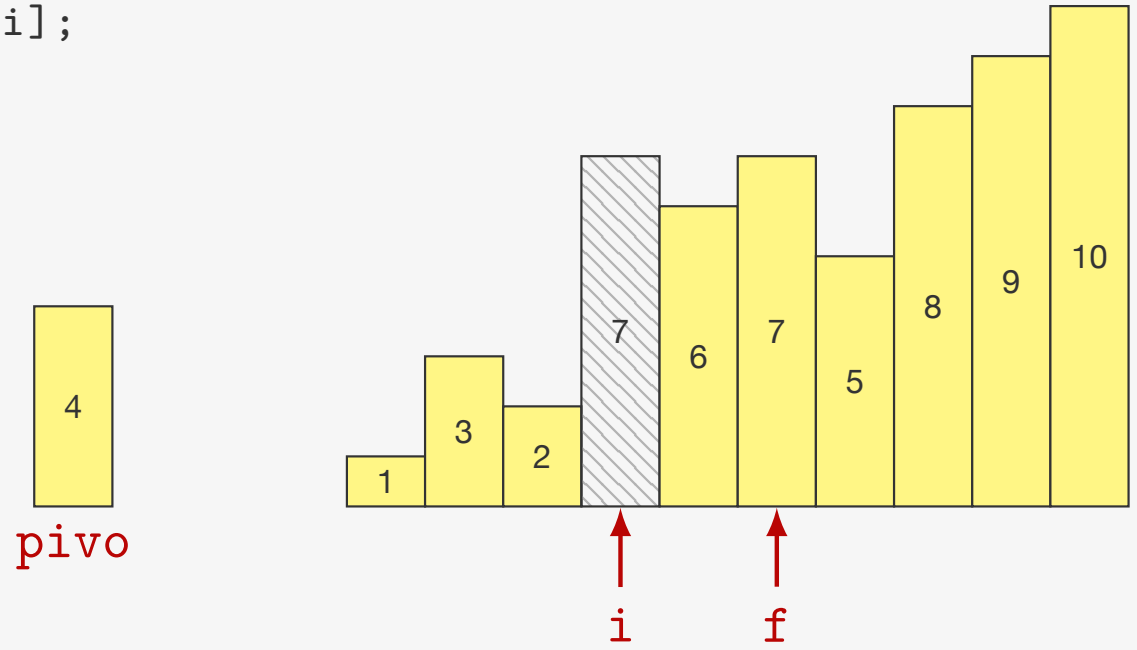
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



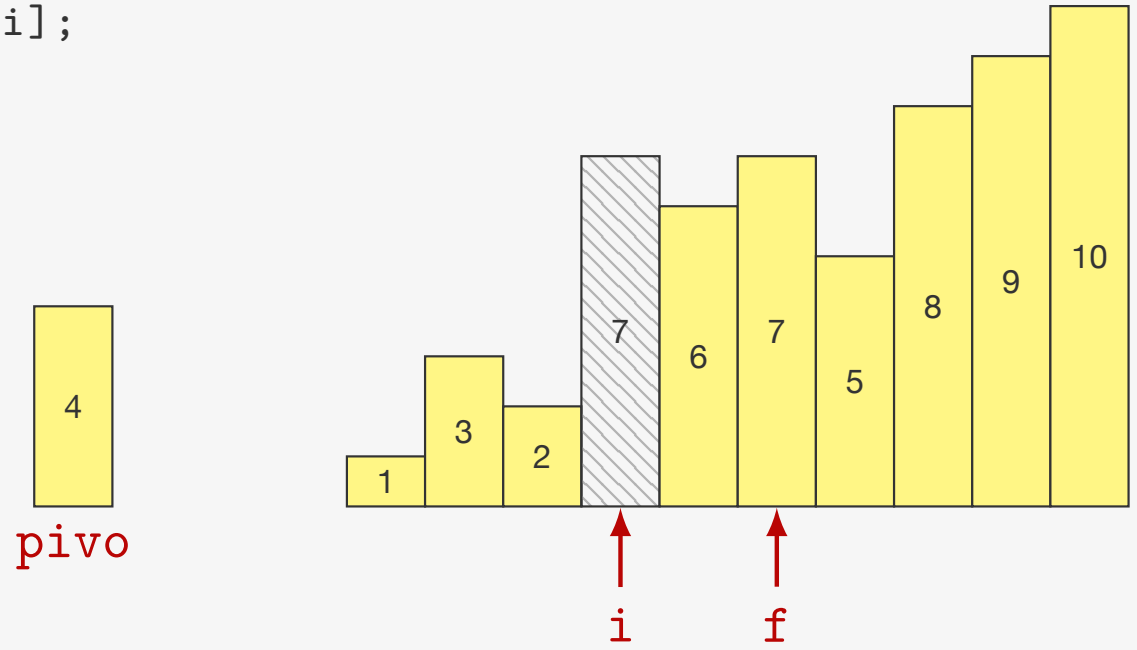
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



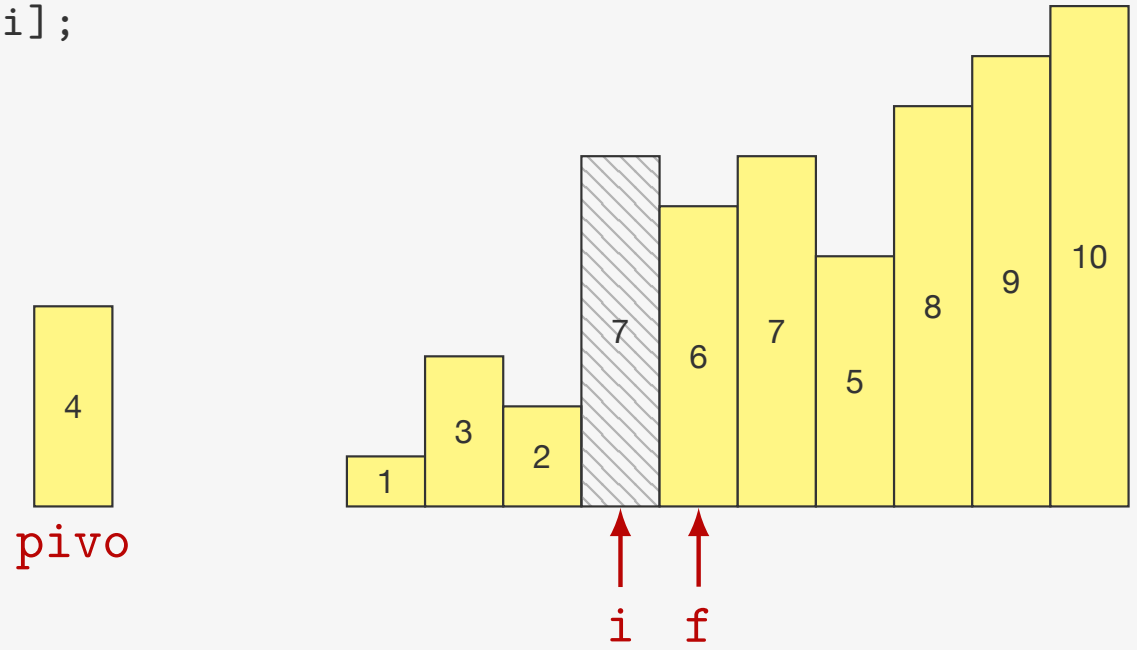
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



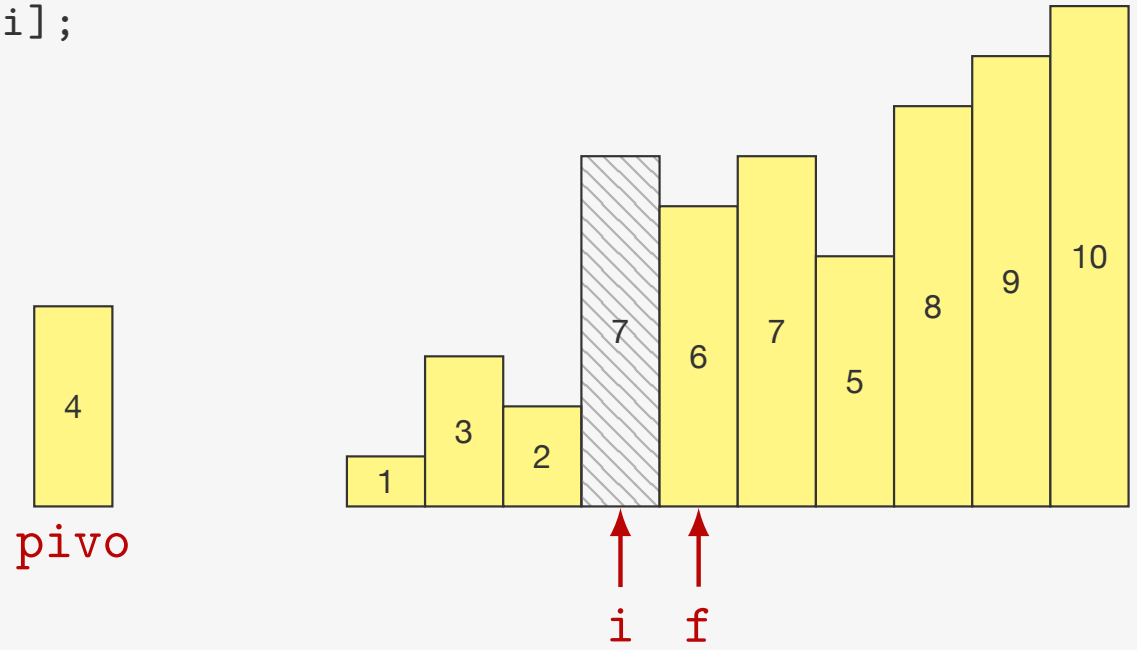
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



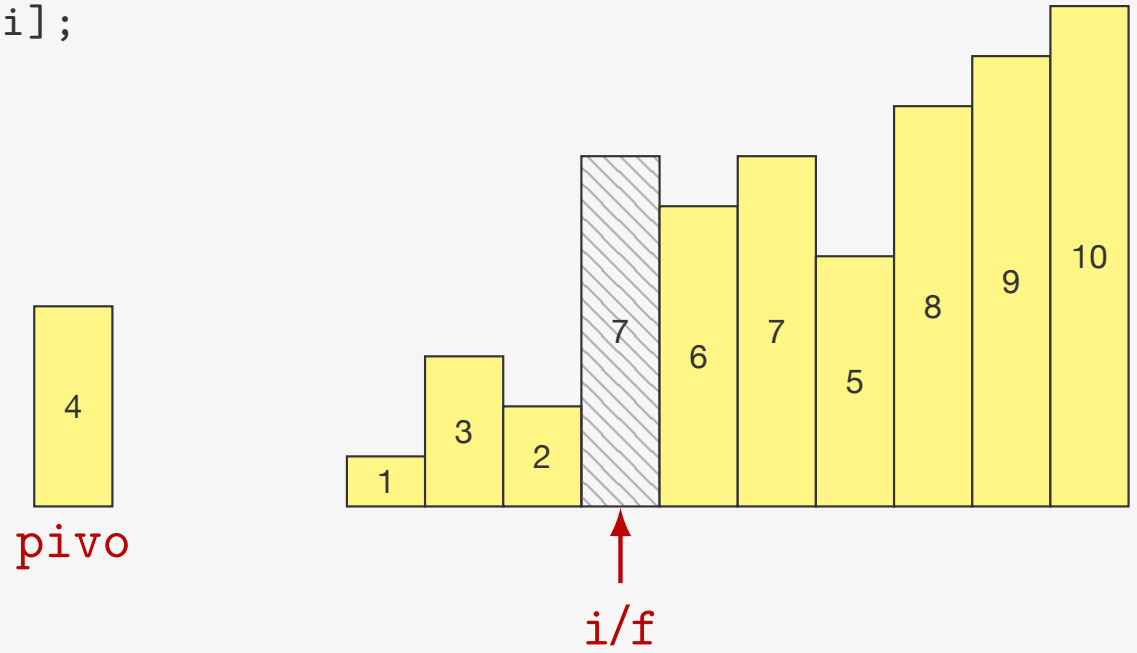
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



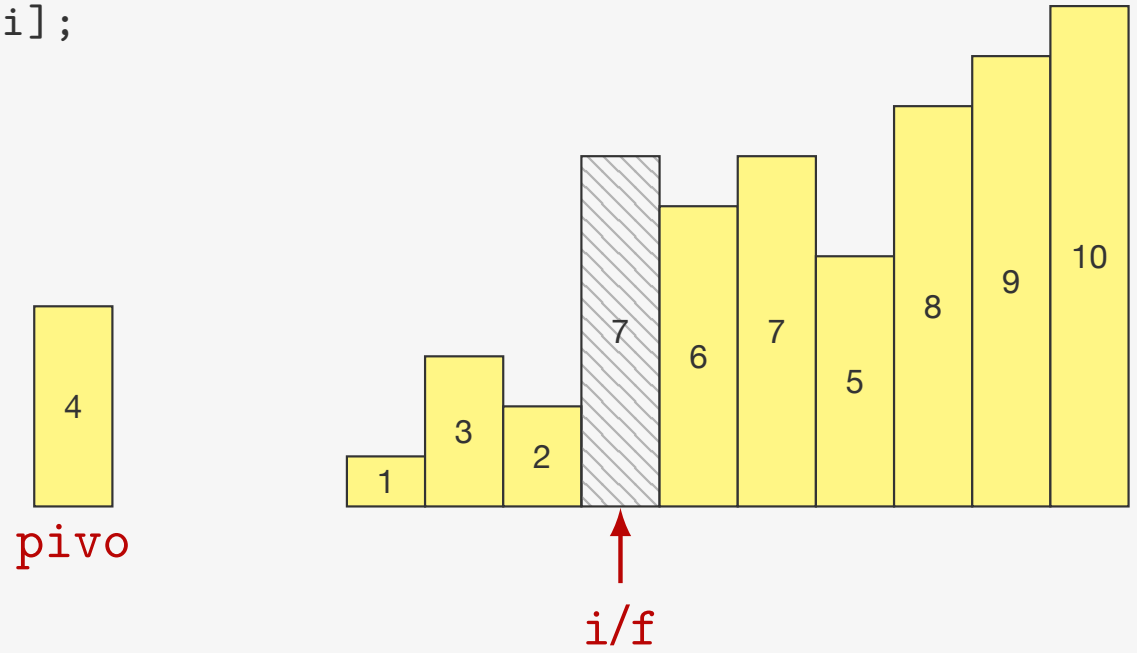
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



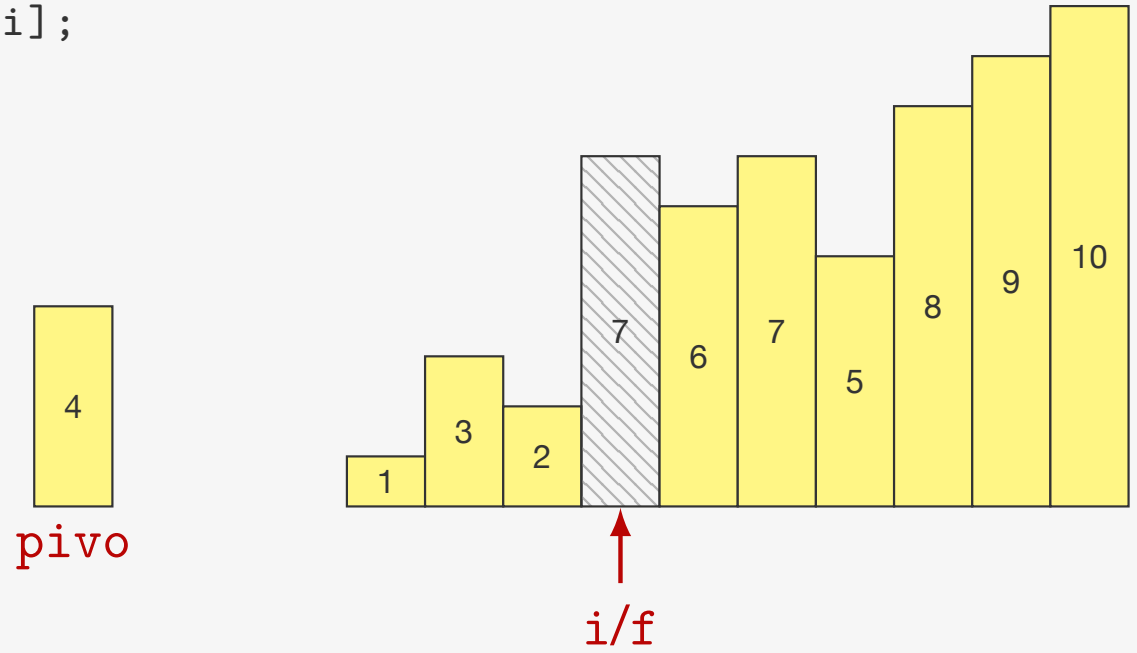
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



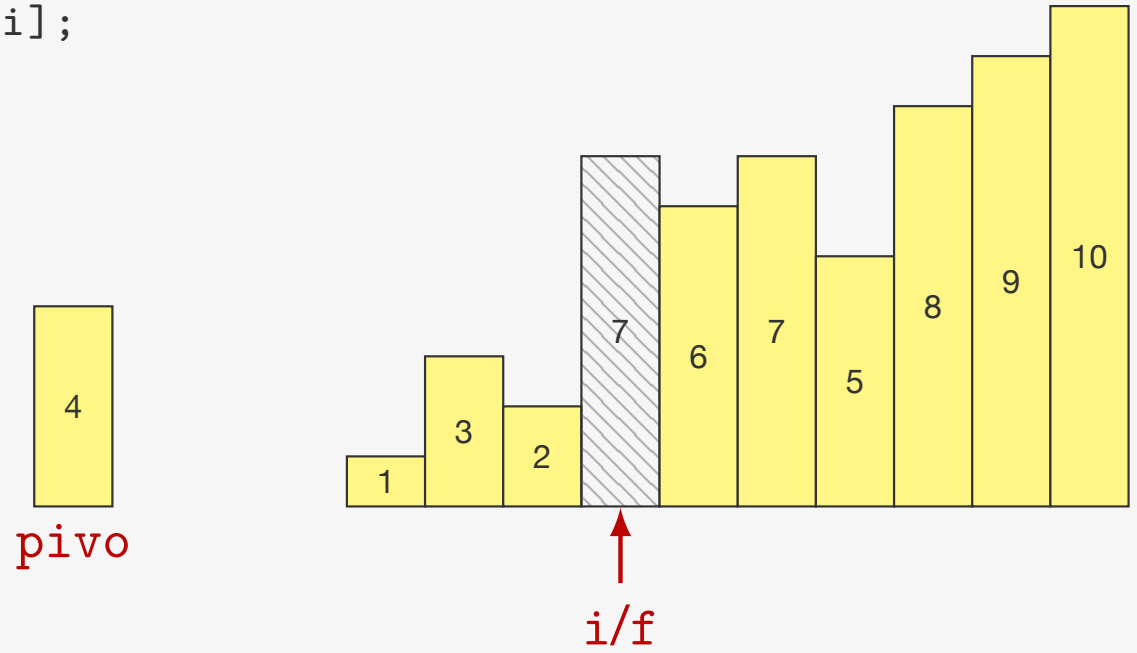
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



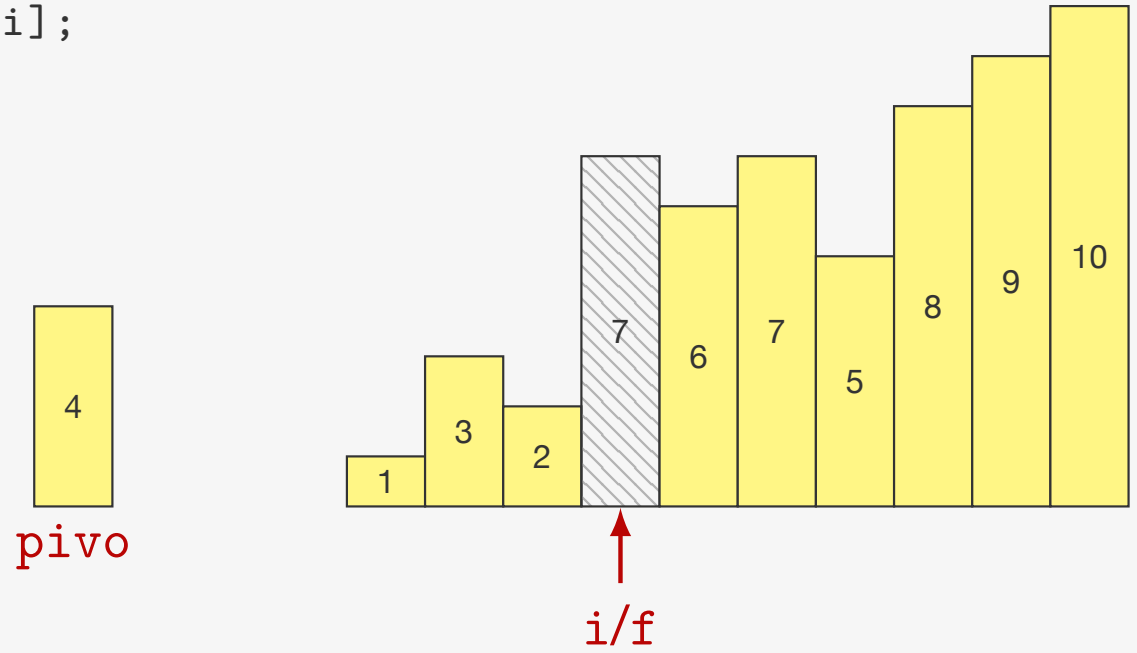
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



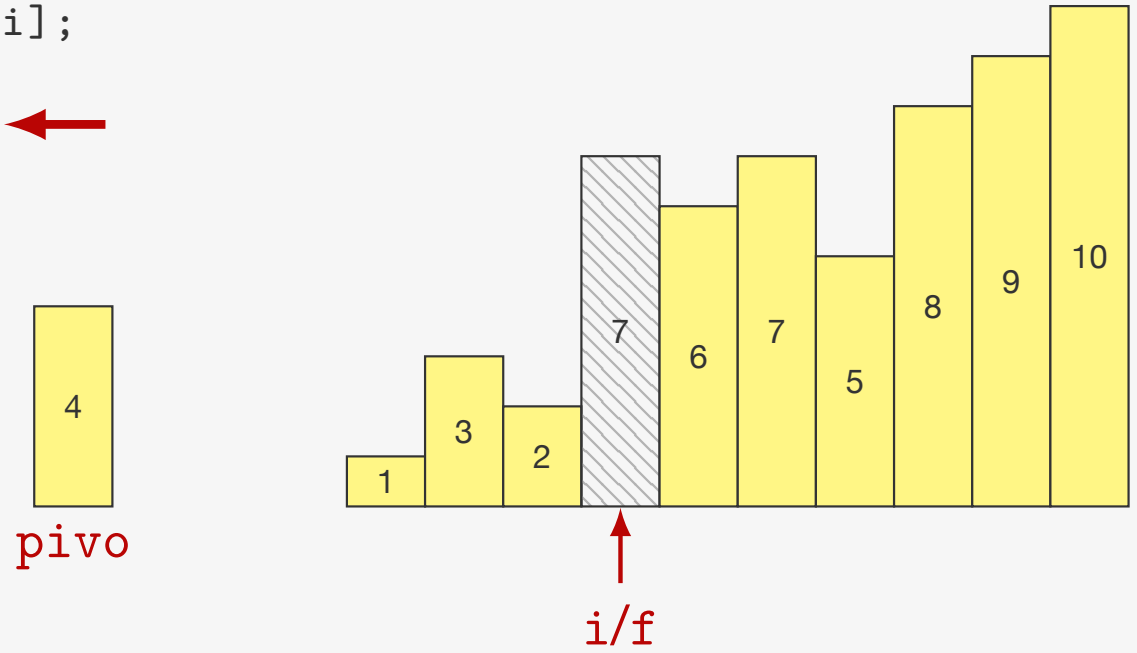
Particionamento

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



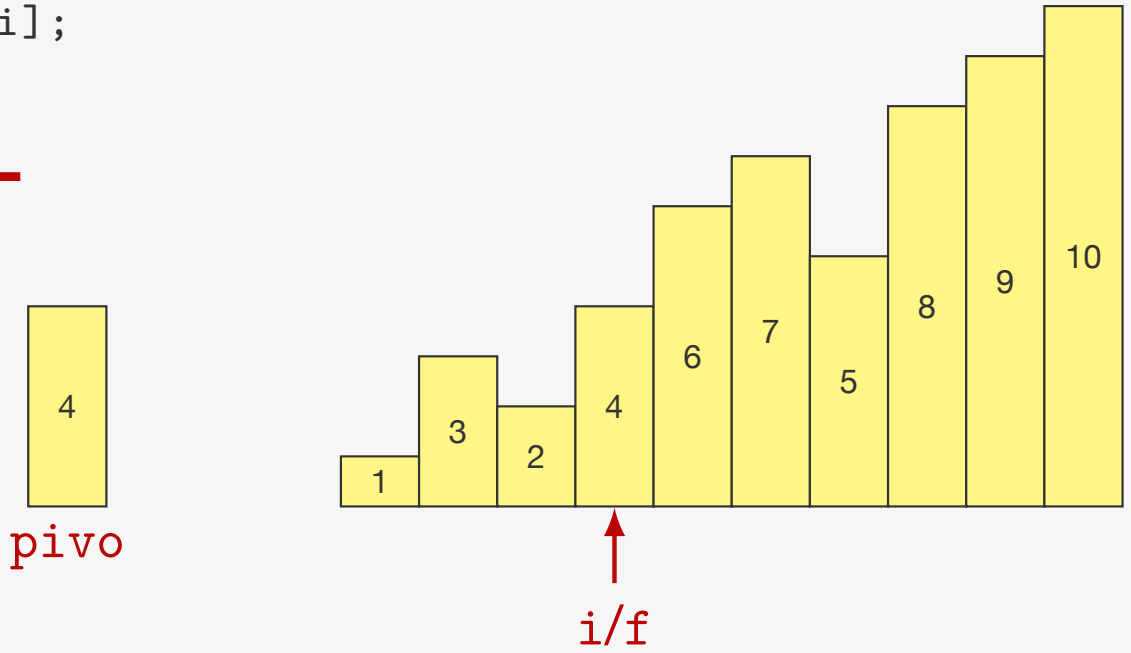
Particionamento

```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



Particionamento

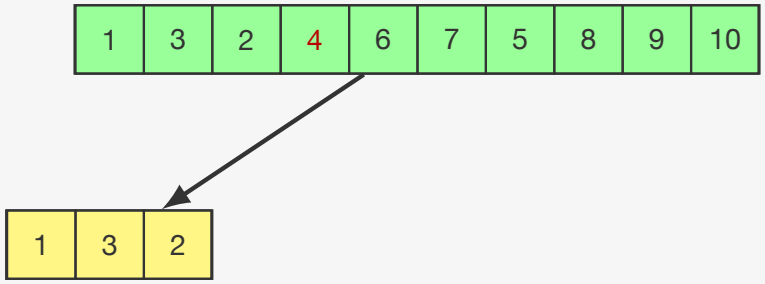
```
1  int partition(int *v, int l, int r) {
2    int pivo = v[l];
3    int i = l, f = r;
4    while (i < f) {
5      while (i < f && v[f] >= pivo)
6        f--;
7      if (i < f)
8        v[i] = v[f];
9      while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```



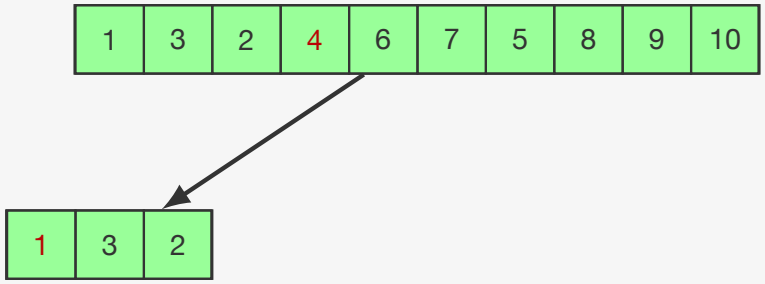
Simulação do Quicksort

Simulação do Quicksort

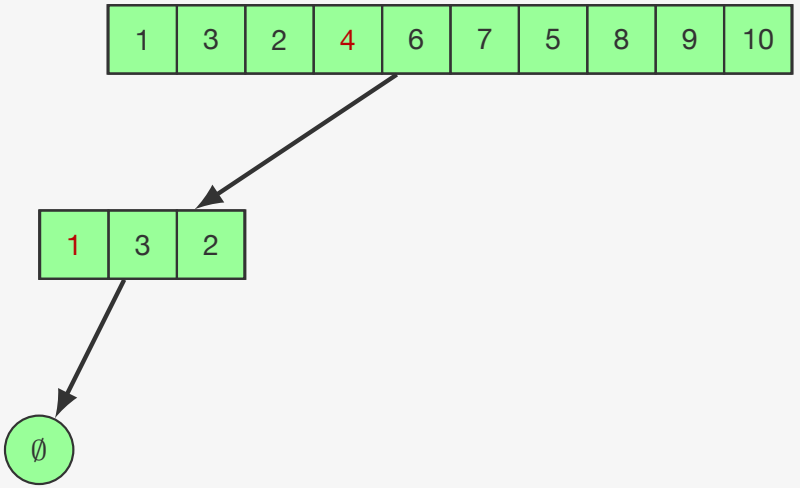
Simulação do Quicksort



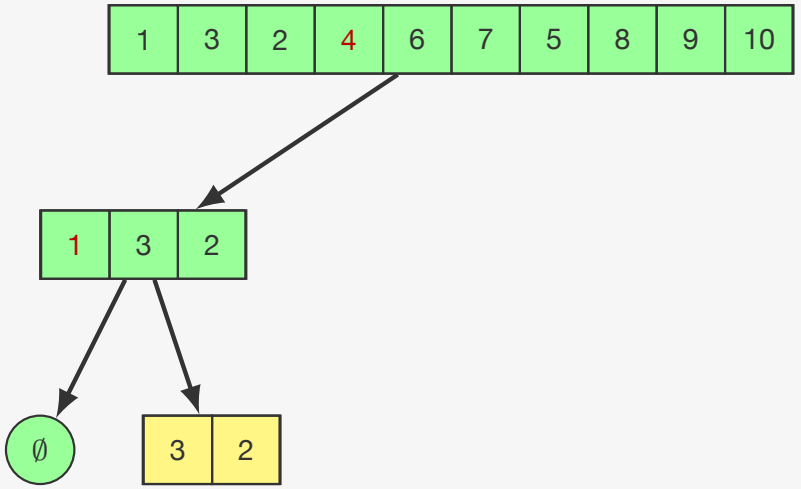
Simulação do Quicksort



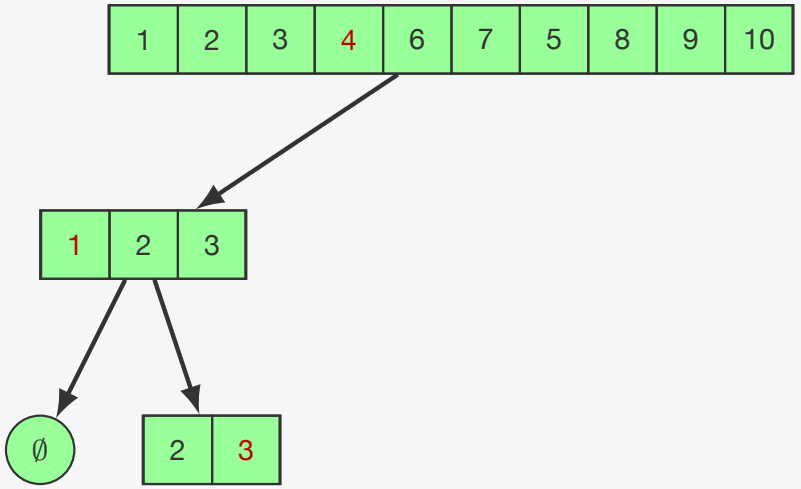
Simulação do Quicksort



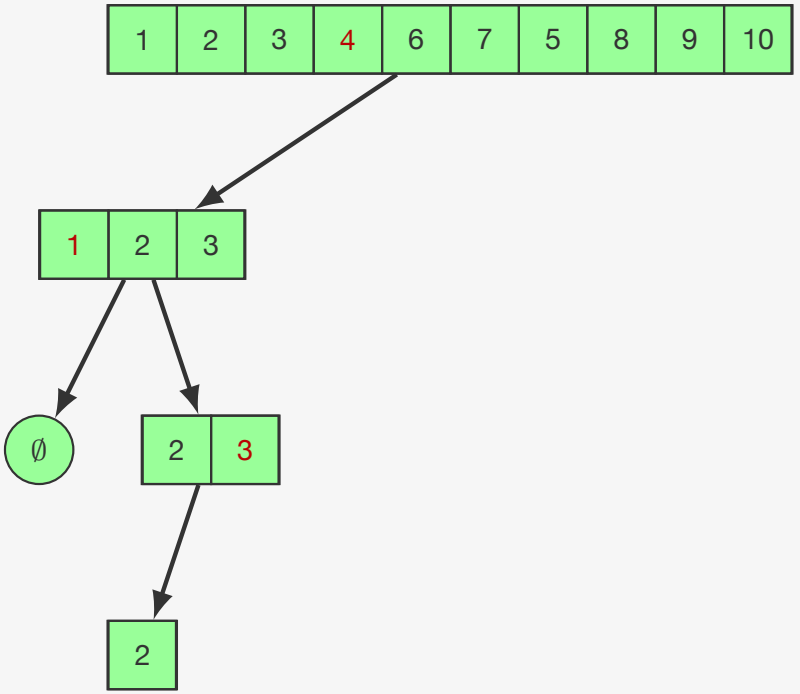
Simulação do Quicksort



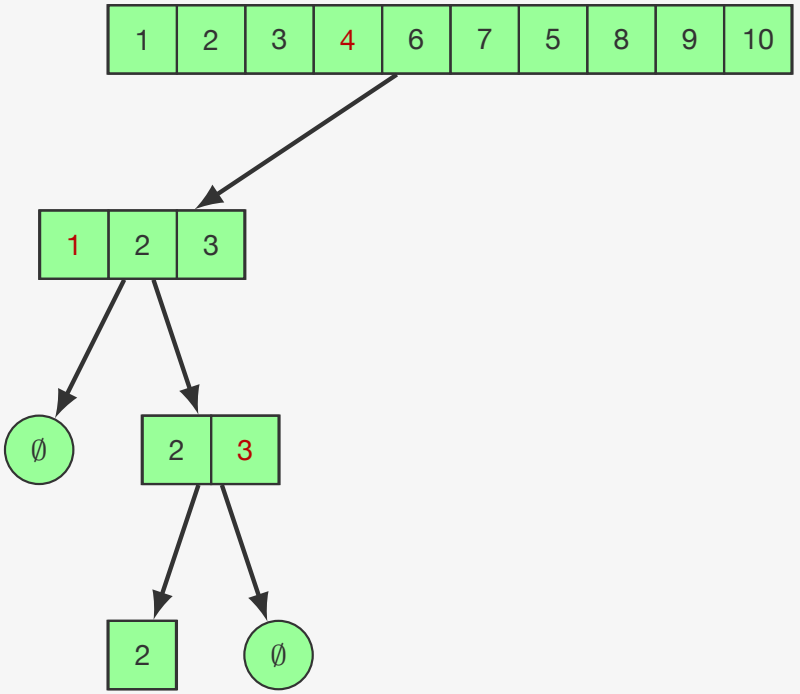
Simulação do Quicksort



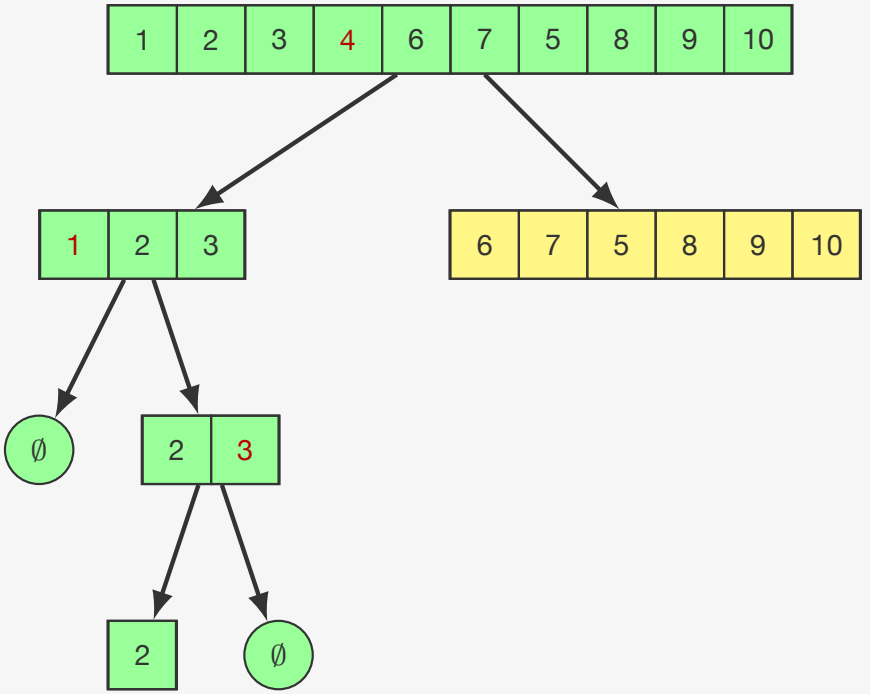
Simulação do Quicksort



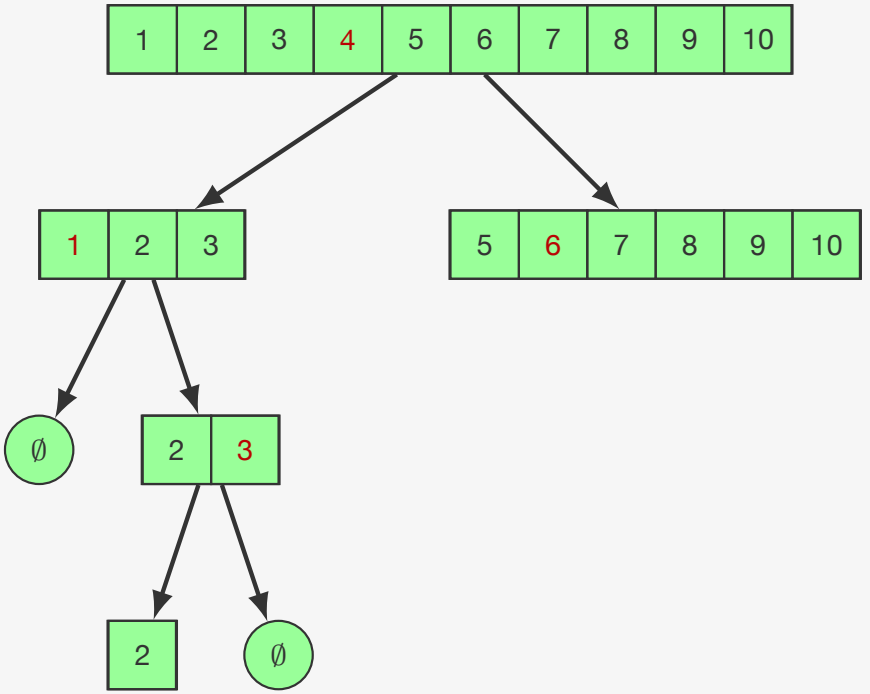
Simulação do Quicksort



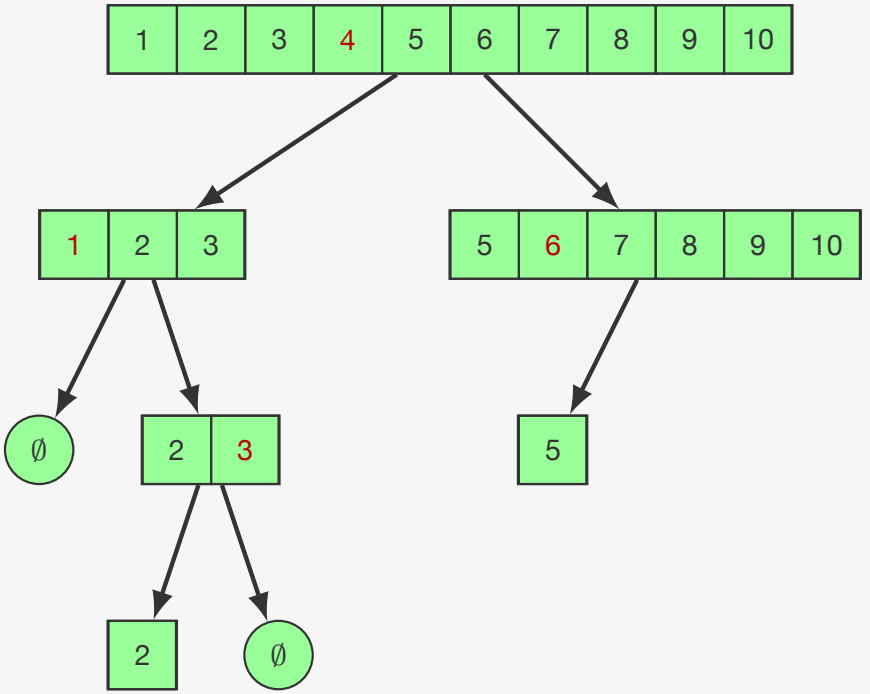
Simulação do Quicksort



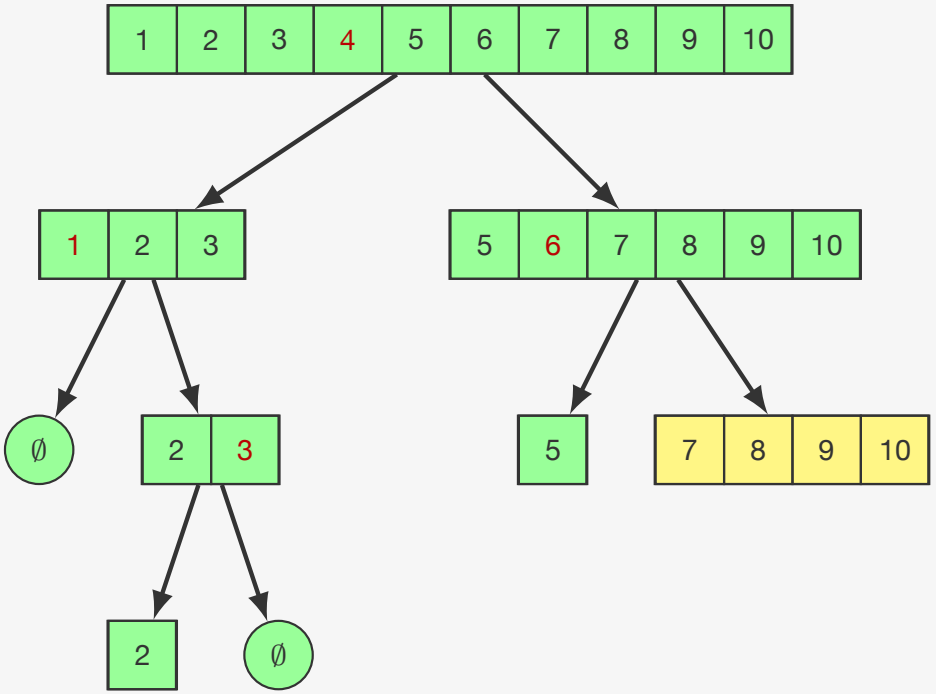
Simulação do Quicksort



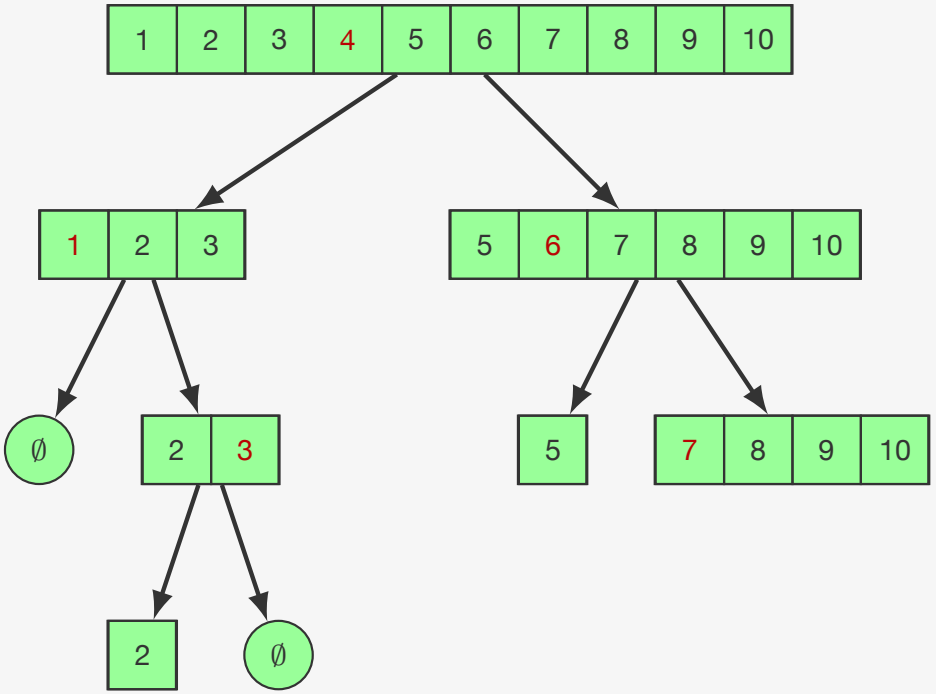
Simulação do Quicksort



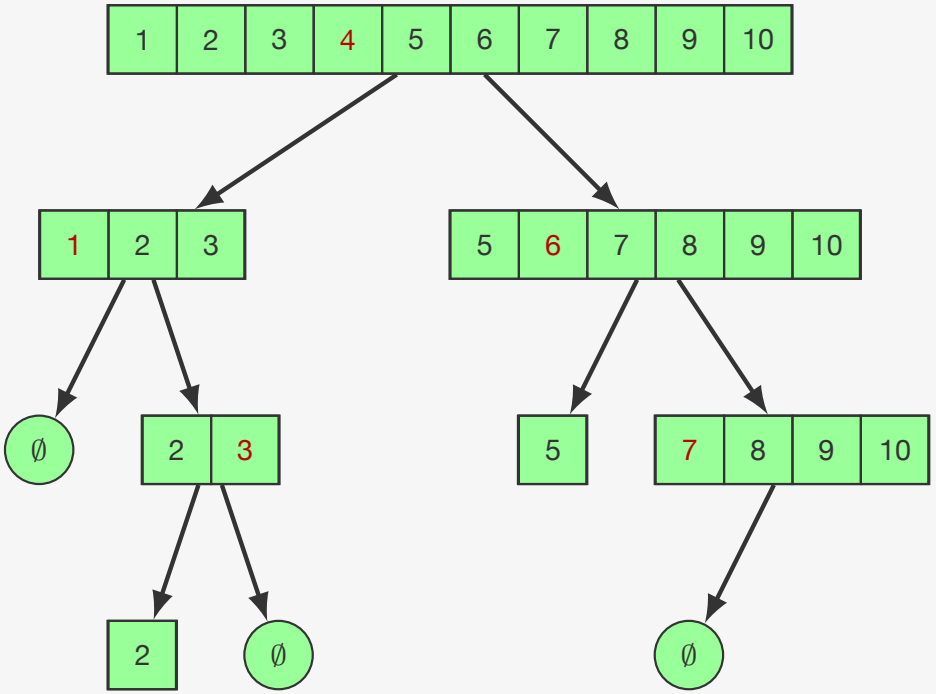
Simulação do Quicksort



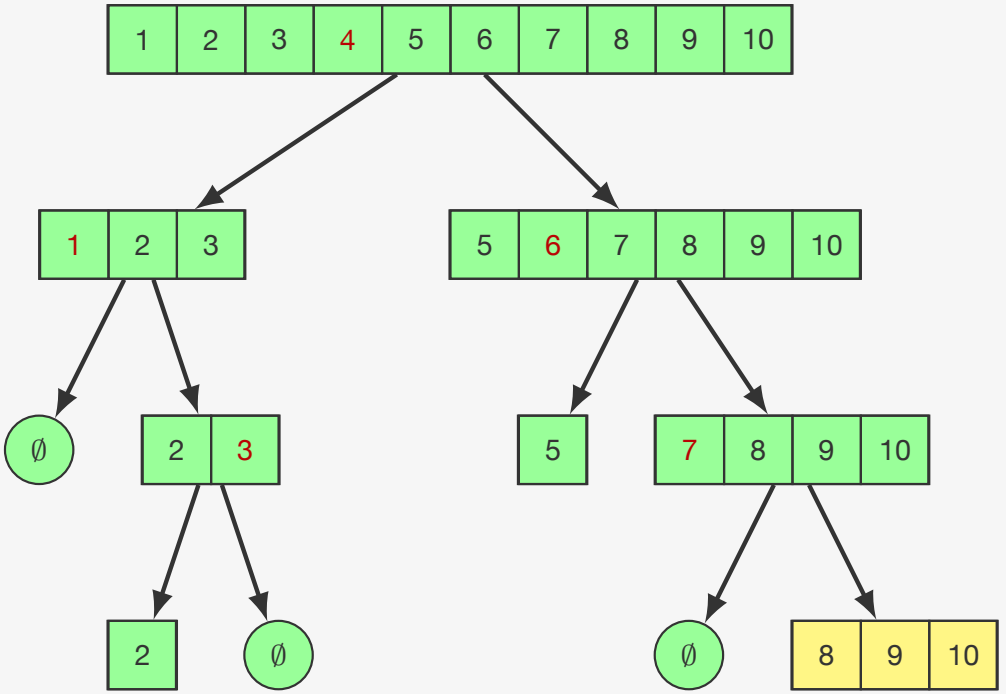
Simulação do Quicksort



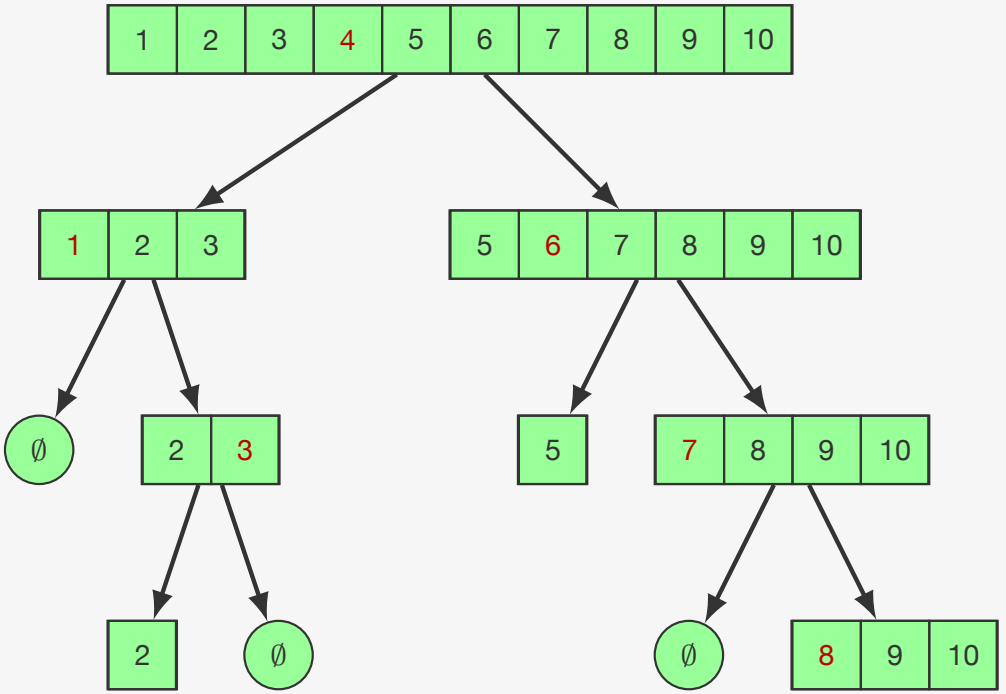
Simulação do Quicksort



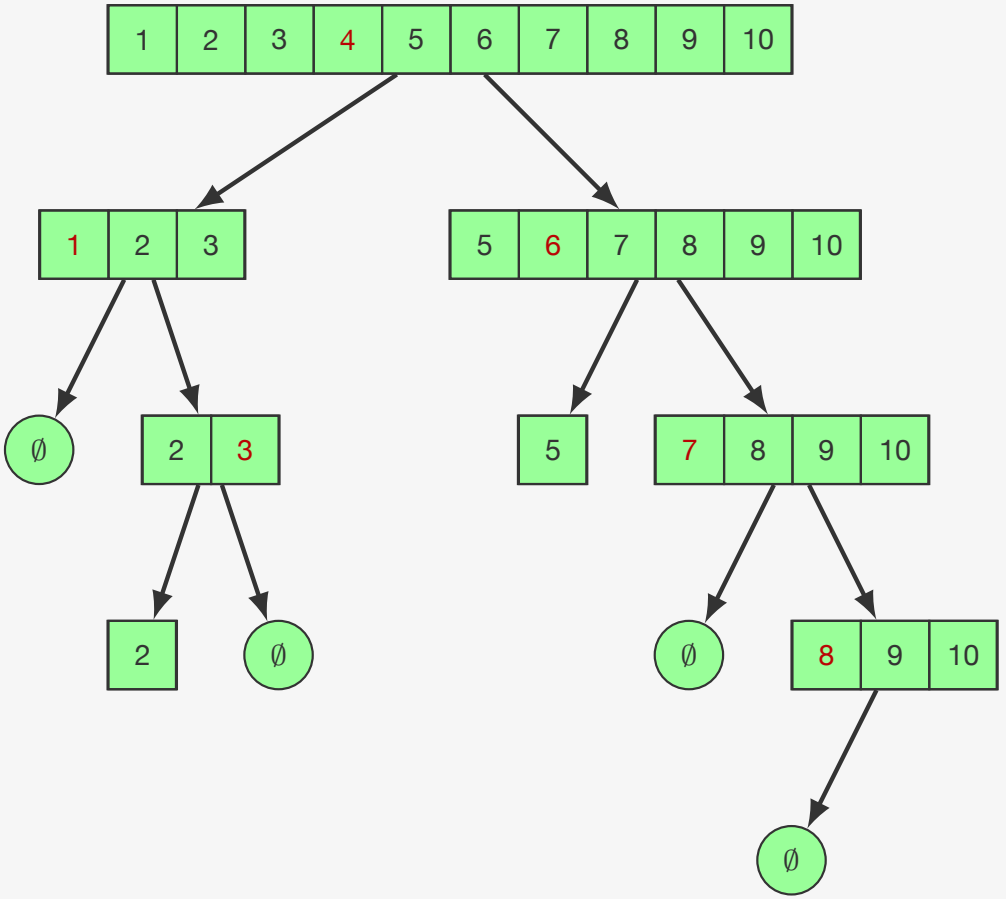
Simulação do Quicksort



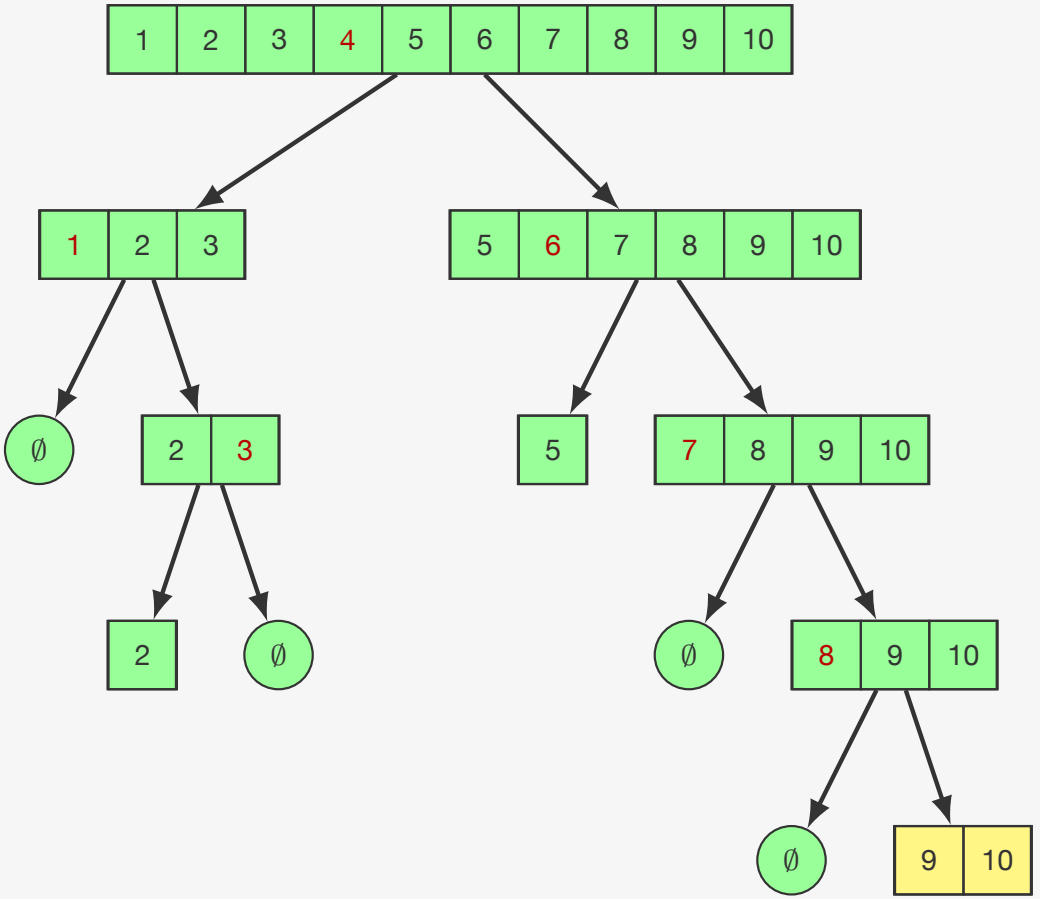
Simulação do Quicksort



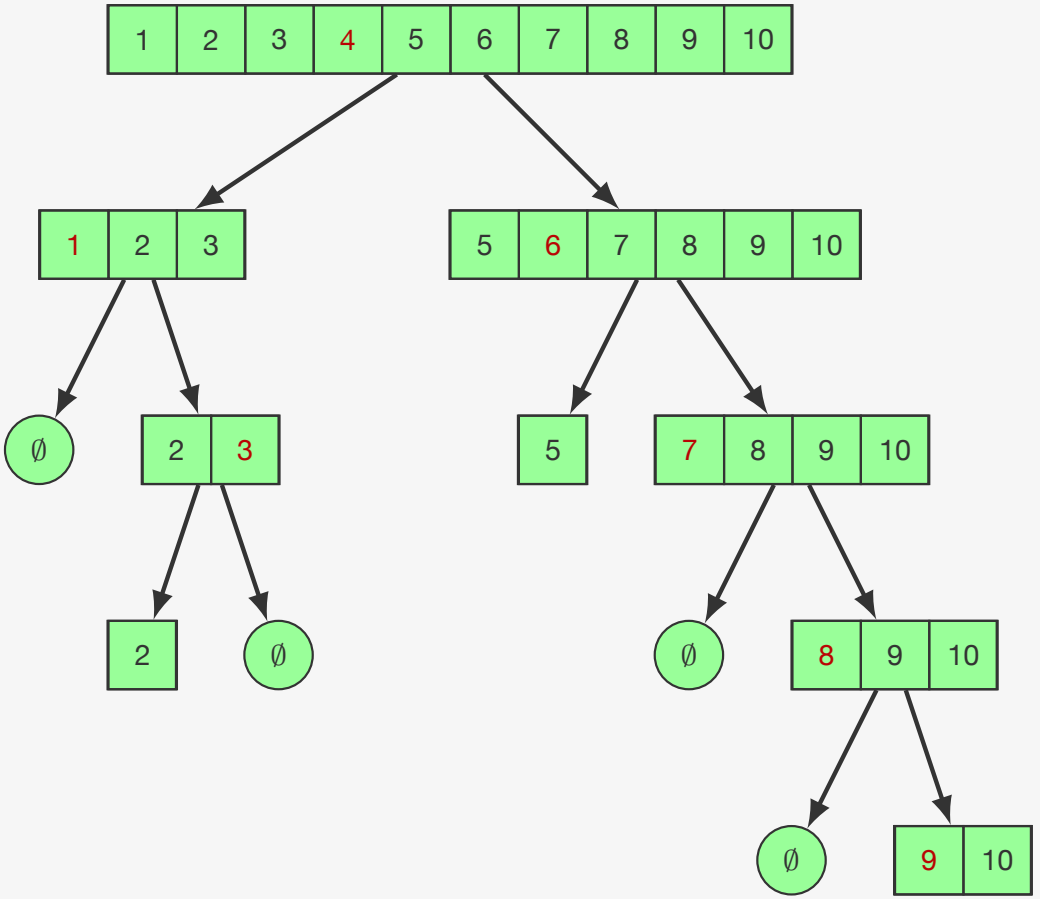
Simulação do Quicksort



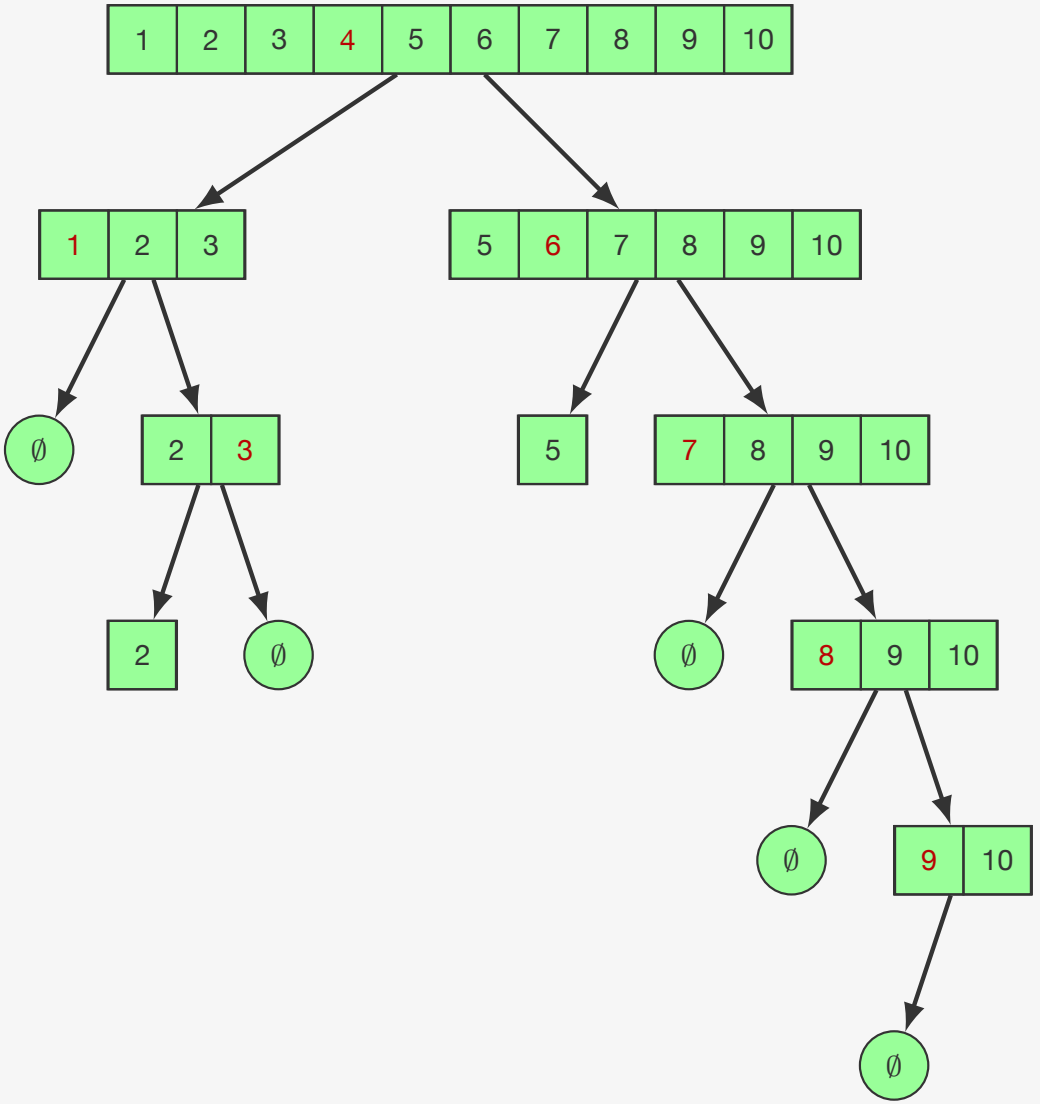
Simulação do Quicksort



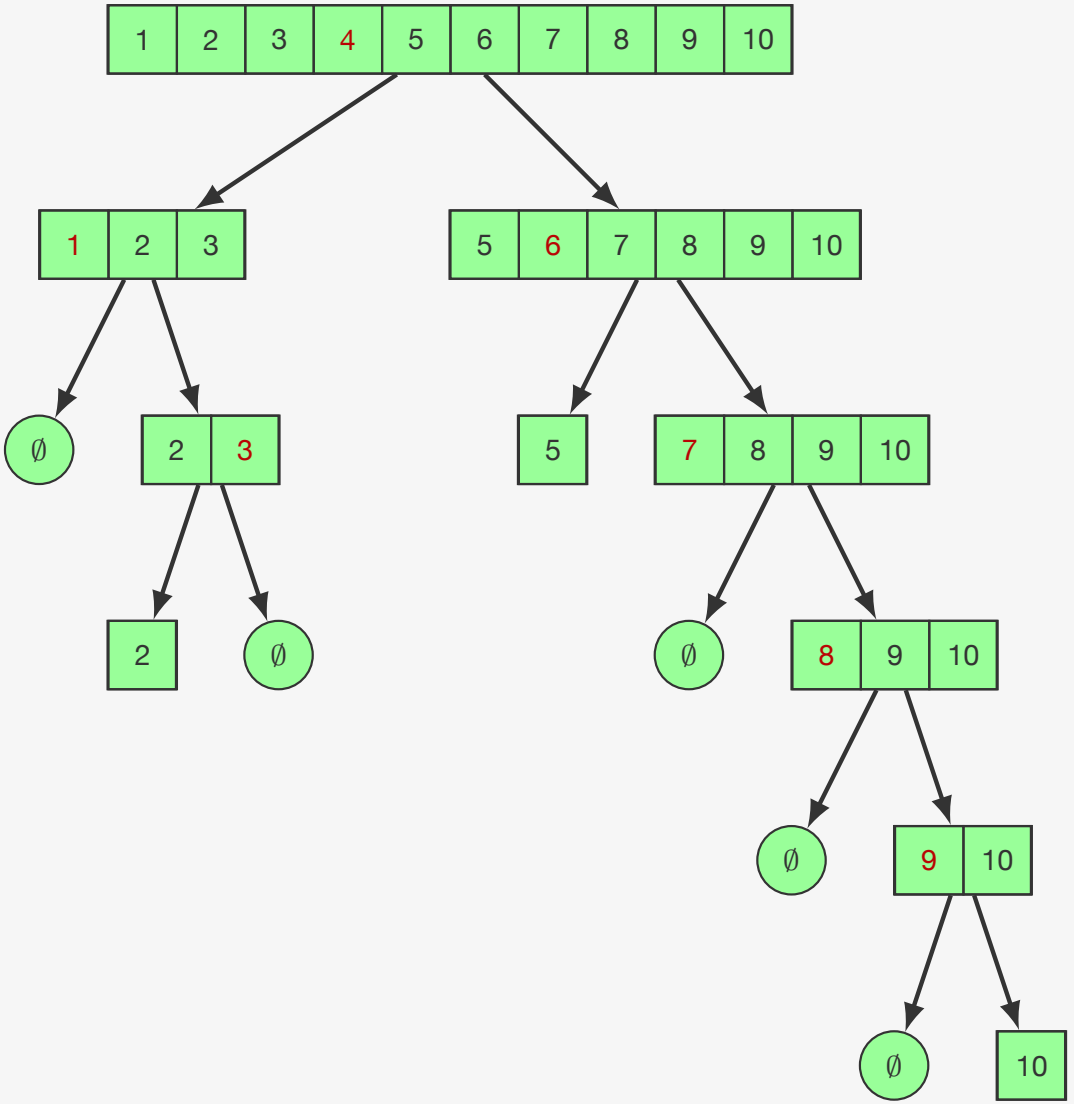
Simulação do Quicksort



Simulação do Quicksort



Simulação do Quicksort



Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

- cada elemento é movido no máximo uma vez

Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

- cada elemento é movido no máximo uma vez
  - ou está à direita e é menor que o pivô

Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

- cada elemento é movido no máximo uma vez
  - ou está à direita e é menor que o pivô
  - ou está à esquerda e é maior que o pivô

Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

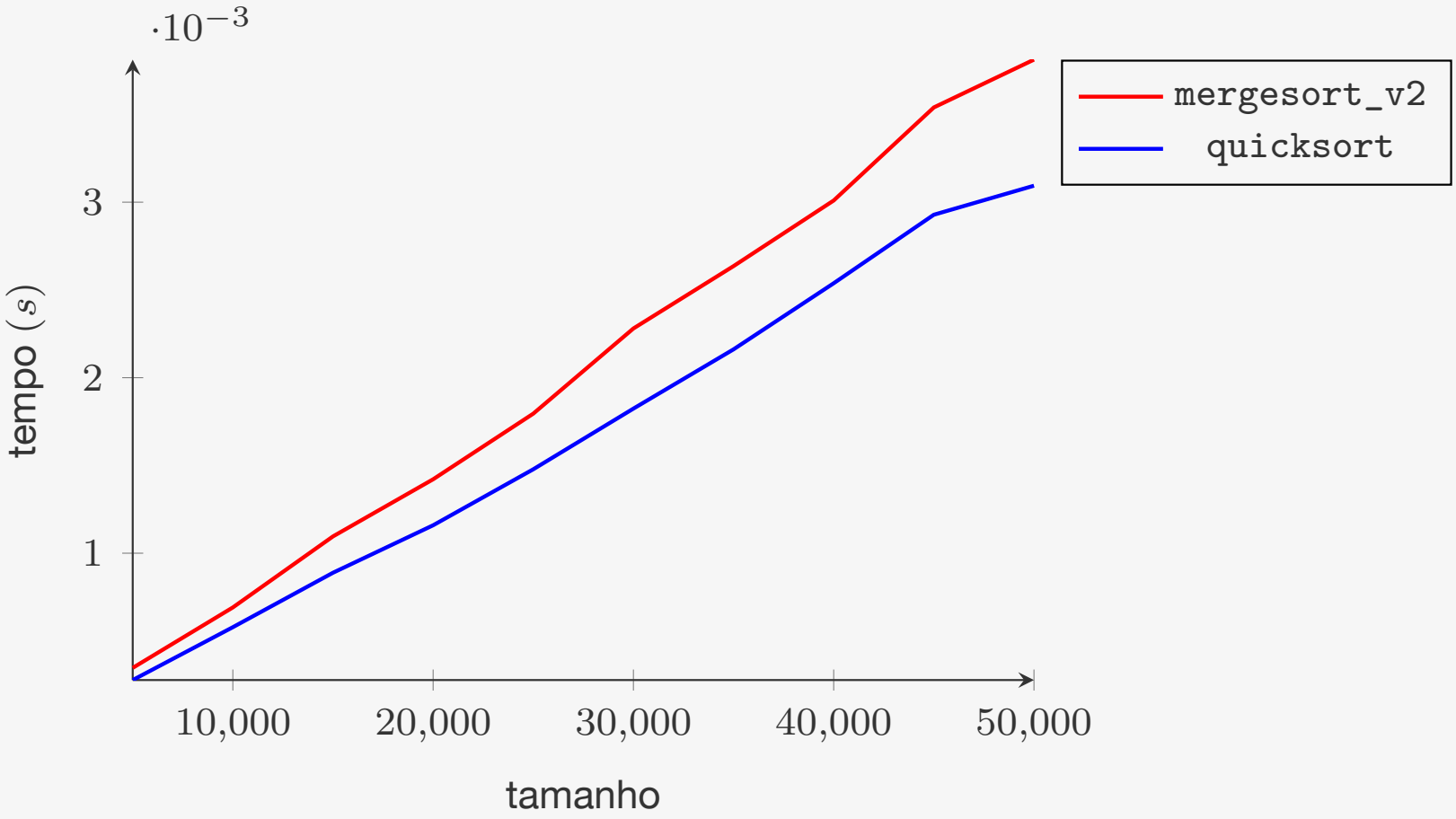
- cada elemento é movido no máximo uma vez
  - ou está à direita e é menor que o pivô
  - ou está à esquerda e é maior que o pivô
- $2(n-1)$ comparações e $n+1$ atribuições no máximo

Tempo do Particiona

```
 1 int partition(int *v, int l, int r) {
 2   int pivo = v[l];
 3   int i = l, f = r;
 4   while (i < f) {
 5     while (i < f && v[f] >= pivo)
 6       f--;
 7     if (i < f)
 8       v[i] = v[f];
 9     while (i < f && v[i] <= pivo)
10       i++;
11     if (i < f)
12       v[f] = v[i];
13   }
14   v[i] = pivo;
15   return i;
16 }
```

- cada elemento é movido no máximo uma vez
  - ou está à direita e é menor que o pivô
  - ou está à esquerda e é maior que o pivô
- $2(n-1)$ comparações e $n+1$ atribuições no máximo
- `partition` executa em tempo $\mathrm{O}(n)$

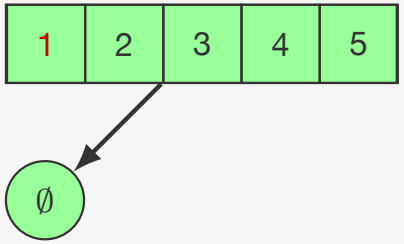
Comparação com o MergeSort



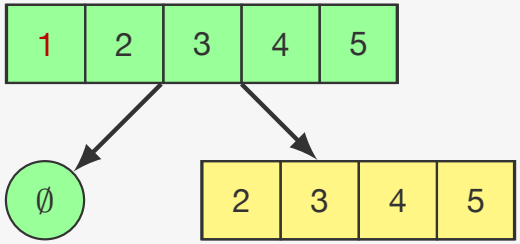
Pior caso do QuickSort

Pior caso do QuickSort

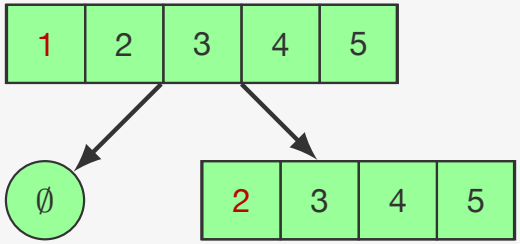
Pior caso do QuickSort



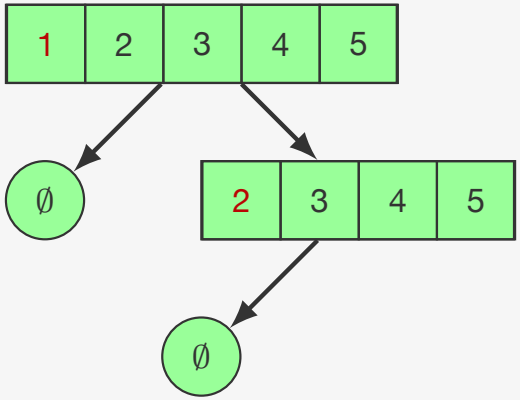
Pior caso do QuickSort



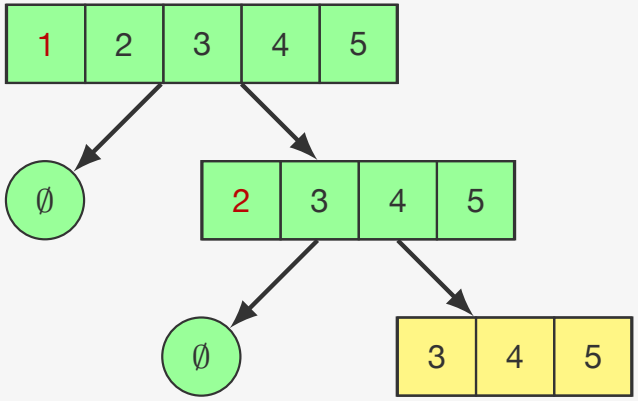
Pior caso do QuickSort



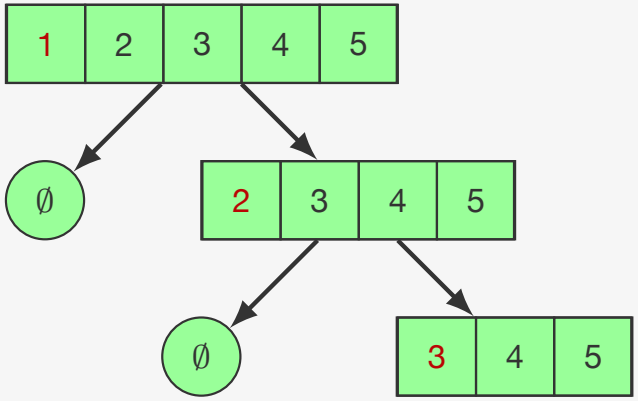
Pior caso do QuickSort



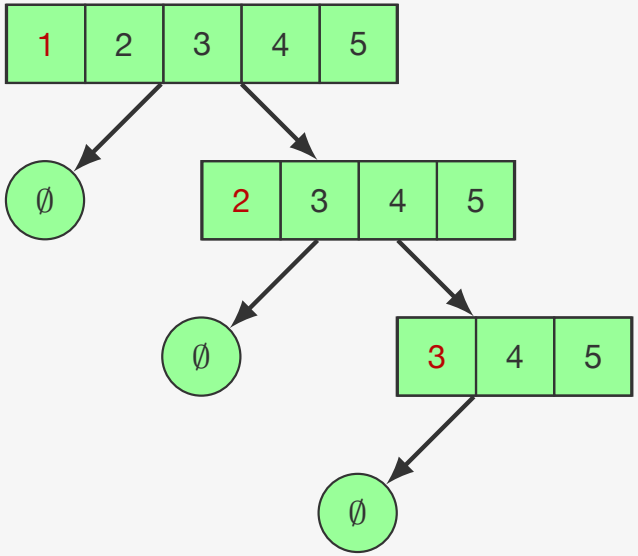
Pior caso do QuickSort



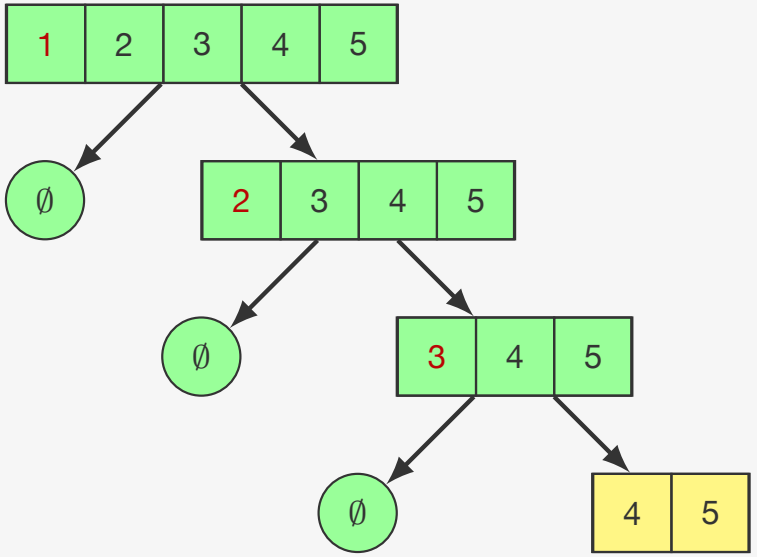
Pior caso do QuickSort



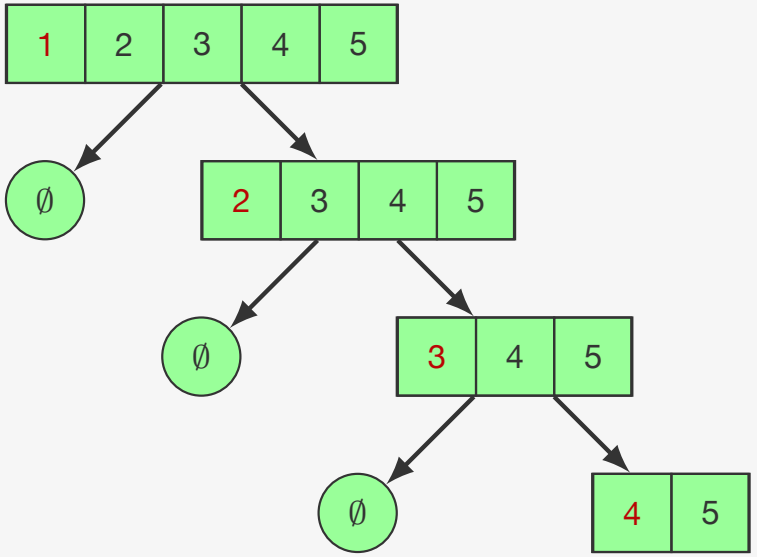
Pior caso do QuickSort



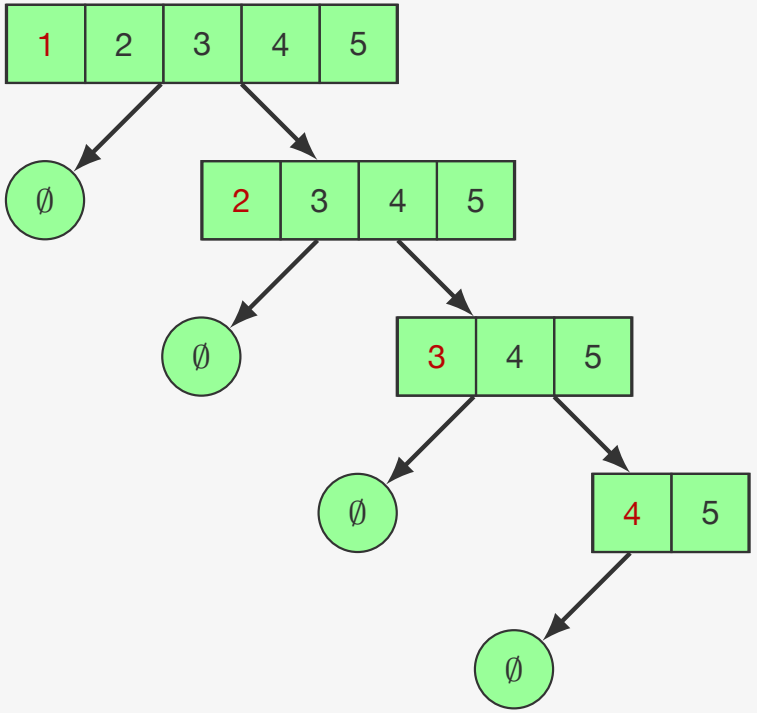
Pior caso do QuickSort



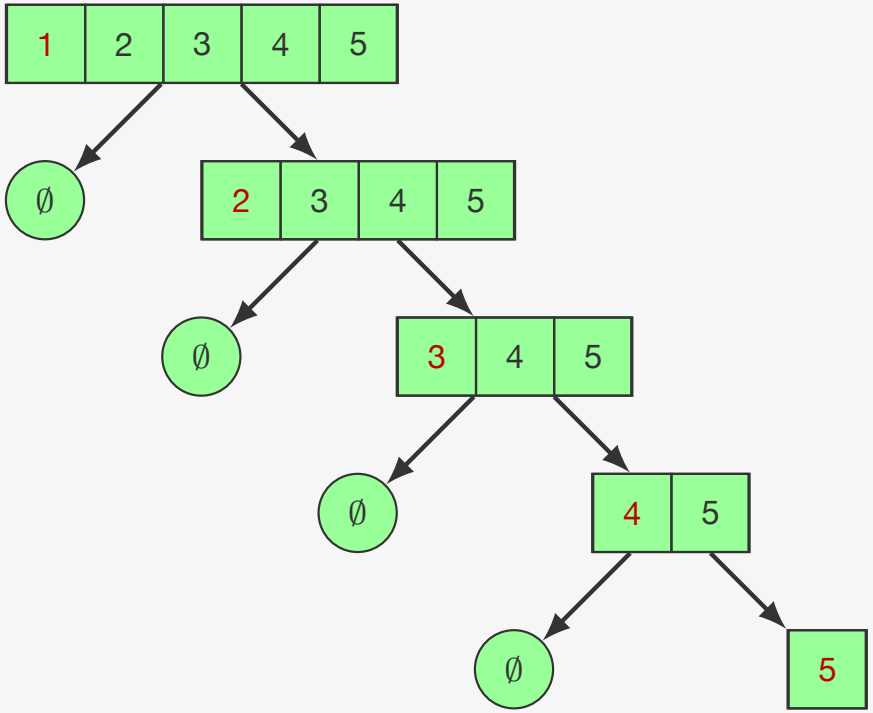
Pior caso do QuickSort



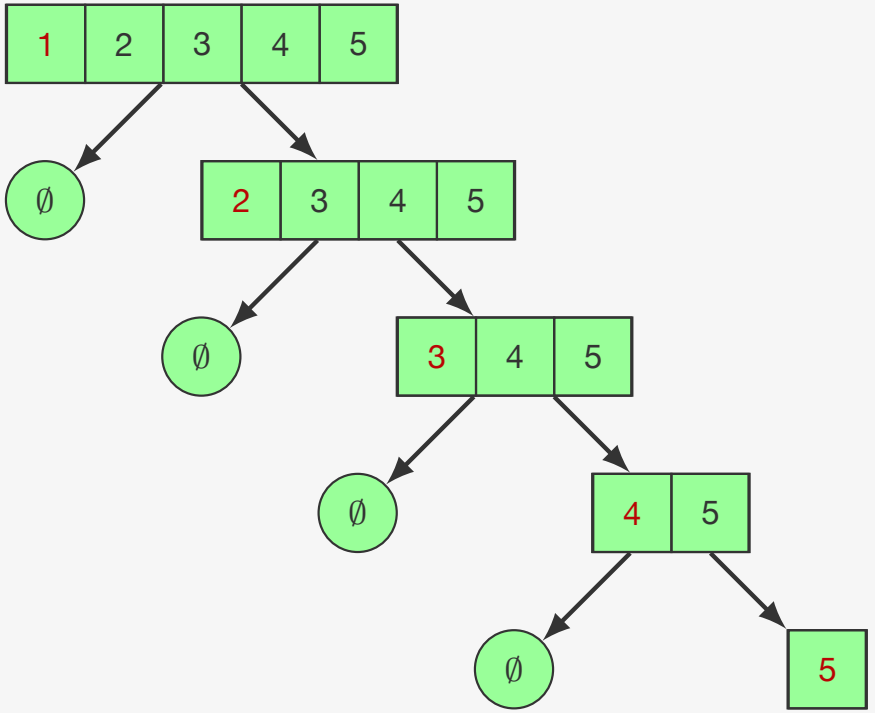
Pior caso do QuickSort



Pior caso do QuickSort

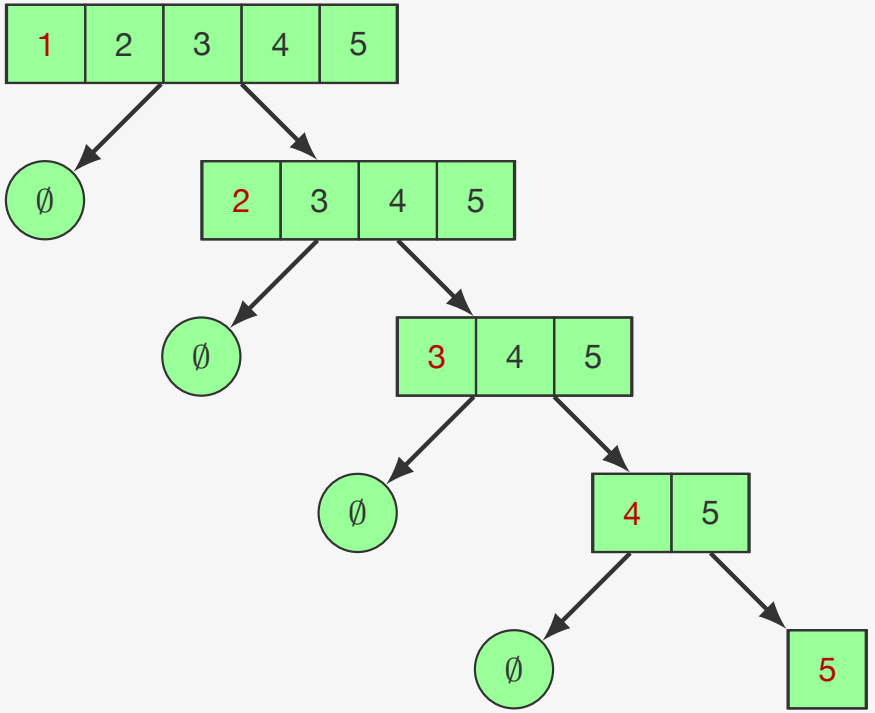


Pior caso do QuickSort



$c \cdot n$

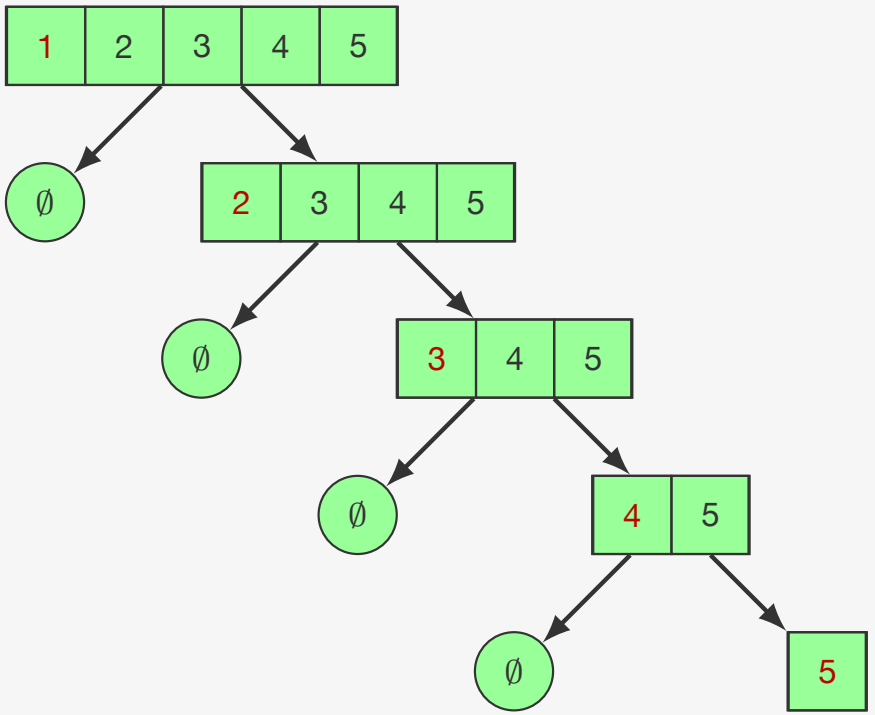
Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

Pior caso do QuickSort

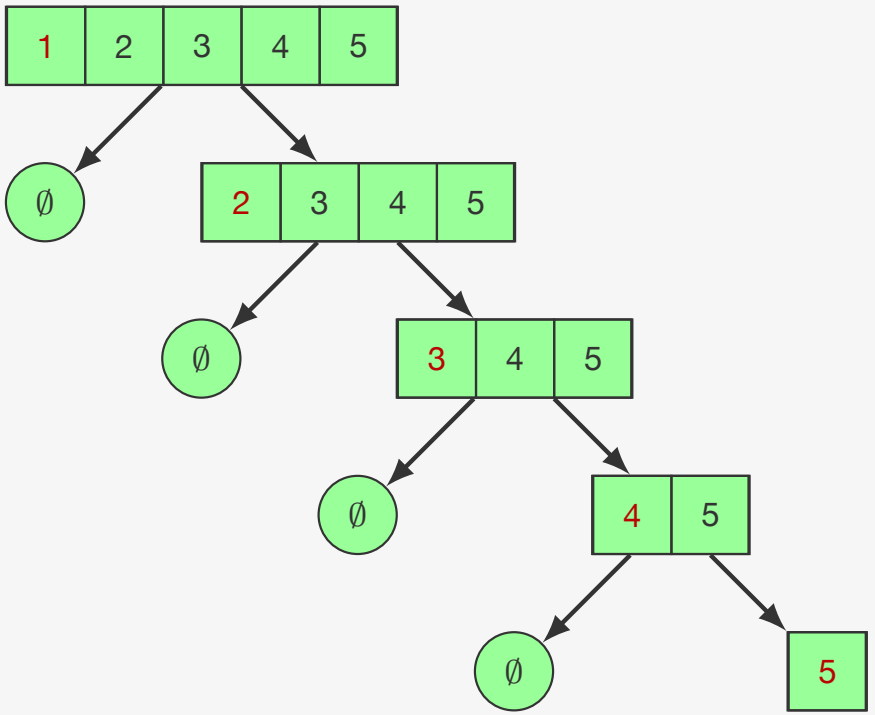


$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

Pior caso do QuickSort



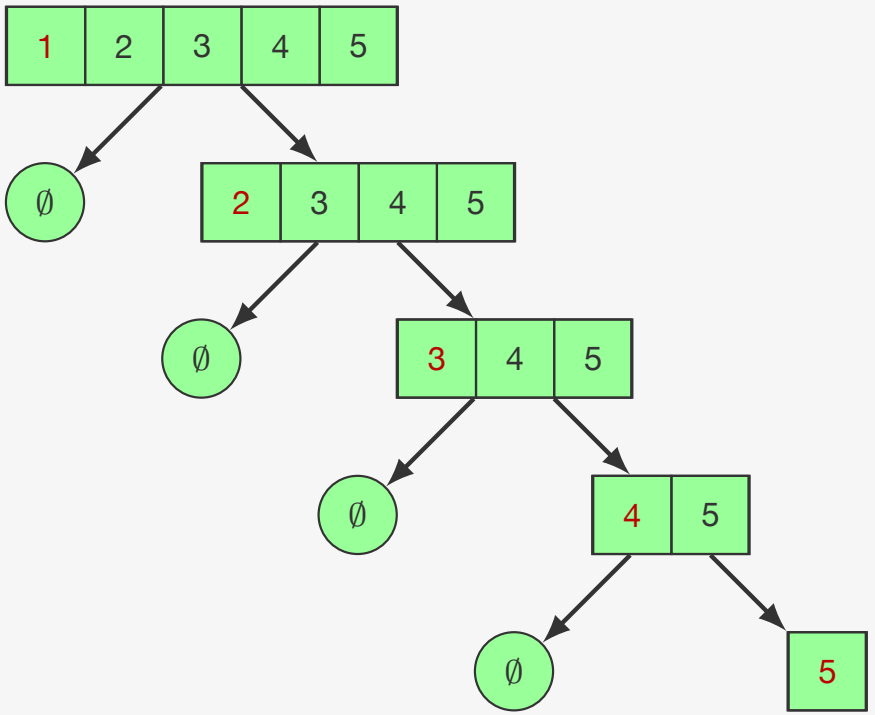
$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

Pior caso do QuickSort



$c \cdot n$

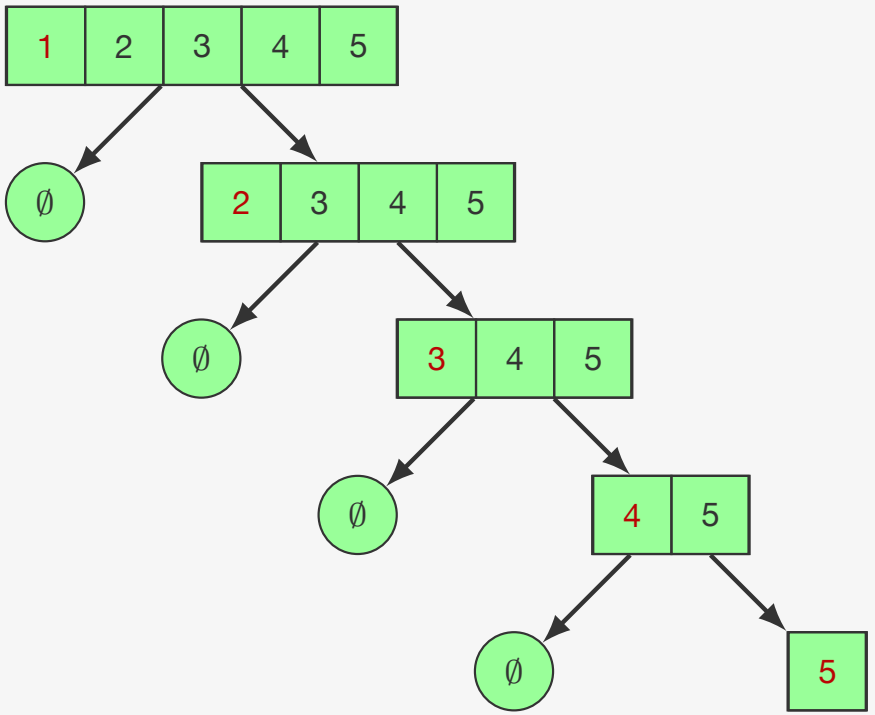
$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

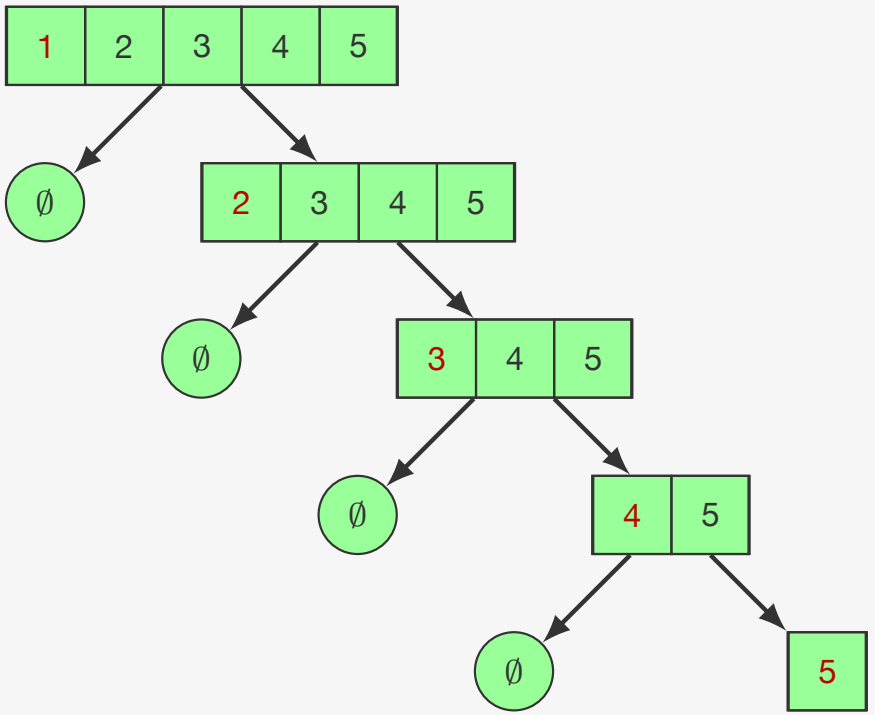
$c \cdot (n-2)$

$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

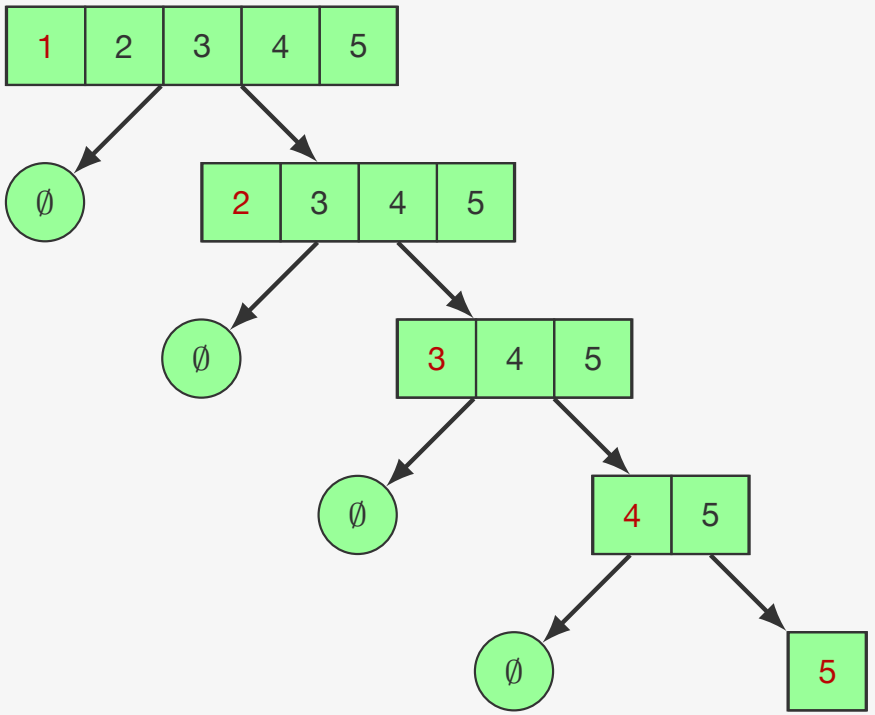
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$c \cdot n + c \cdot (n-1) + \cdots + c$

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

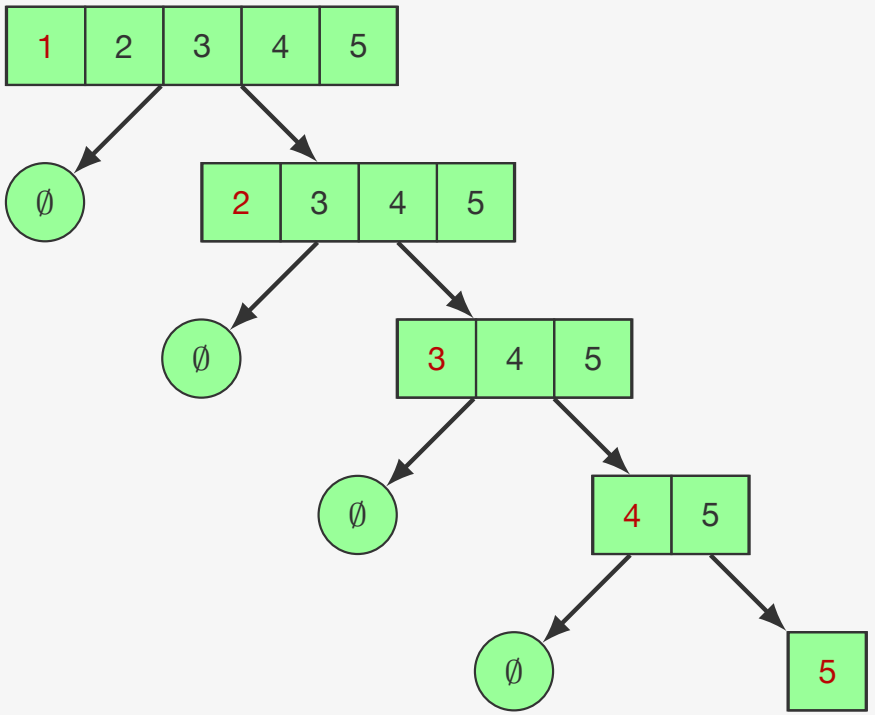
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{i=0}^{n-1} (n-i)$$

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

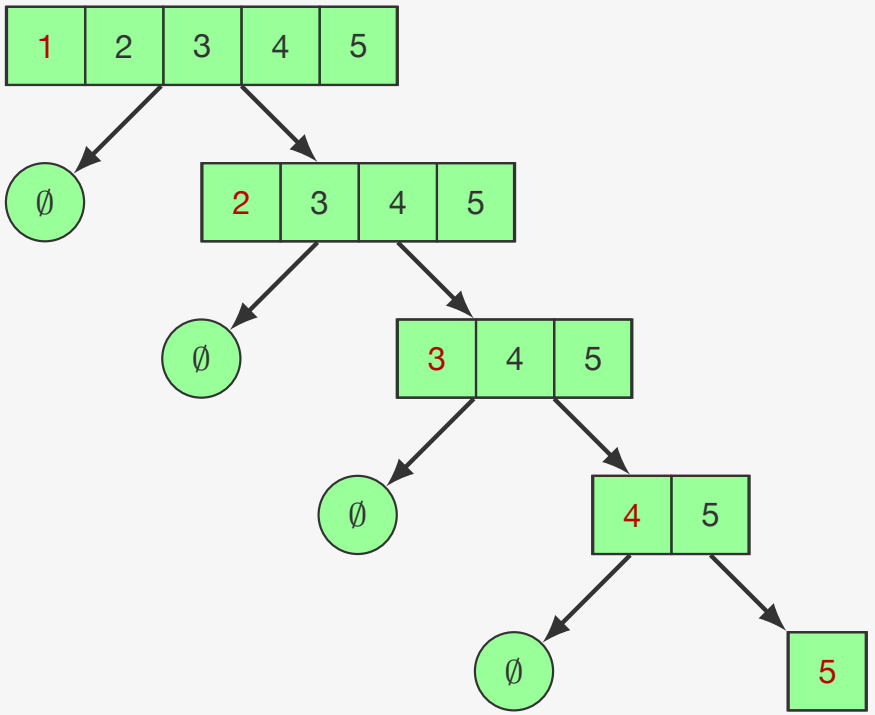
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c{\cdot}n{+}c{\cdot}(n{-}1){+}\cdots{+}c = c\sum_{i=0}^{n-1}(n-i) = c\sum_{j=1}^{n} j$$

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

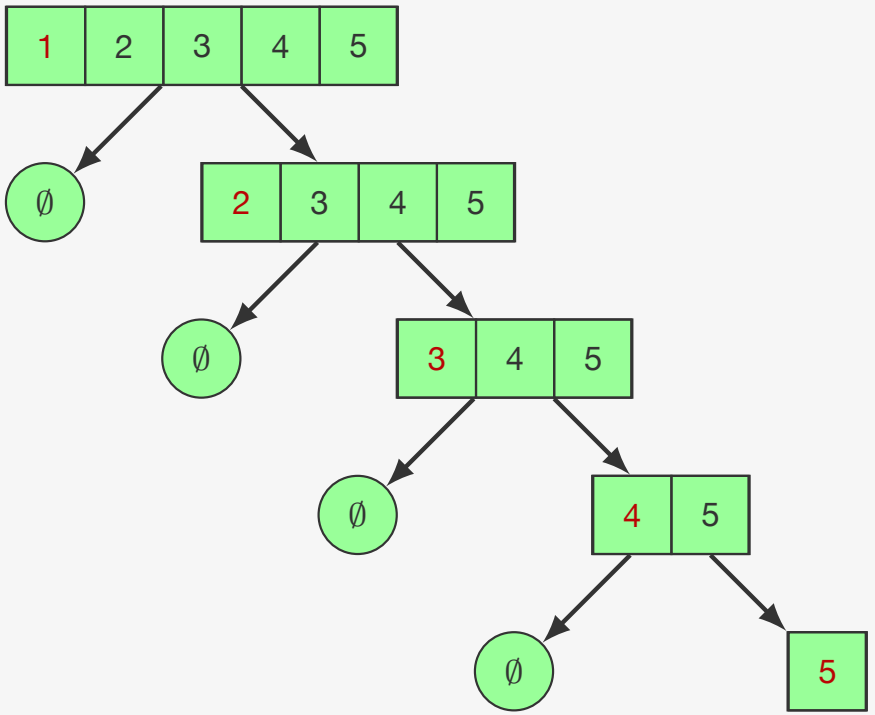
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c{\cdot}n{+}c{\cdot}(n{-}1){+}\cdots{+}c = c\sum_{i=0}^{n-1}(n-i) = c\sum_{j=1}^{n} j = c\frac{n(n+1)}{2}$$

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c{\cdot}n{+}c{\cdot}(n{-}1){+}\cdots{+}c = c\sum_{i=0}^{n-1}(n-i) = c\sum_{j=1}^{n} j = c\frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é “raro” nesse experimento

- Isso nem sempre é verdade

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é "raro" nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é "raro" nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados
  - exemplo: inserção em blocos em um vetor ordenado

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o MergeSort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é "raro" nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados
  - exemplo: inserção em blocos em um vetor ordenado

Vamos ver duas formas de mitigar esse problema

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três
  - já que a mediana do vetor particiona ele no meio

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três
  - já que a mediana do vetor particiona ele no meio

```
 1 void quicksort_mdt(int *v, int l, int r) {
 2   int i;
 3   if(r <= l) return;
 4   troca(&v[(l+r)/2], &v[l+1]);
 5   if(v[l] > v[l+1])
 6     troca(&v[l], &v[l+1]);
 7   if(v[l] > v[r])
 8     troca(&v[l], &v[r]);
 9   if(v[l+1] > v[r])
10     troca(&v[l+1], &v[r]);
11   i = partition(v, l+1, r-1);
12   quicksort_mdt(v, l, i-1);
13   quicksort_mdt(v, i+1, r);
14 }
```

- trocamos `v[(l+r)/2]` com `v[l+1]`
- ordenamos `v[l]`, `v[l+1]` e `v[r]`
- particionamos `v[l+1]`, $\cdots$, `v[r-1]`
  - `v[l]` já é menor que o pivô
  - `v[r]` já é maior que o pivô

Quicksort Aleatorizado

Quicksort Aleatorizado

```
 1 int pivo_aleatorio(int l, int r) {
 2   return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
 3 }
 4
 5 void quicksort_ale(int *v, int l, int r) {
 6   int i;
 7   if(r <= l) return;
 8   troca(&v[pivo_aleatorio(l,r)], &v[l]);
 9   i = partition(v, l, r);
10   quicksort_ale(v, l, i-1);
11   quicksort_ale(v, i+1, r);
12 }
```

Quicksort Aleatorizado

```
 1 int pivo_aleatorio(int l, int r) {
 2   return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
 3 }
 4
 5 void quicksort_ale(int *v, int l, int r) {
 6   int i;
 7   if(r <= l) return;
 8   troca(&v[pivo_aleatorio(l,r)], &v[l]);
 9   i = partition(v, l, r);
10   quicksort_ale(v, l, i-1);
11   quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

Quicksort Aleatorizado

```
 1 int pivo_aleatorio(int l, int r) {
 2   return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
 3 }
 4
 5 void quicksort_ale(int *v, int l, int r) {
 6   int i;
 7   if(r <= l) return;
 8   troca(&v[pivo_aleatorio(l,r)], &v[l]);
 9   i = partition(v, l, r);
10   quicksort_ale(v, l, i-1);
11   quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

- O tempo médio é $\mathrm{O}(n \lg n)$

Quicksort Aleatorizado

```
 1 int pivo_aleatorio(int l, int r) {
 2   return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
 3 }
 4
 5 void quicksort_ale(int *v, int l, int r) {
 6   int i;
 7   if(r <= l) return;
 8   troca(&v[pivo_aleatorio(l,r)], &v[l]);
 9   i = partition(v, l, r);
10   quicksort_ale(v, l, i-1);
11   quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

- O tempo médio é $\mathrm{O}(n \lg n)$
  - as vezes é lento, as vezes é rápido

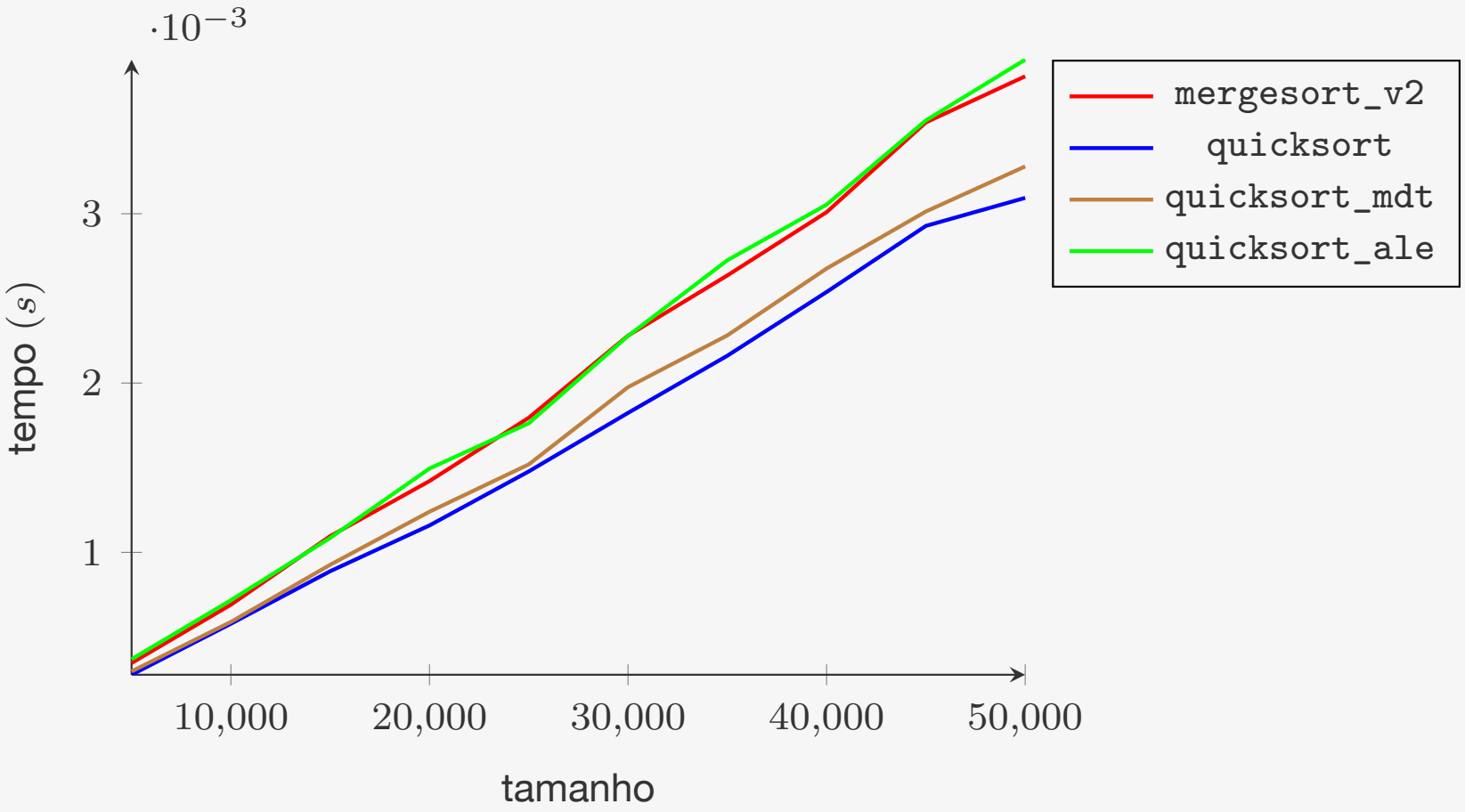
Quicksort Aleatorizado

```
 1 int pivo_aleatorio(int l, int r) {
 2   return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
 3 }
 4
 5 void quicksort_ale(int *v, int l, int r) {
 6   int i;
 7   if(r <= l) return;
 8   troca(&v[pivo_aleatorio(l,r)], &v[l]);
 9   i = partition(v, l, r);
10   quicksort_ale(v, l, i-1);
11   quicksort_ale(v, i+1, r);
12 }
```

O tempo de execução depende dos pivôs sorteados

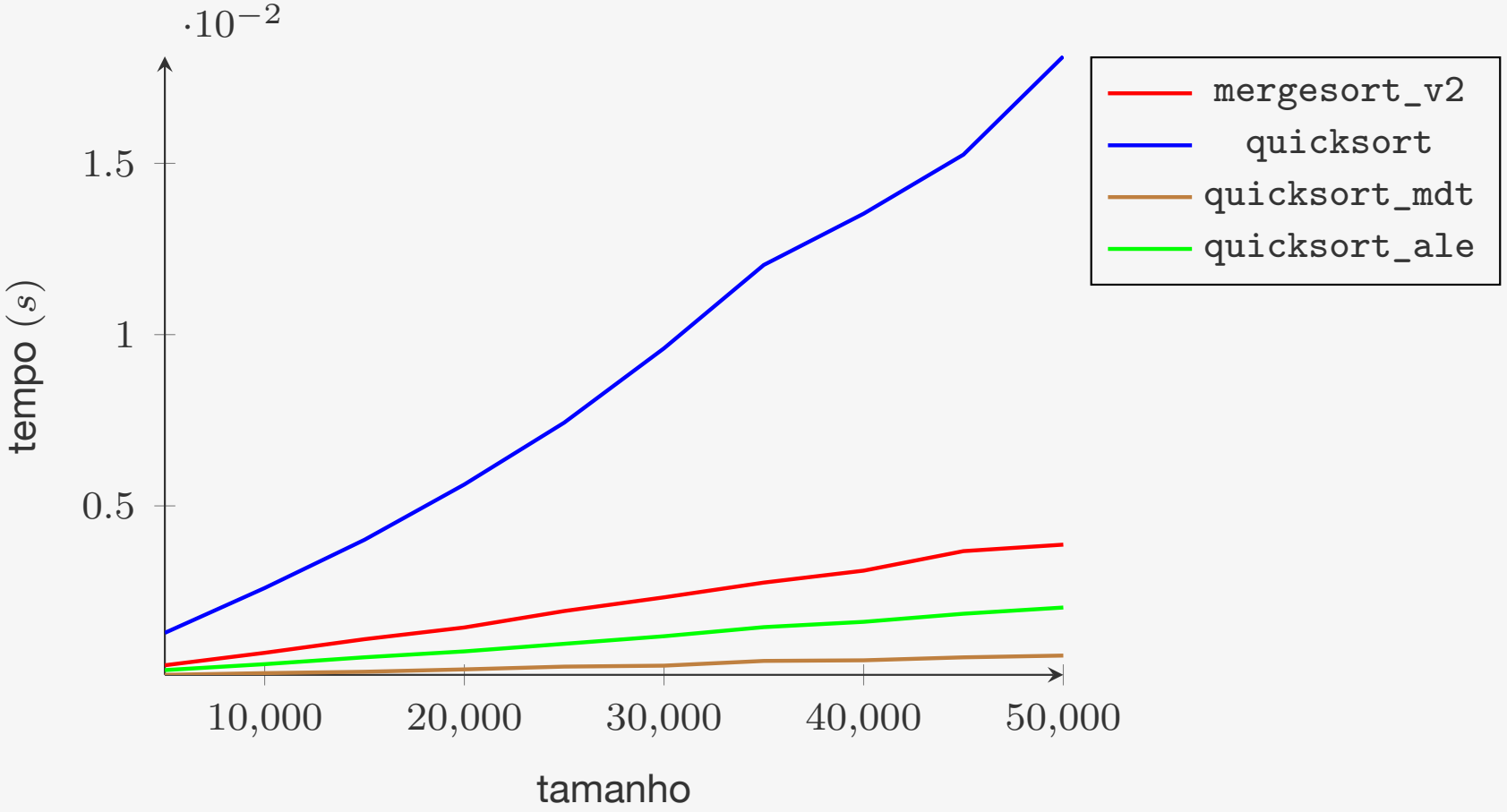
- O tempo médio é $\mathrm{O}(n \lg n)$
  - as vezes é lento, as vezes é rápido
  - mas não depende do vetor dado

Experimentos - Vetores aleatórios



Quando o vetor é aleatório, `quicksort_mdt` e `quicksort_ale` adicionam um overhead desnecessário

Experimentos - Vetores quase ordenados



$0{,}5\%$ de trocas entre pares feitas de maneira aleatória

`quicksort_mdt` é melhor mesmo sendo $\mathrm{O}(n^2)$ e `quicksort_ale` sendo em média $\mathrm{O}(n \lg n)$

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é “trabalhosa”

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é “trabalhosa”
- utiliza recursão

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é "trabalhosa"
- utiliza recursão

O overhead é diluído se o vetor é grande

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é “trabalhosa”
- utiliza recursão

O overhead é diluído se o vetor é grande

- Mas e quando o vetor é pequeno?

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é “trabalhosa”
- utiliza recursão

O overhead é diluído se o vetor é grande

- Mas e quando o vetor é pequeno?

O InsertionSort é bem rápido para vetores muito pequenos

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é “trabalhosa”
- utiliza recursão

O overhead é diluído se o vetor é grande

- Mas e quando o vetor é pequeno?

O InsertionSort é bem rápido para vetores muito pequenos

- Apesar de estarmos interessados em vetores grandes

Overhead para vetores pequenos

O QuickSort é um algoritmo elaborado:

- precisa realizar a partição que é "trabalhosa"
- utiliza recursão

O overhead é diluído se o vetor é grande

- Mas e quando o vetor é pequeno?

O InsertionSort é bem rápido para vetores muito pequenos

- Apesar de estarmos interessados em vetores grandes
- Ordenamos muitos vetores pequenos recursivamente

Lidando com vetores pequenos

Se o vetor for pequeno, chama o InsertionSort

```
1  #define M 10
2
3  void quicksort_mdt_v2(int *v, int l, int r) {
4    int i;
5    if (r - l <= M)
6      insertionsort_v4(v, l, r);
7    else {
8      troca(v[(l+r)/2], v[l+1]);
9      if(v[l] > v[l+1]) troca(v[l], v[l+1]);
10     if(v[l] > v[r]) troca(v[l], v[r]);
11     if(v[l+1] > v[r]) troca(v[l+1], v[r]);
12     i = partition(v, l+1, r-1);
13     quicksort_mdt_v2(v, l, i-1);
14     quicksort_mdt_v2(v, i+1, r);
15   }
16 }
```

Lidando com vetores pequenos - Outra opção

- Deixamos o vetor quase ordenado usando o QuickSort

Lidando com vetores pequenos - Outra opção

- Deixamos o vetor quase ordenado usando o QuickSort
  - pedaços de tamanho no máximo `M+1` sem ordenar

Lidando com vetores pequenos - Outra opção

- Deixamos o vetor quase ordenado usando o QuickSort
  - pedaços de tamanho no máximo `M+1` sem ordenar
- Ordenamos o vetor usando InsertionSort

Lidando com vetores pequenos - Outra opção

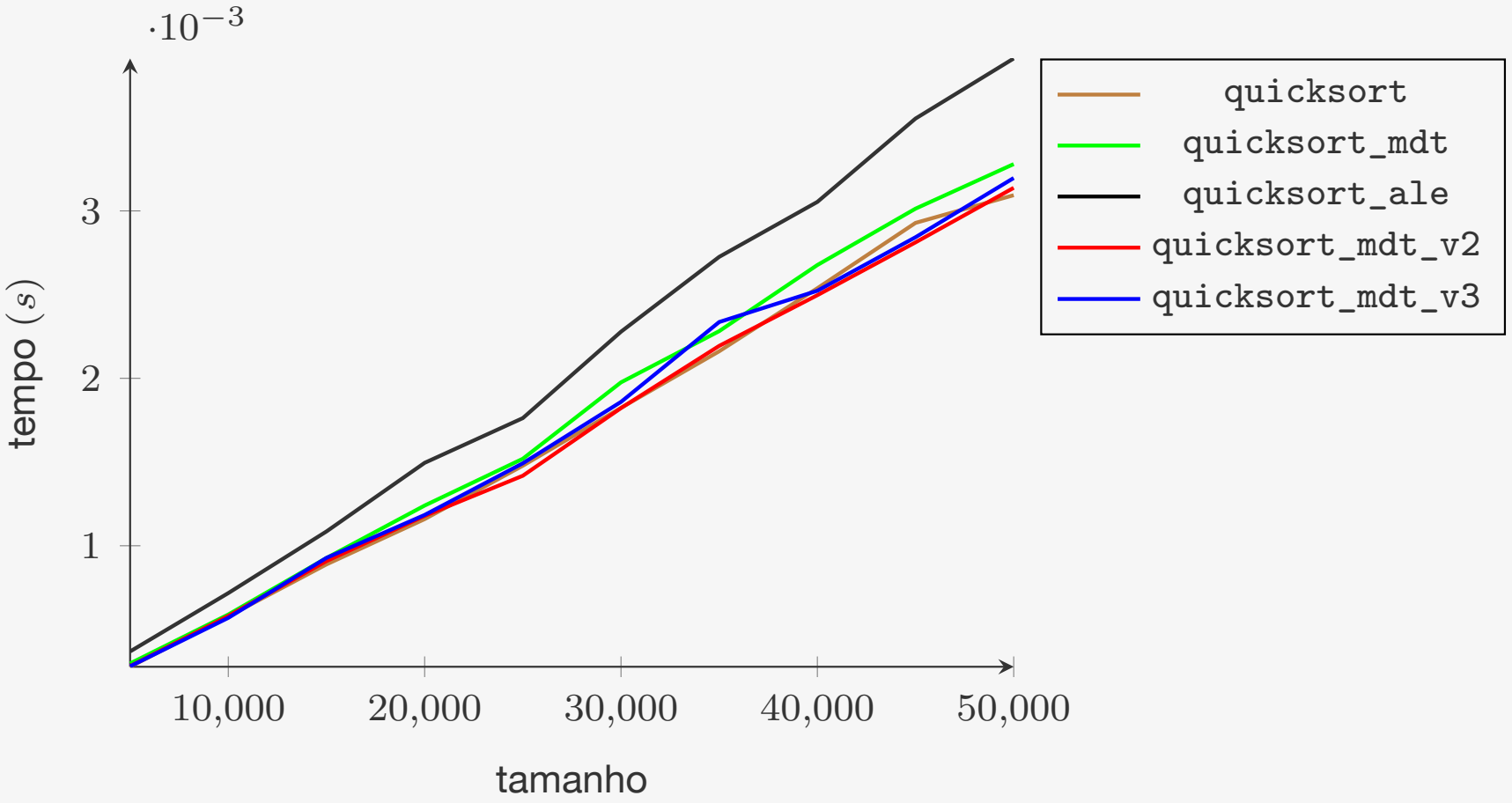
- Deixamos o vetor quase ordenado usando o QuickSort
  - pedaços de tamanho no máximo `M+1` sem ordenar
- Ordenamos o vetor usando InsertionSort
  - Ele é rápido para vetor quase ordenado

Lidando com vetores pequenos - Outra opção

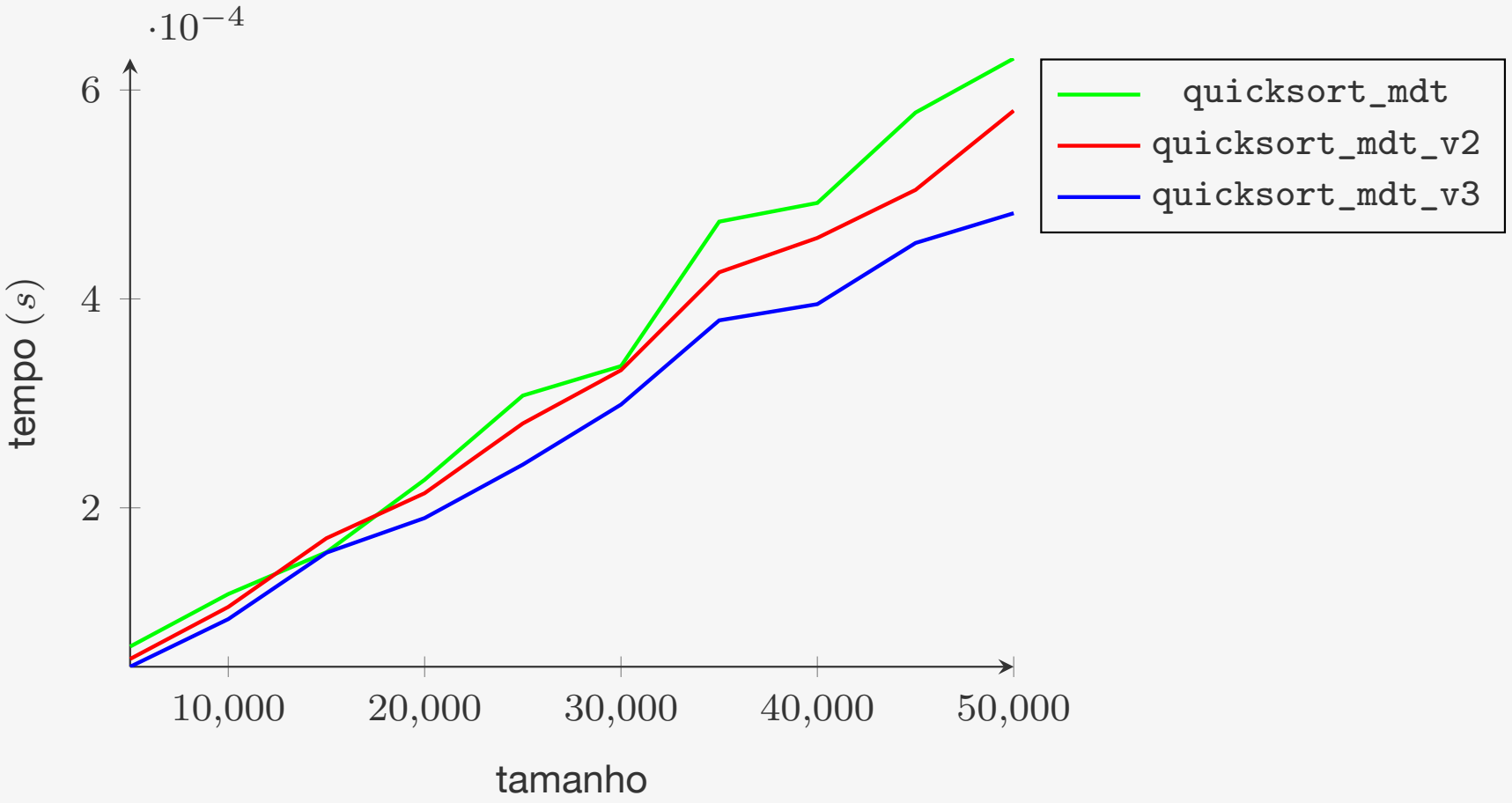
- Deixamos o vetor quase ordenado usando o QuickSort
  - pedaços de tamanho no máximo `M+1` sem ordenar
- Ordenamos o vetor usando InsertionSort
  - Ele é rápido para vetor quase ordenado

```
1  #define M 10
2
3  void quicksort_mdt_v3_rec(int *v, int l, int r) {
4    int i;
5    if(r - l <= M) return;
6    troca(v[(l+r)/2], v[l+1]);
7    if(v[l] > v[l+1]) troca(v[l], v[l+1]);
8    if(v[l] > v[r]) troca(v[l], v[r]);
9    if(v[l+1] > v[r]) troca(v[l+1], v[r]);
10   i = partition(v, l+1, r-1);
11   quicksort_mdt_v3_rec(v, l, i-1);
12   quicksort_mdt_v3_rec(v, i+1, r);
13 }
14
15 void quicksort_mdt_v3(int *v, int l, int r) {
16   quicksort_mdt_v3_rec(v, l, r);
17   insertionsort_v4(v, l, r);
18 }
```

Experimentos para vetores aleatórios



Experimentos para vetores quase ordenados



Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é $\mathrm{O}(n \lg n)$ em média

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é $\mathrm{O}(n \lg n)$ em média
  - Não importa qual é o vetor de entrada

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é $\mathrm{O}(n \lg n)$ em média
  - Não importa qual é o vetor de entrada
- Usar a mediana de três elementos como pivô pode melhorar o resultado

Conclusão

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser mais rápido que o MergeSort na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Sua versão aleatorizada é $\mathrm{O}(n \lg n)$ em média
  - Não importa qual é o vetor de entrada
- Usar a mediana de três elementos como pivô pode melhorar o resultado
- Ele pode ser melhorado usando o InsertionSort na base da recursão