Ordenação

Prof. Paulo Henrique Pisani

abril/2022

Ordenação

- Ordenação é o processo de **rearranjar uma sequência de elementos em ordem ascendente ou descendente**, de acordo com a **chave** de cada elemento;
- Um dos principais objetivos de realizar a ordenação é **facilitar a recuperação** dos elementos por sua chave.

Ordenação eficiente

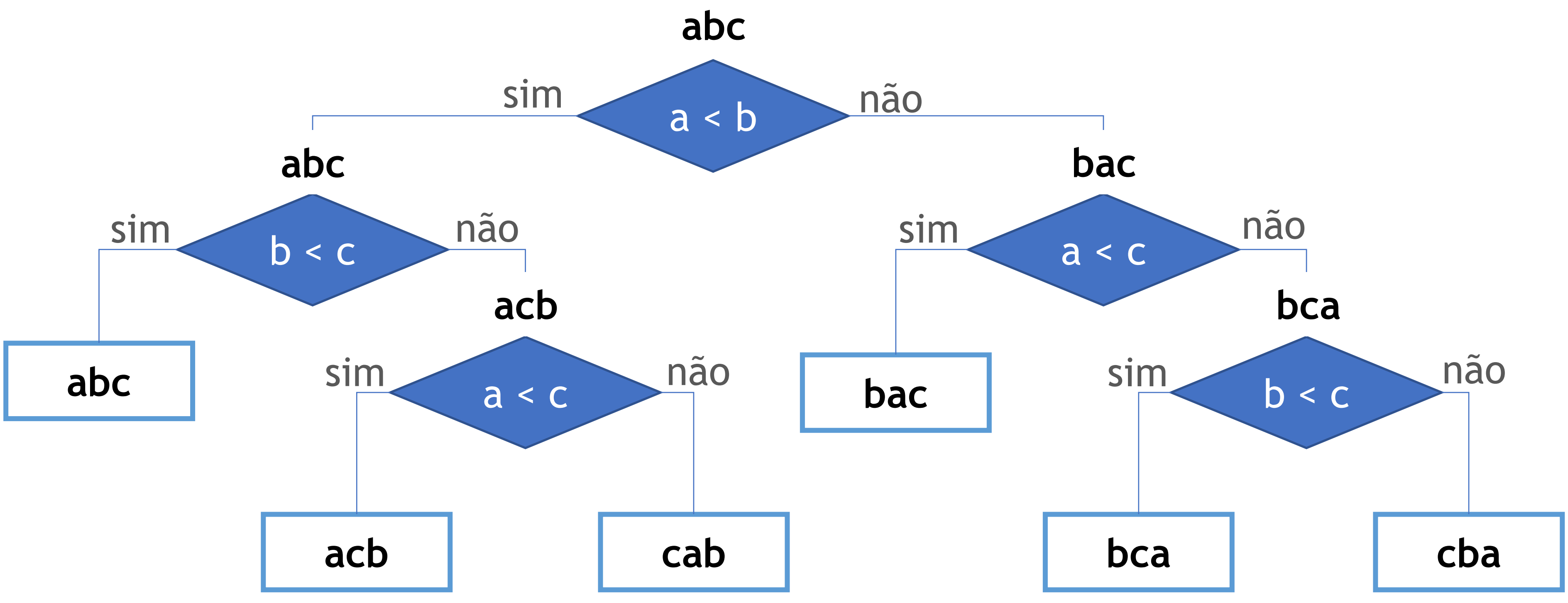
- Um algoritmo de ordenação baseado em comparações pode chegar a complexidade de tempo $\Omega(n.\log(n))$ no pior caso;
  - Há uma demonstração em:

    SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3a edição. Rio de Janeiro, RJ: LTC, 2012. (páginas 175 a 177)

  - A seguir, é discutido como chegar nesse limite assintótico.

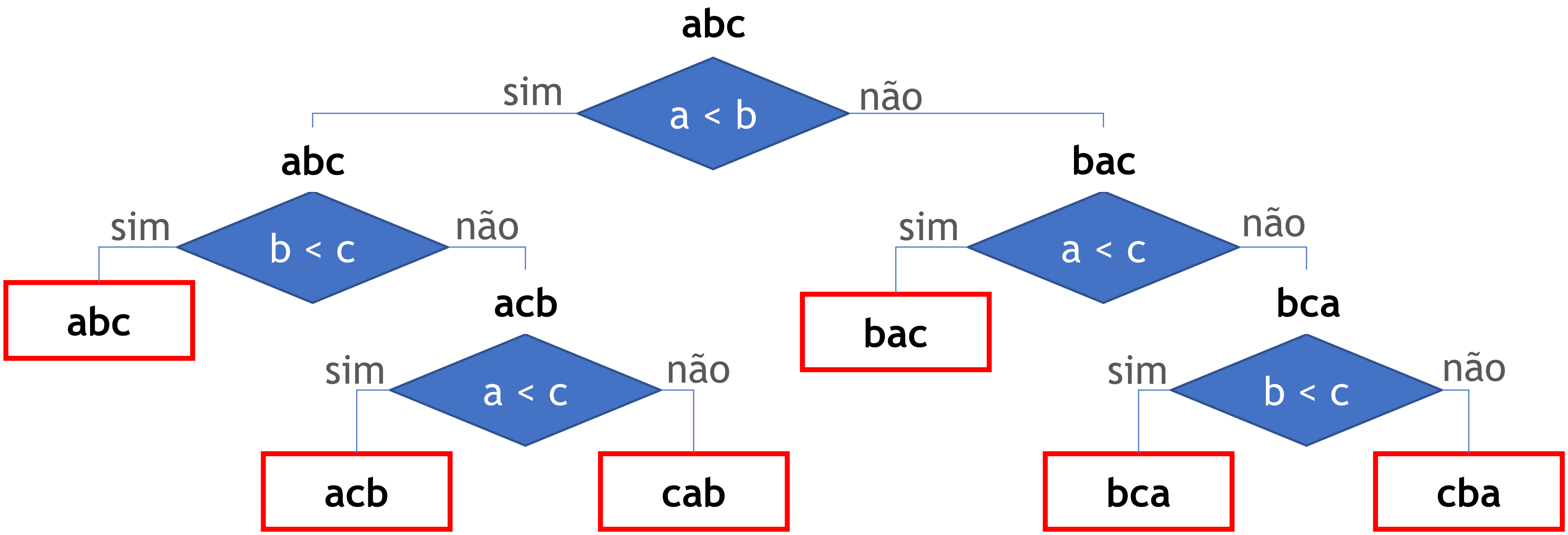
Limite assintótico da ordenação baseada em comparações

- Podemos representar a ordenação por comparação usando uma árvore binária, conforme figura a seguir.



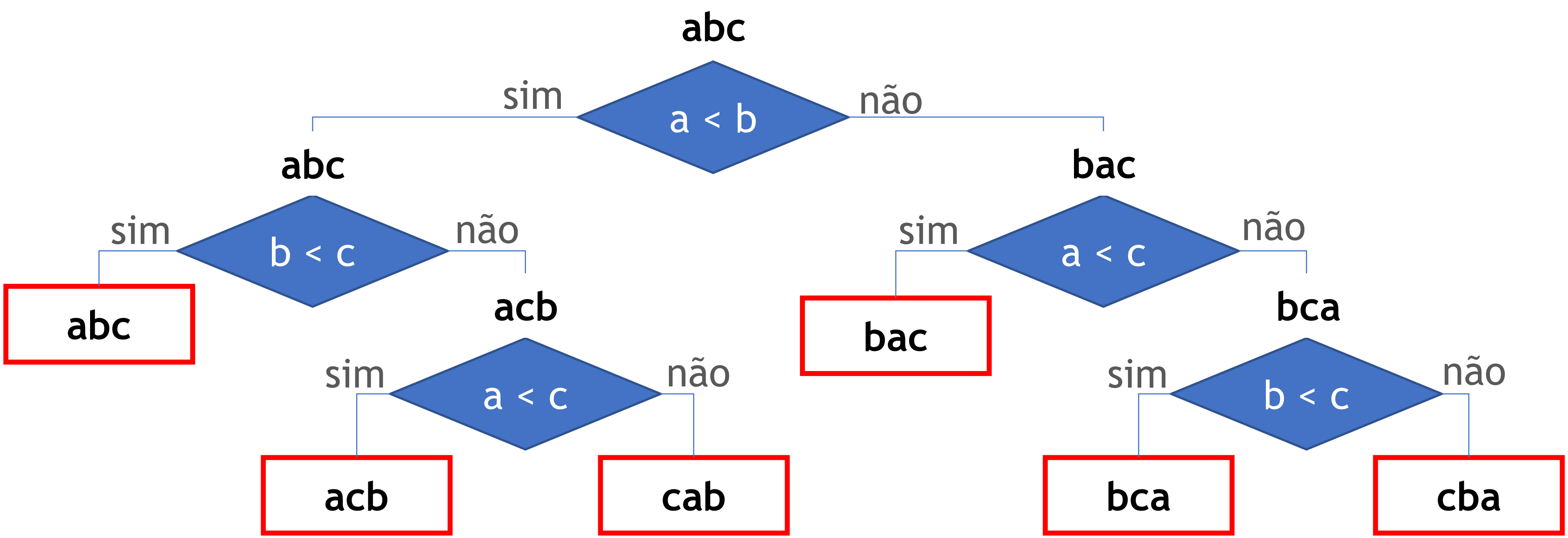
Limite assintótico da ordenação baseada em comparações

- Para ordenar n elementos, existem n! saídas possíveis; Portanto, uma árvore para um algoritmo de ordenação por comparação correto possui **n! folhas** (que seriam todas as saídas possíveis);



Limite assintótico da ordenação baseada em comparações

- Em uma árvore com altura h, o número máximo de nós folha é 2^h
- Portanto, $n! = 2^h$; Ou seja, $h = \lg(n!)$



Limite assintótico da ordenação baseada em comparações

- O número de comparações no pior caso depende da altura da árvore, que é $h = \lg(n!)$
- Uma aproximação para n! é $\left(\frac{n}{e}\right)^n$
- Com isso, temos que:

$$h = \lg(n!) = \lg\left(\left(\frac{n}{e}\right)^n\right)$$

$$h = n.\lg(n) - n.\lg(e)$$

$$h = \Omega(n.\lg(n))$$

Esse é o limite assintótico que um algoritmo baseado em comparações pode chegar no pior caso.

Merge sort

Merge sort

- Divide a lista em duas subsequências (com tamanho n/2);
- Aplica ordenação (por merge sort) nas duas subsequências;
- Intercala as duas subsequências ordenadas.

Merge sort

- Divide a lista em duas subsequências (com tamanho n/2);
- Aplica ordenação (por merge sort) nas duas subsequências;
- Intercala as duas subsequências ordenadas.

Um ponto importante do merge sort é o algoritmo de intercalação.

Intercalação

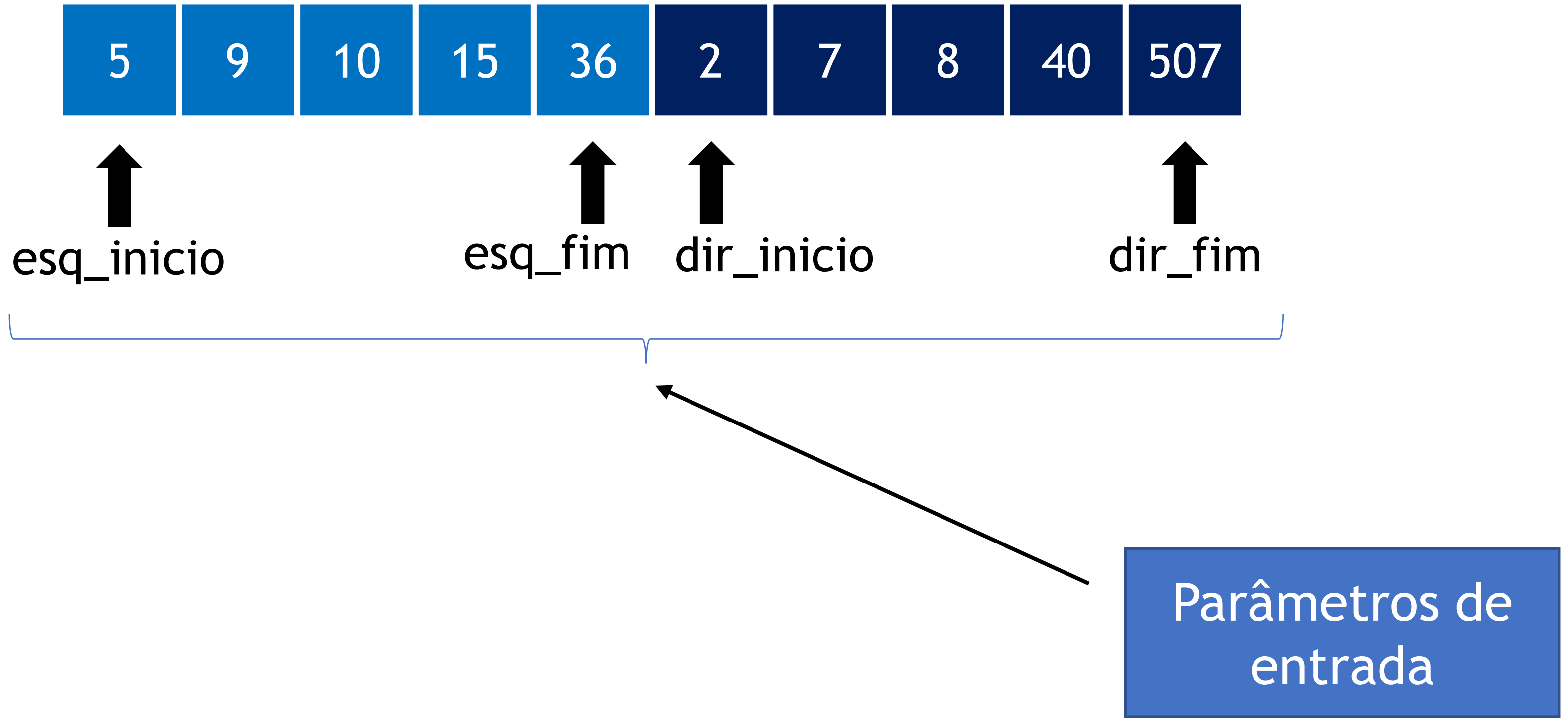
Podemos implementar a intercalação em O(n)

- Definição do problema: dado um vetor, intercalar duas subsequências ordenadas de forma que o vetor fique ordenado.

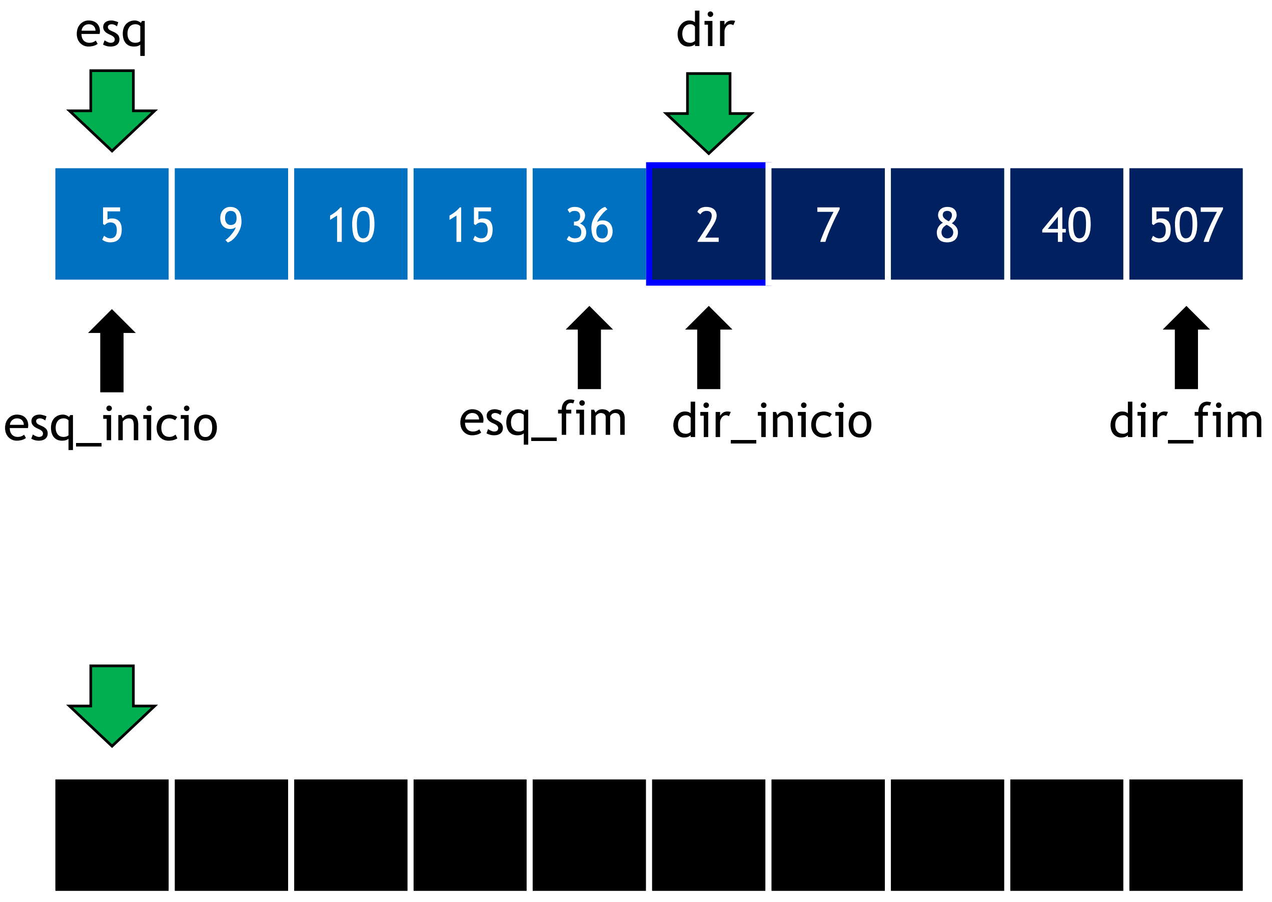
5	9	10	15	36	2	7	8	40	507

2	5	7	8	9	10	15	36	40	507

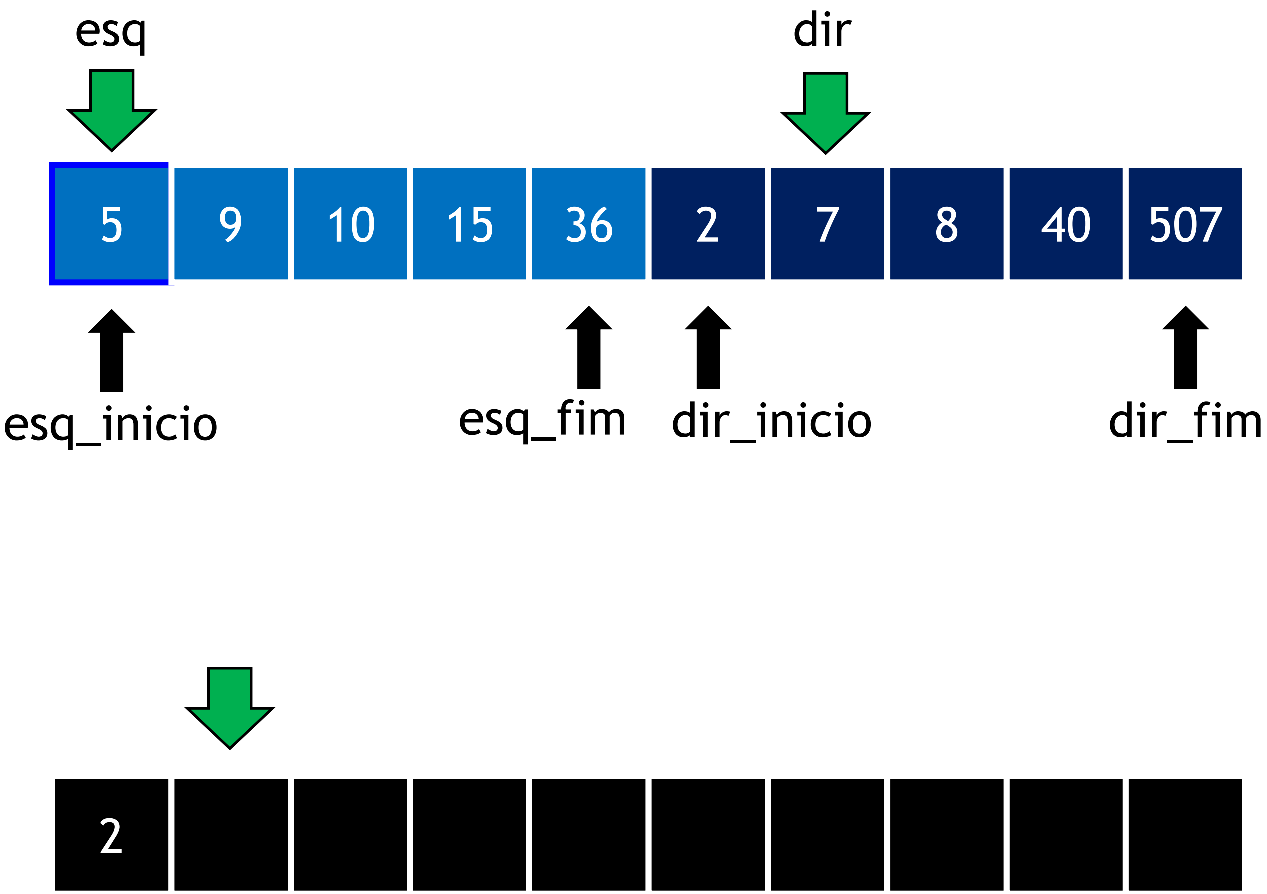
Intercalação



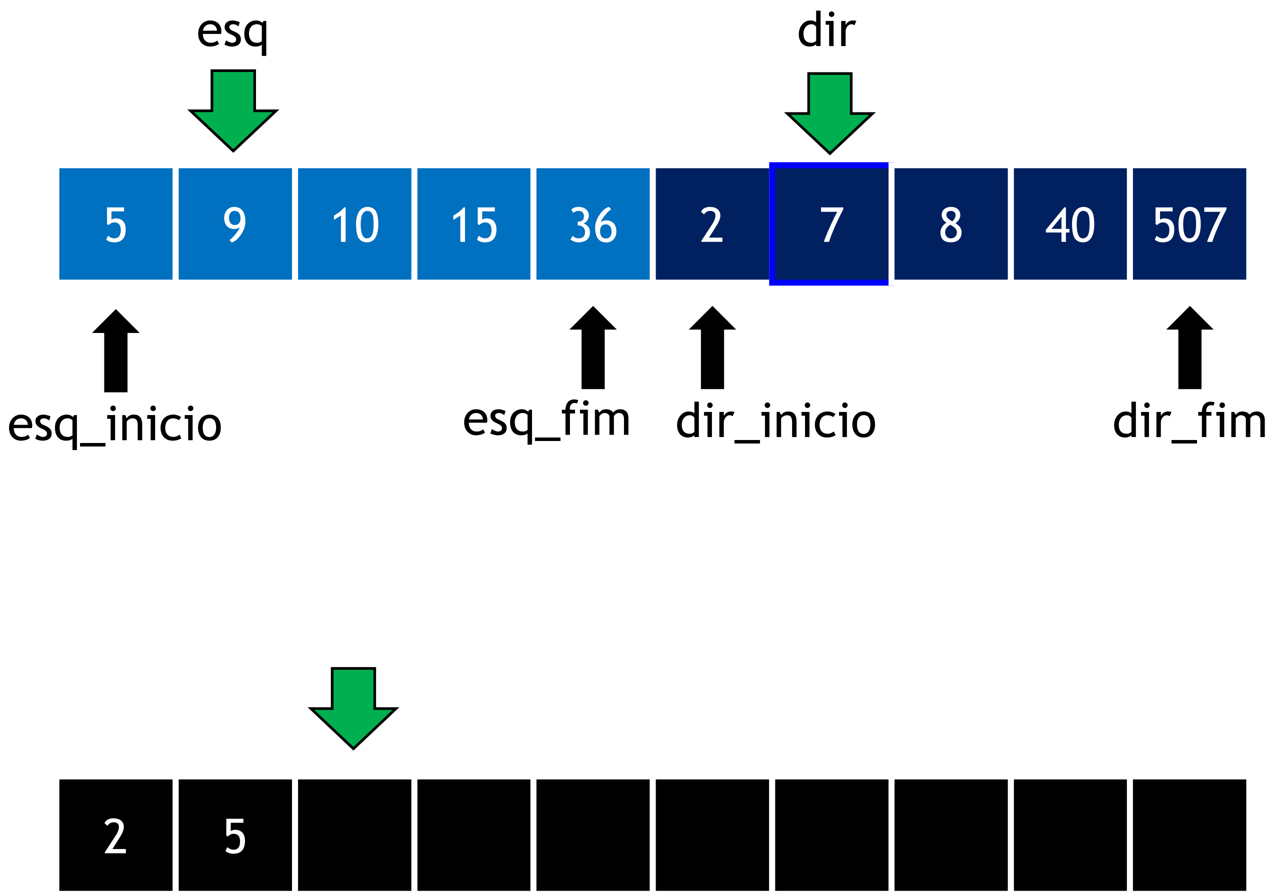
Intercalação



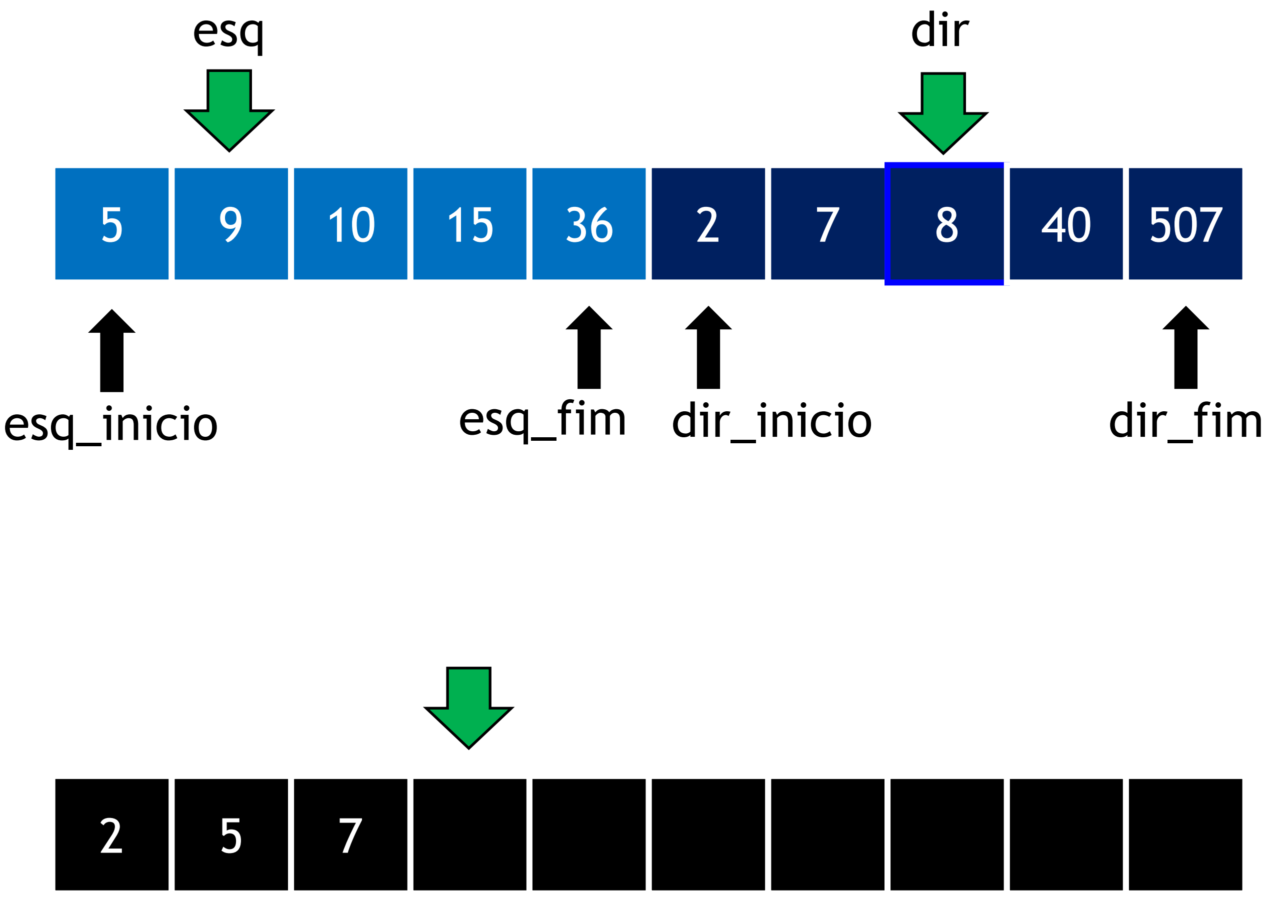
Intercalação



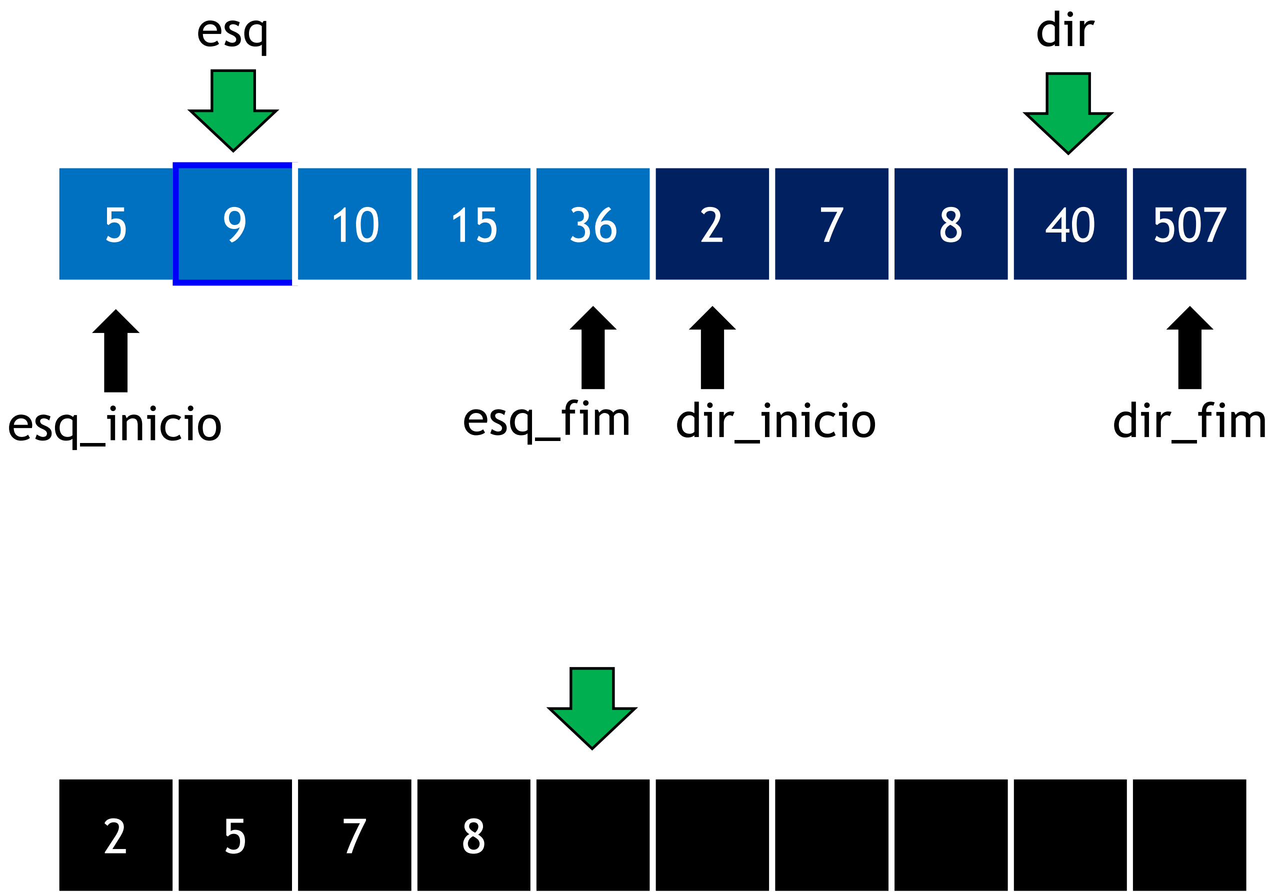
Intercalação



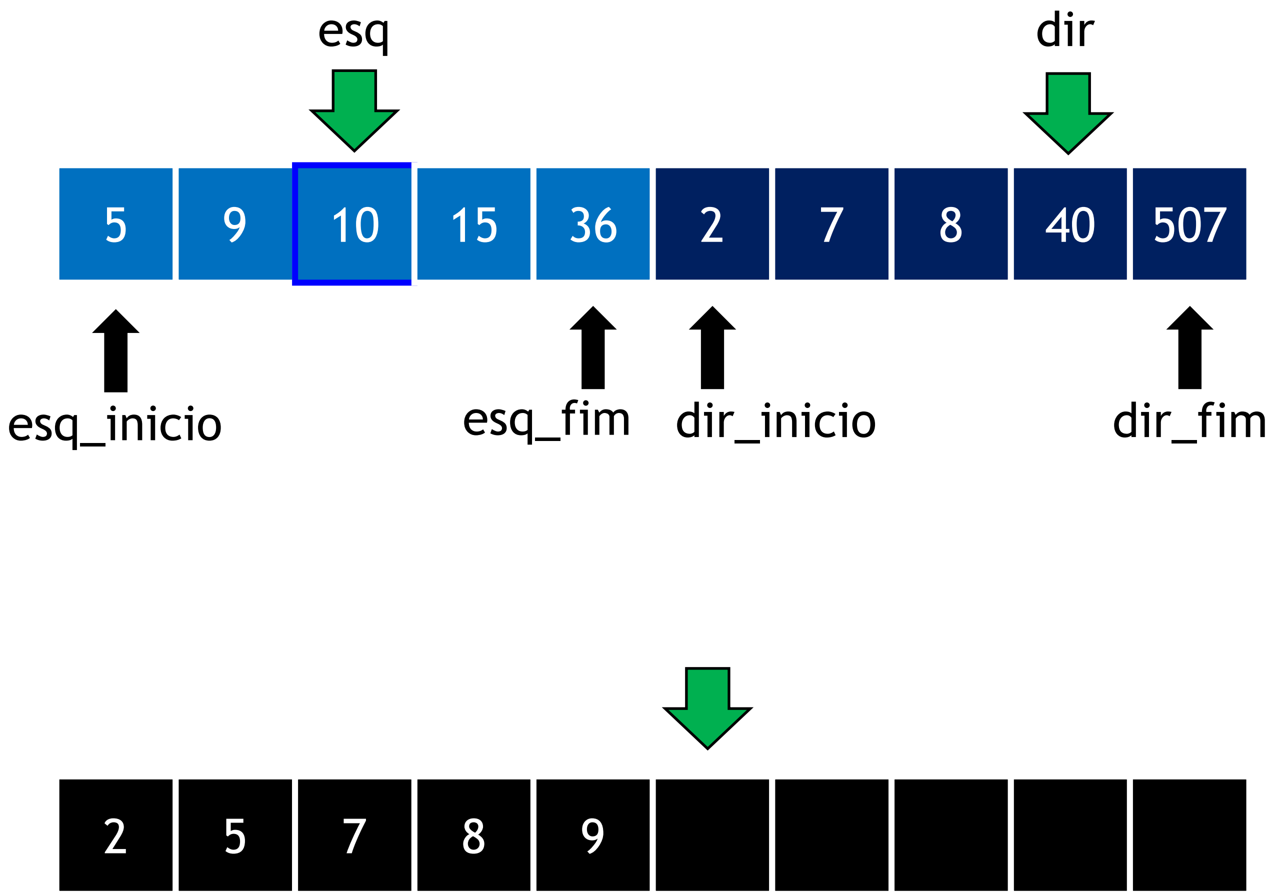
Intercalação



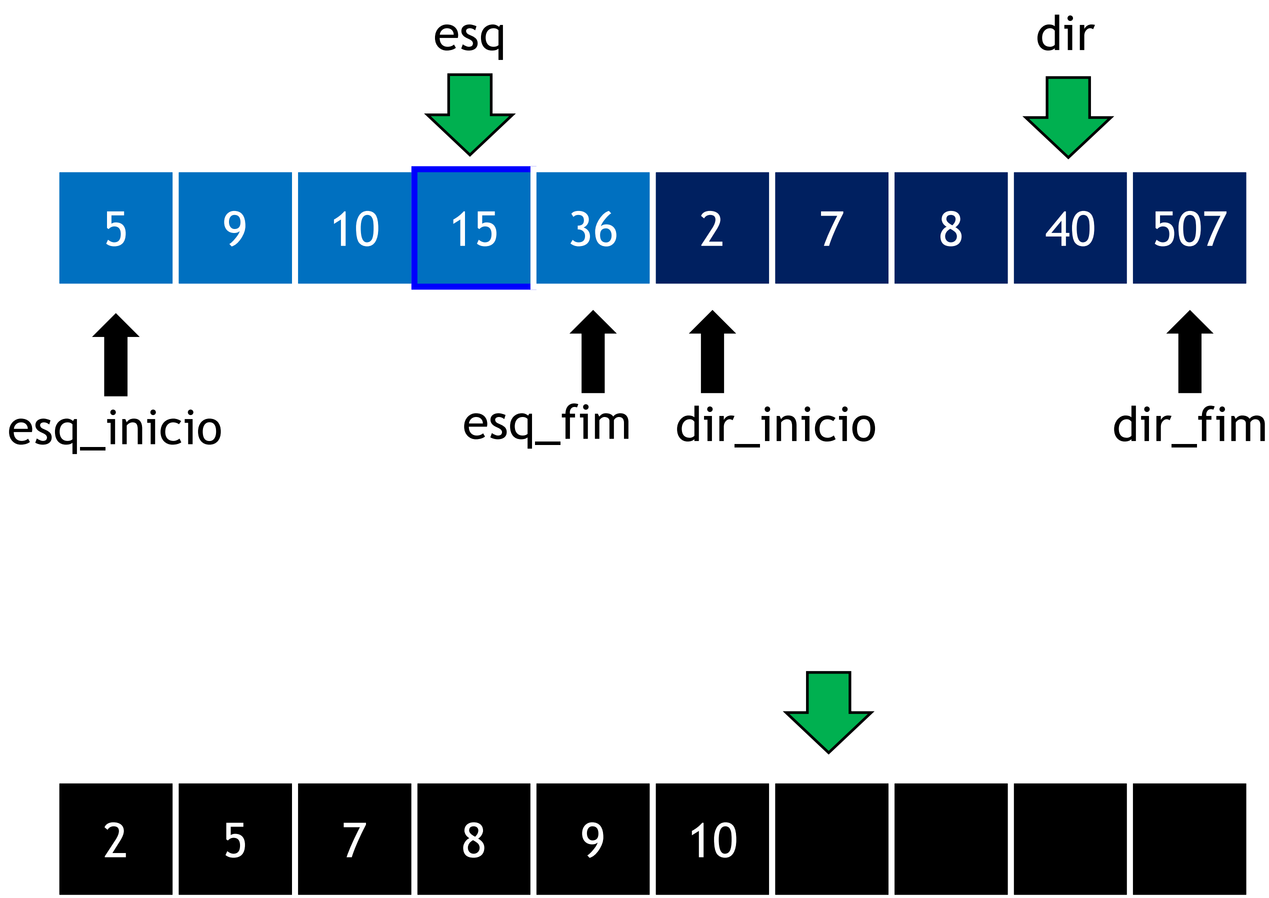
Intercalação



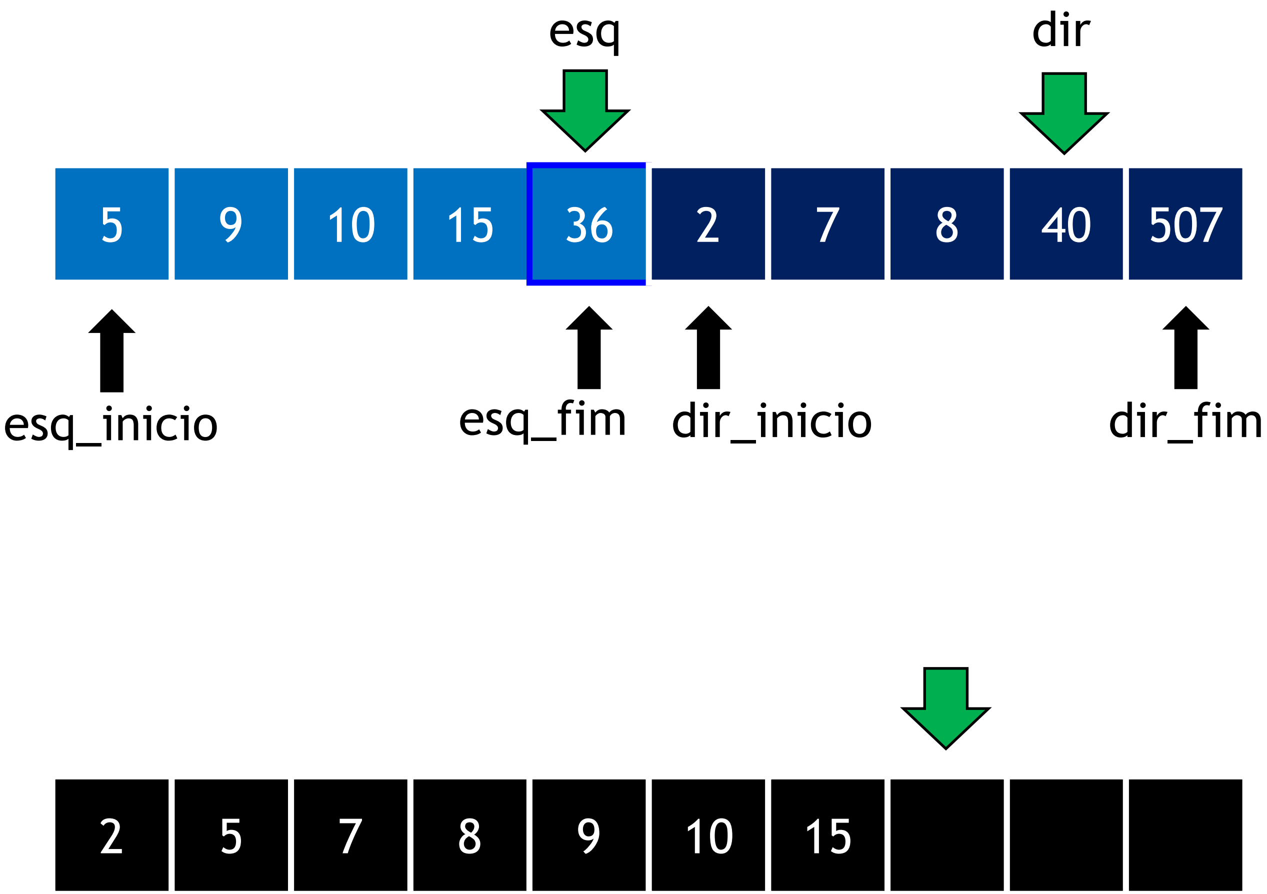
Intercalação



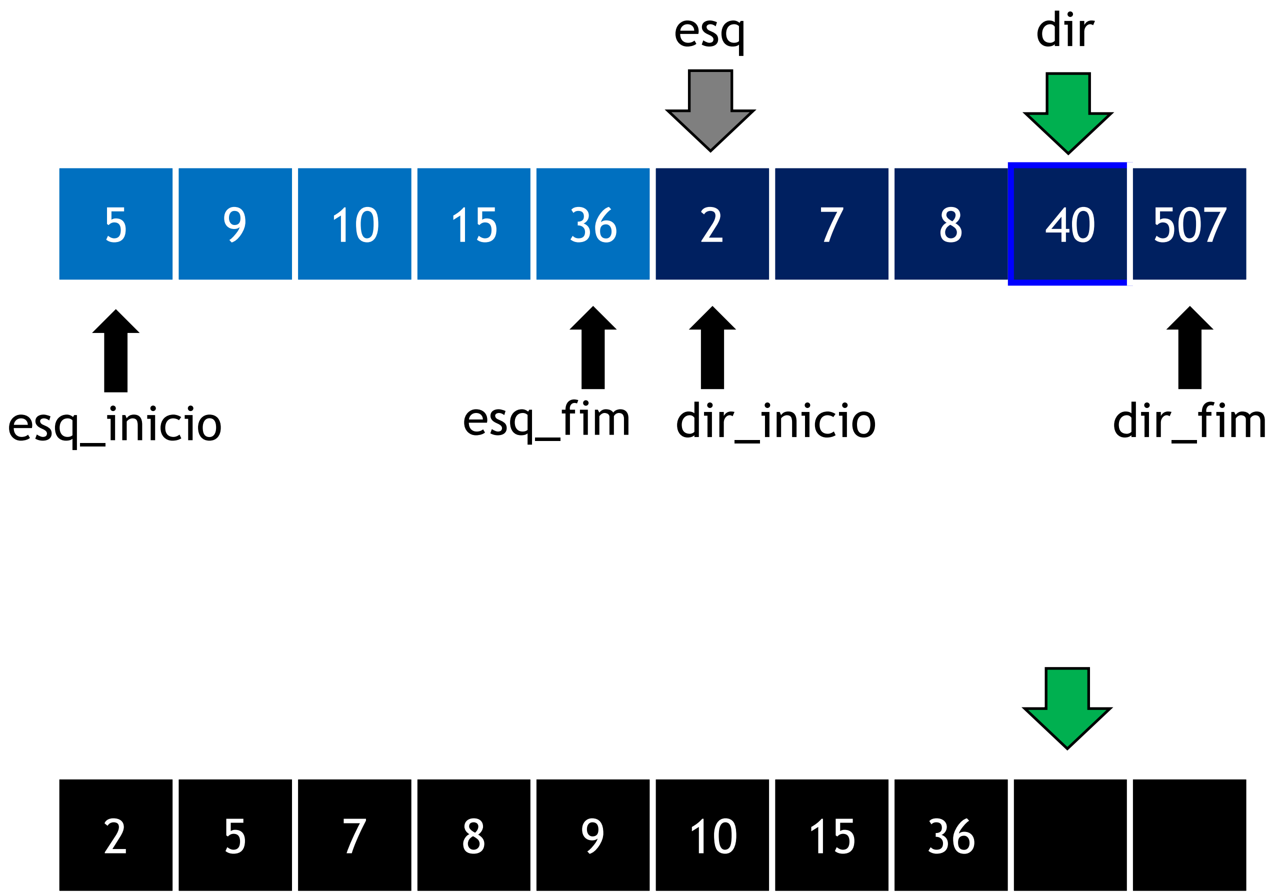
Intercalação



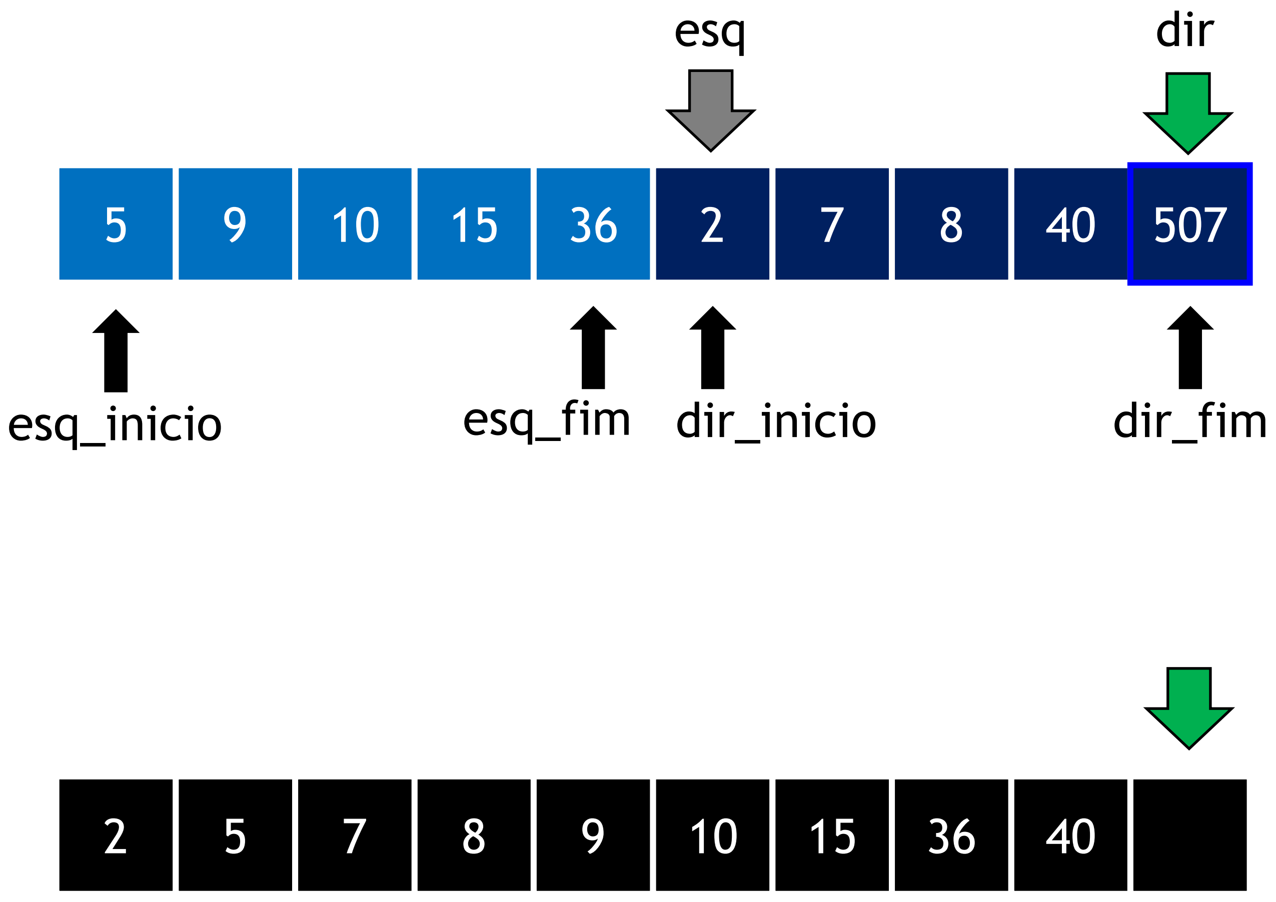
Intercalação



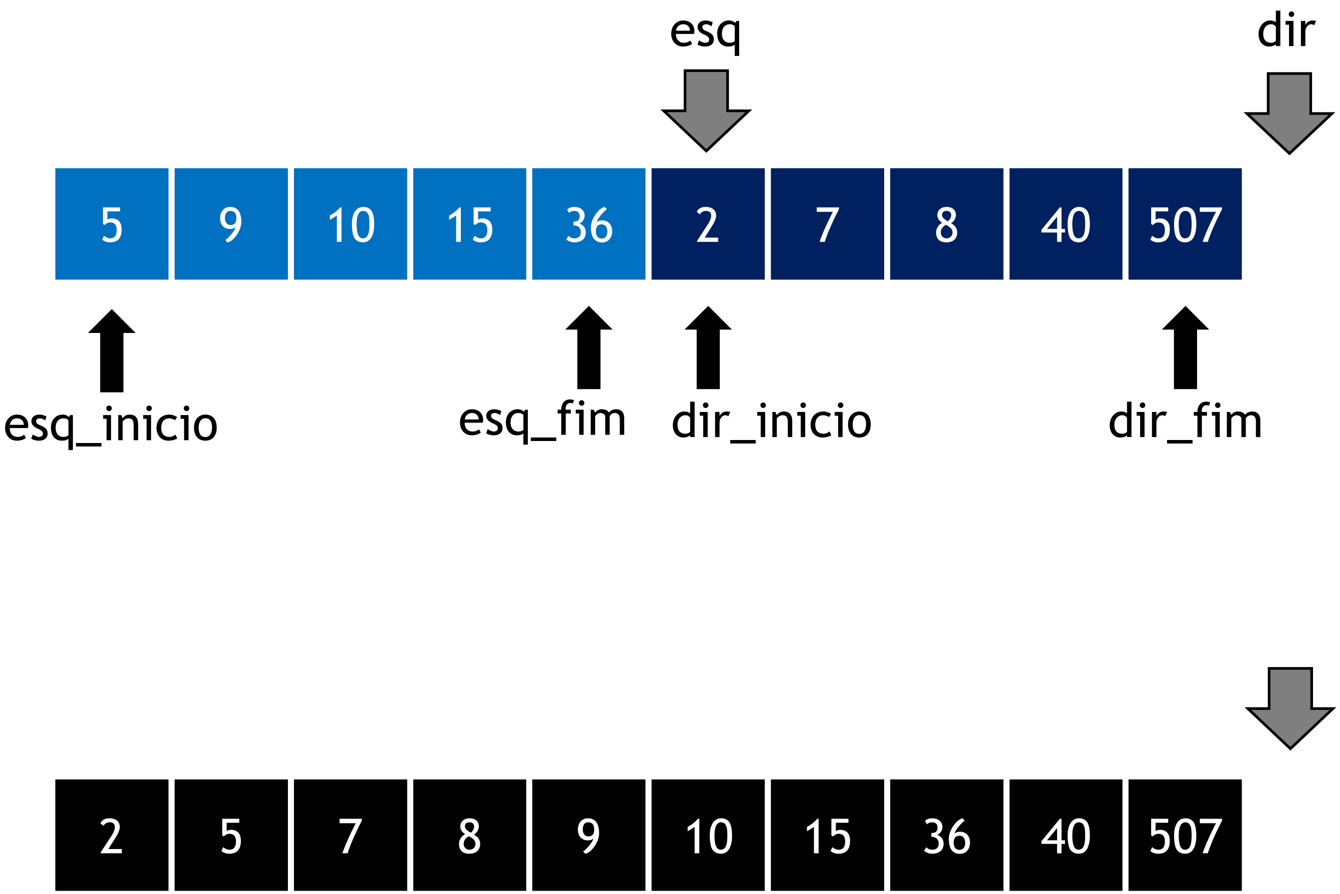
Intercalação



Intercalação



Intercalação



Intercalação

- Implementação C

Chamada:
```
intercala(vetor, esq_inicio, esq_fim, dir_fim);
```

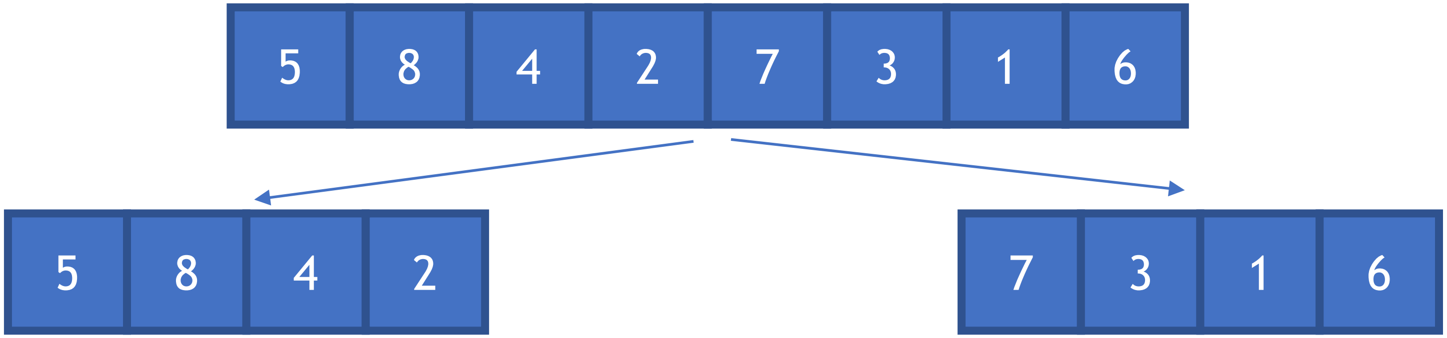
Implementação

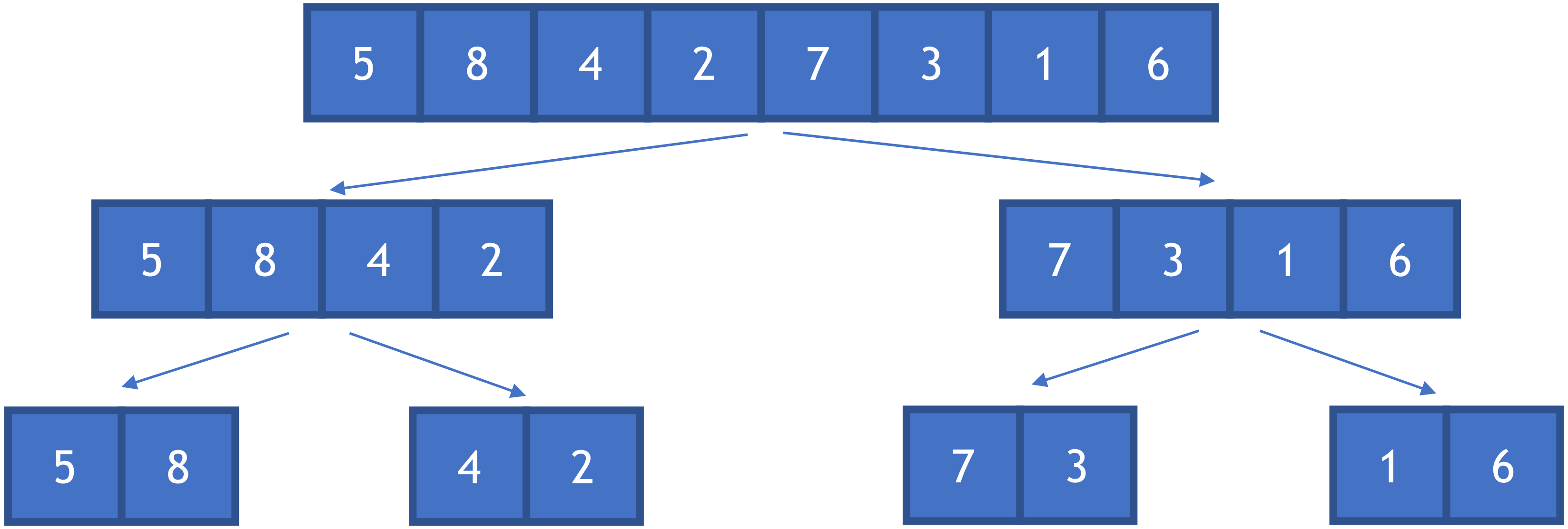
```
void intercala(int *v, int esq_inicio, int esq_fim, int dir_fim) {
     int dir_inicio = esq_fim + 1;
     int esq = esq_inicio, dir = dir_inicio;

     int comp = dir_fim - esq_inicio + 1;
     int *v_aux = malloc(sizeof(int) * comp);

     int i;
     for (i = 0; i < comp; i++) {
          if (esq > esq_fim)
               v_aux[i] = v[dir++];
          else if (dir > dir_fim)
               v_aux[i] = v[esq++];
          else if (v[esq] <= v[dir])
               v_aux[i] = v[esq++];
          else
               v_aux[i] = v[dir++];
     }

     for (i = 0; i < comp; i++)
          v[esq_inicio + i] = v_aux[i];

     free(v_aux);
}
```

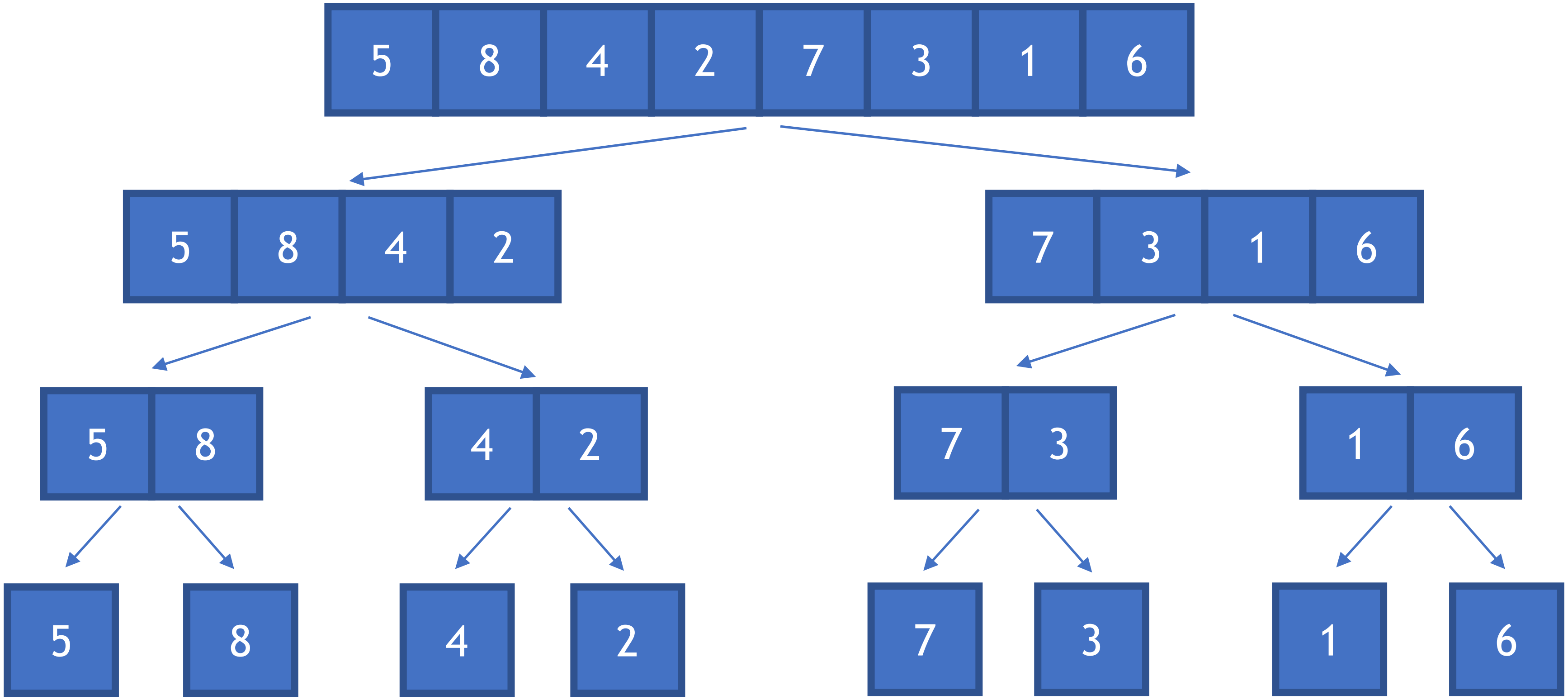
Merge sort

- Divide a lista em duas subsequências (com tamanho n/2);
- Aplica ordenação (por merge sort) nas duas subsequências;
- Intercala as duas subsequências ordenadas.

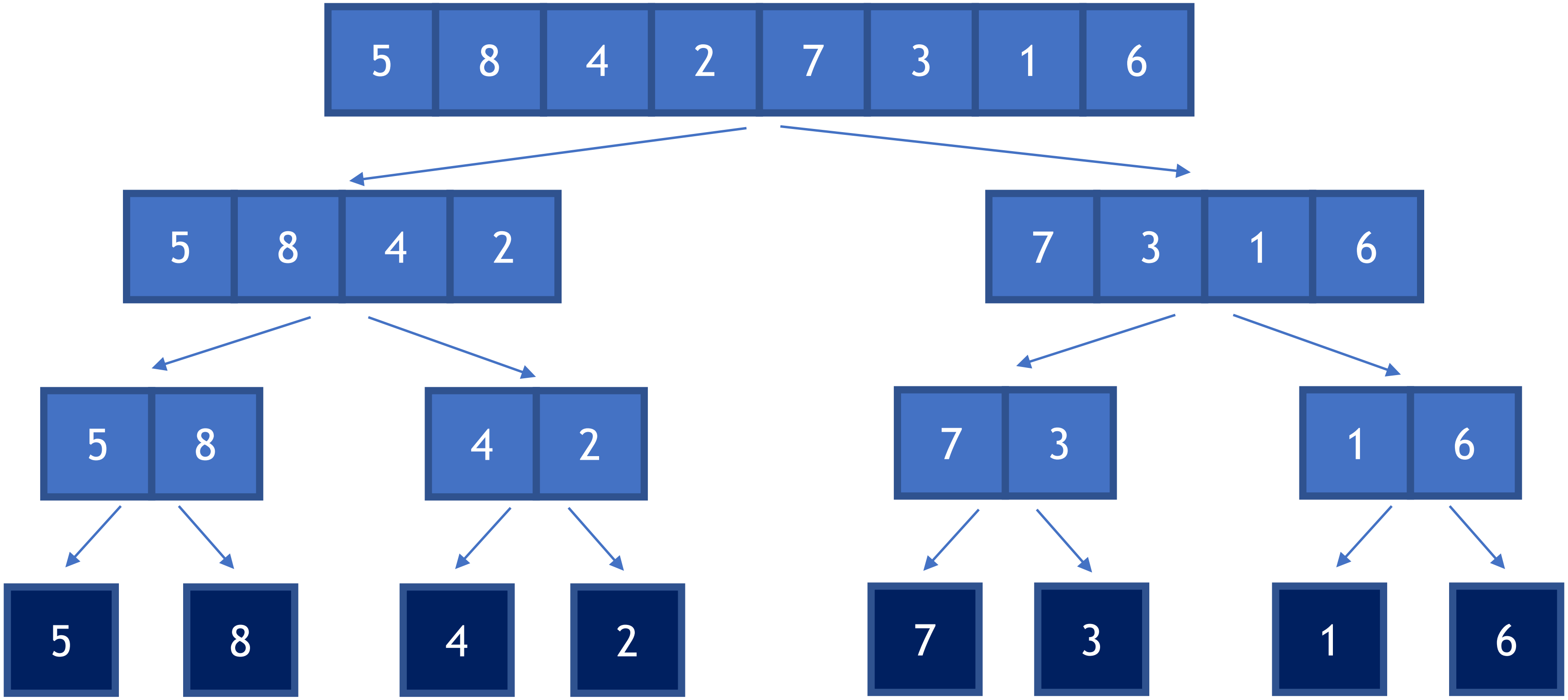
5	8	4	2	7	3	1	6

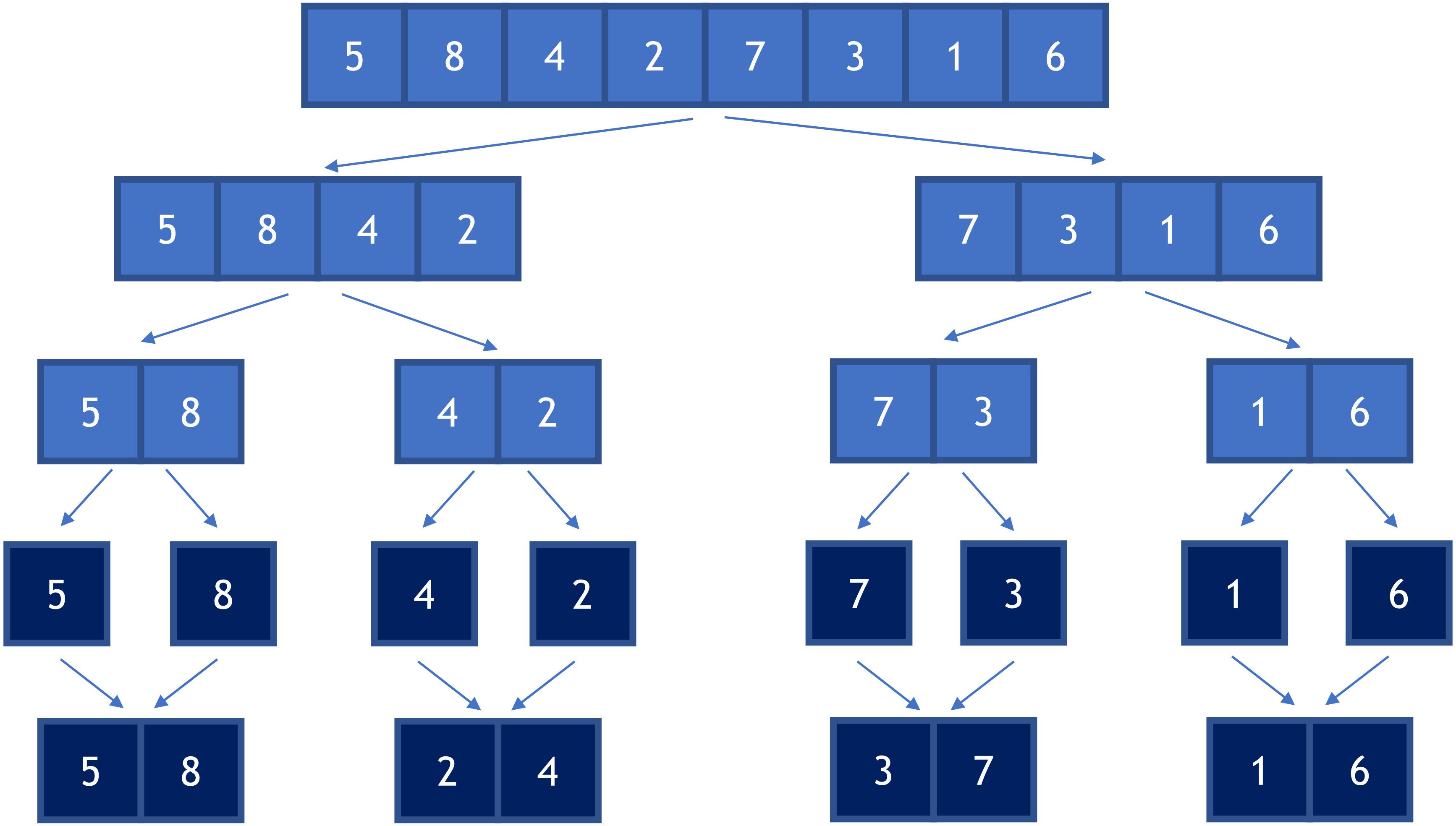

5 8 4 2 7 3 1 6
5 8 4 2
7 3 1 6

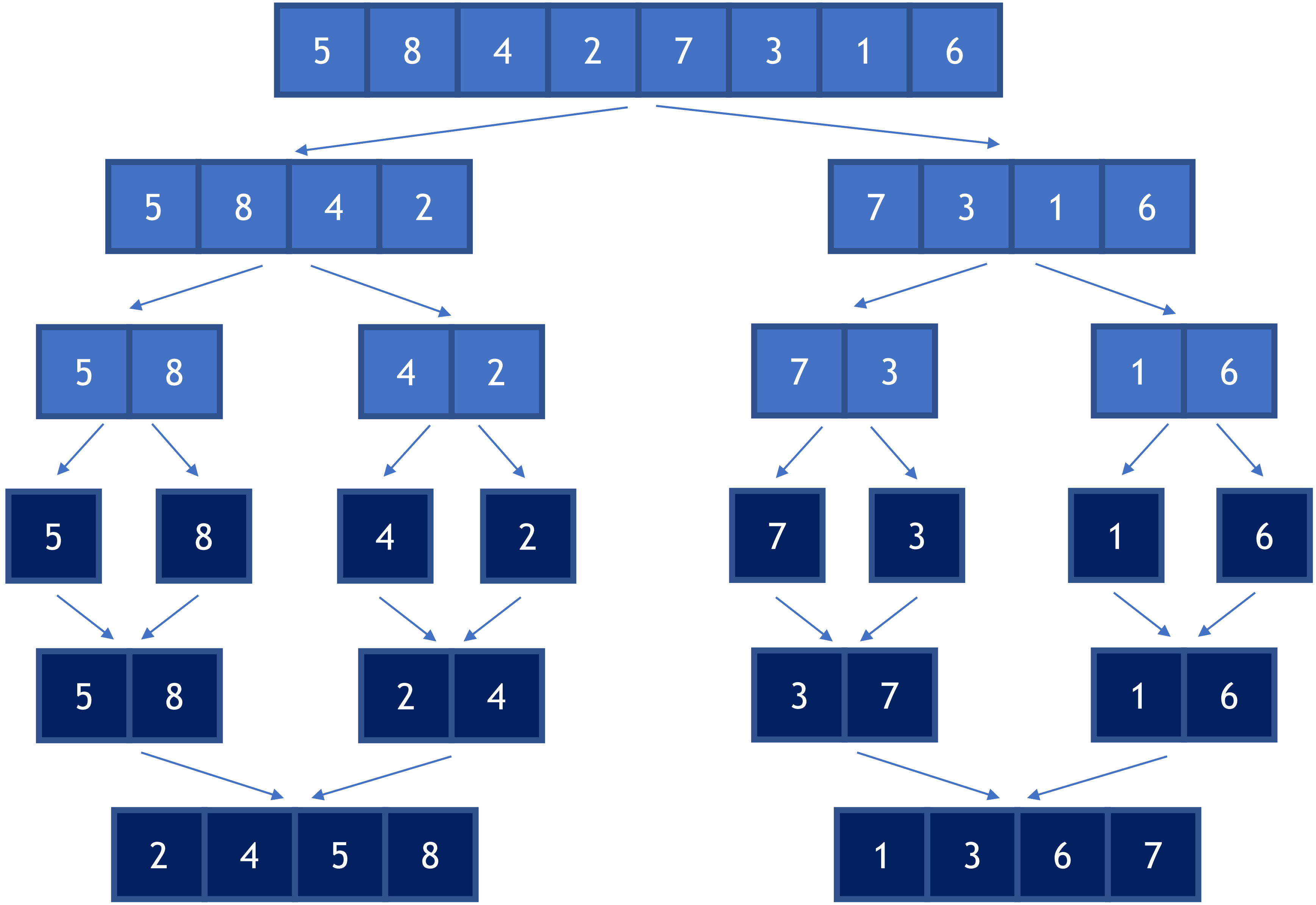

5 8 4 2 7 3 1 6
5 8 4 2
7 3 1 6
5 8
4 2
7 3
1 6

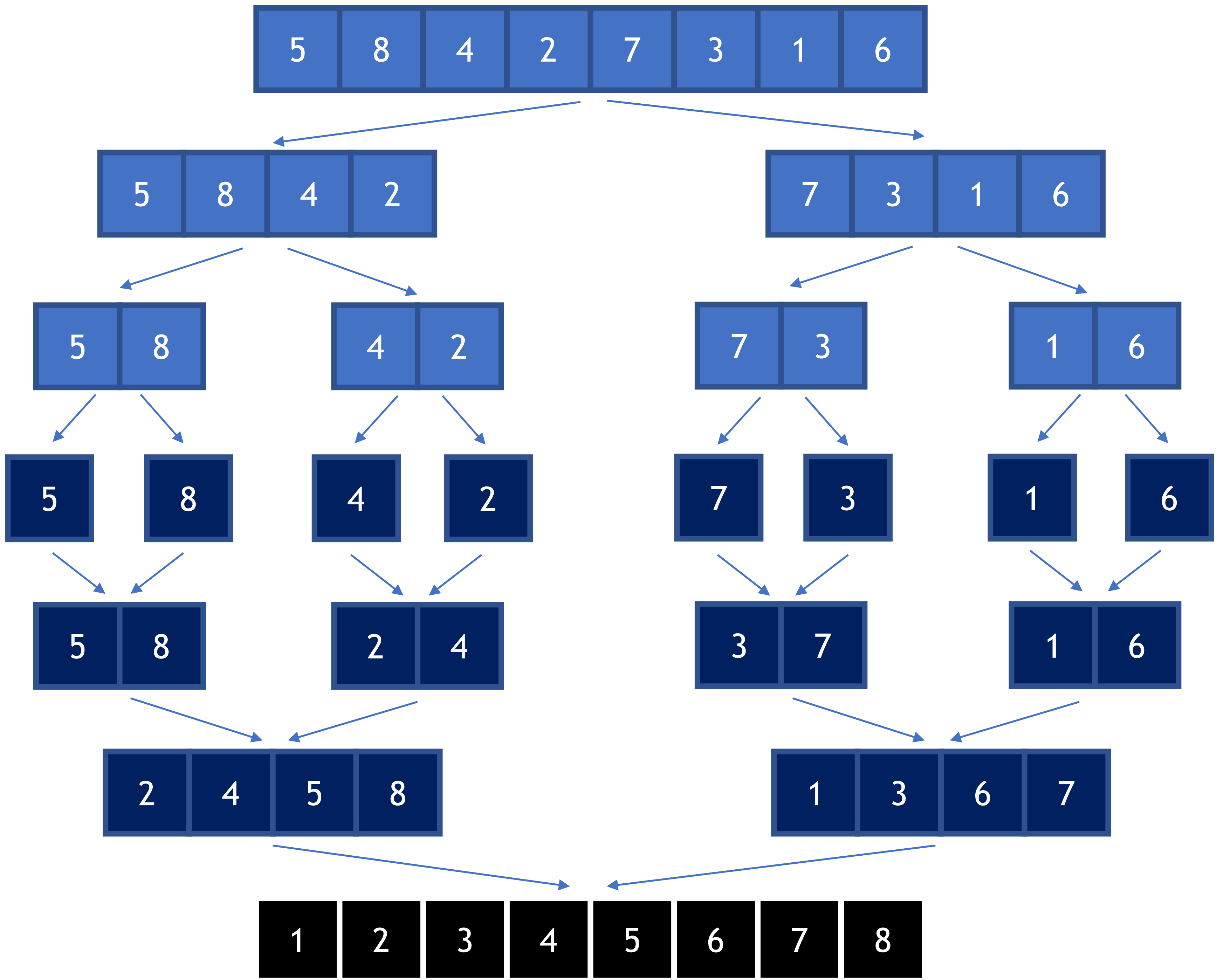


5 8 4 2 7 3 1 6
5 8 4 2
7 3 1 6
5 8
4 2
7 3
1 6
5
8
4
2
7
3
1
6


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


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


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


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

Merge sort

- Implementação em C

Chamada:
```
mergesort(vetor, n);
```

Implementação

```
void merge_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int meio = (esq + dir) / 2;
        merge_sort(v, esq, meio);
        merge_sort(v, meio+1, dir);
        intercala(v, esq, meio, dir);
    }
}

void mergesort(int *v, int n) {
    merge_sort(v, 0, n - 1);
}
```

Custo do merge sort

```
void merge_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int meio = (esq + dir) / 2;
        merge_sort(v, esq, meio);
        merge_sort(v, meio+1, dir);
        intercala(v, esq, meio, dir);
    }
}
```

$T(n)$

Custo do merge sort

```
void merge_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int meio = (esq + dir) / 2;
        merge_sort(v, esq, meio);
        merge_sort(v, meio+1, dir);
        intercala(v, esq, meio, dir);
    }
}
```

$T(n)$

$T(n/2)$

$T(n/2)$

$O(n)$

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2.T(n/2) + O(n), & n > 1 \end{cases}$$

Custo do merge sort

```
void merge_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int meio = (esq + dir) / 2;
        merge_sort(v, esq, meio);
        merge_sort(v, meio+1, dir);
        intercala(v, esq, meio, dir);
    }
}
```

$T(n)$

$T(n/2)$

$T(n/2)$

$O(n)$

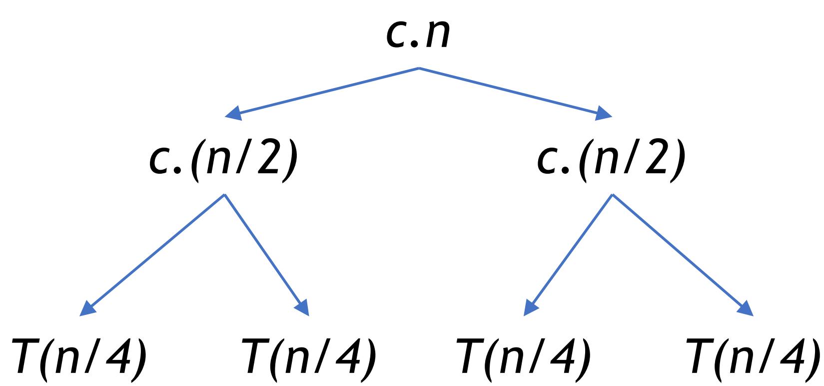
$$T(n) = \begin{cases} c, & n = 1 \\ 2.T(n/2) + c.n, & n > 1 \end{cases}$$

Custo do merge sort

$$\begin{tikzcd} & c.n \arrow[dl] \arrow[dr] & \\ T(n/2) & & T(n/2) \end{tikzcd}$$

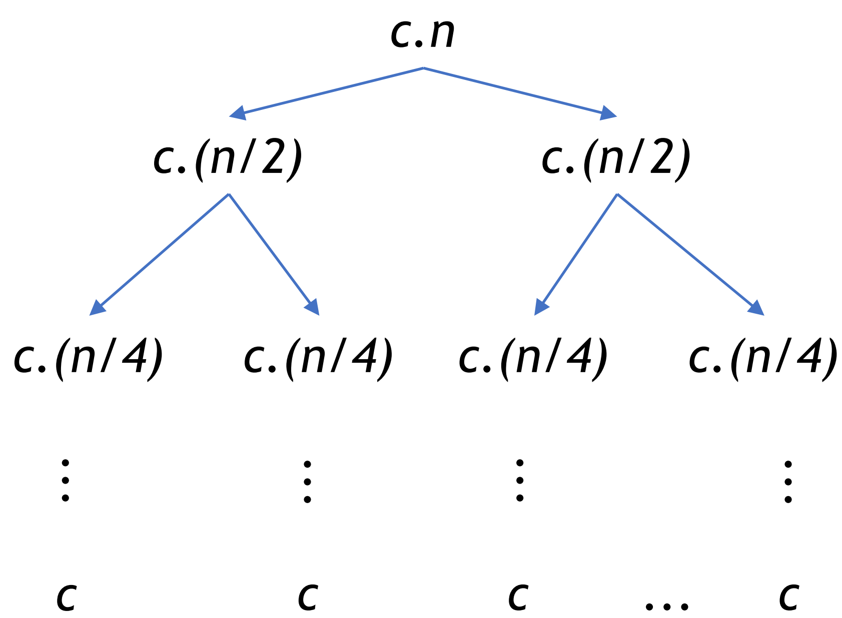
$$T(n) = \begin{cases} c, & n = 1 \\ 2.T(n/2) + c.n, & n > 1 \end{cases}$$

Custo do merge sort



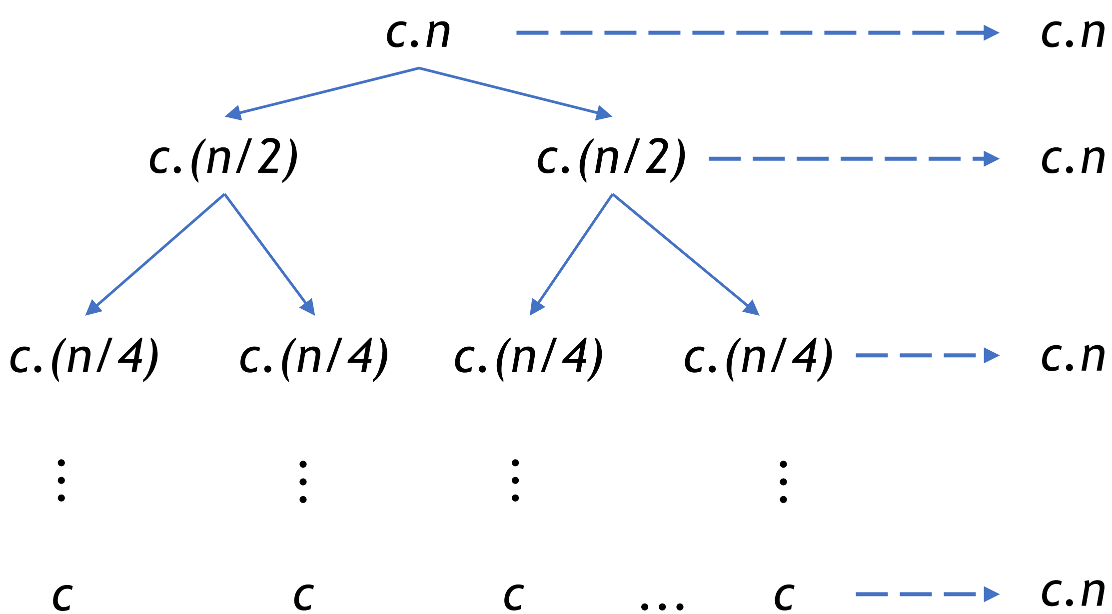
$$T(n) = \begin{cases} c, & n = 1 \\ 2.T(n/2) + c.n, & n > 1 \end{cases}$$

Custo do merge sort

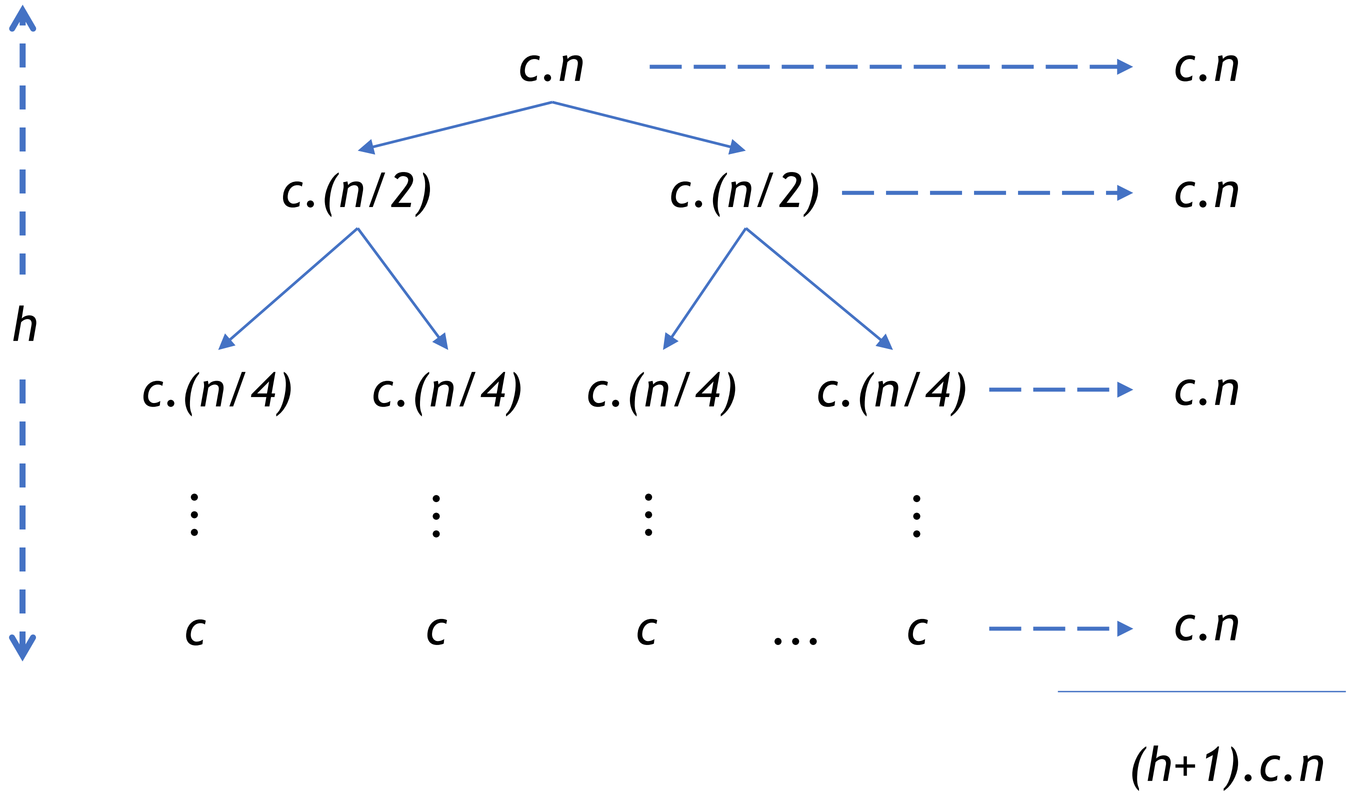


$$T(n) = \begin{cases} c, & n = 1 \\ 2.T(n/2) + c.n, & n > 1 \end{cases}$$

Custo do merge sort

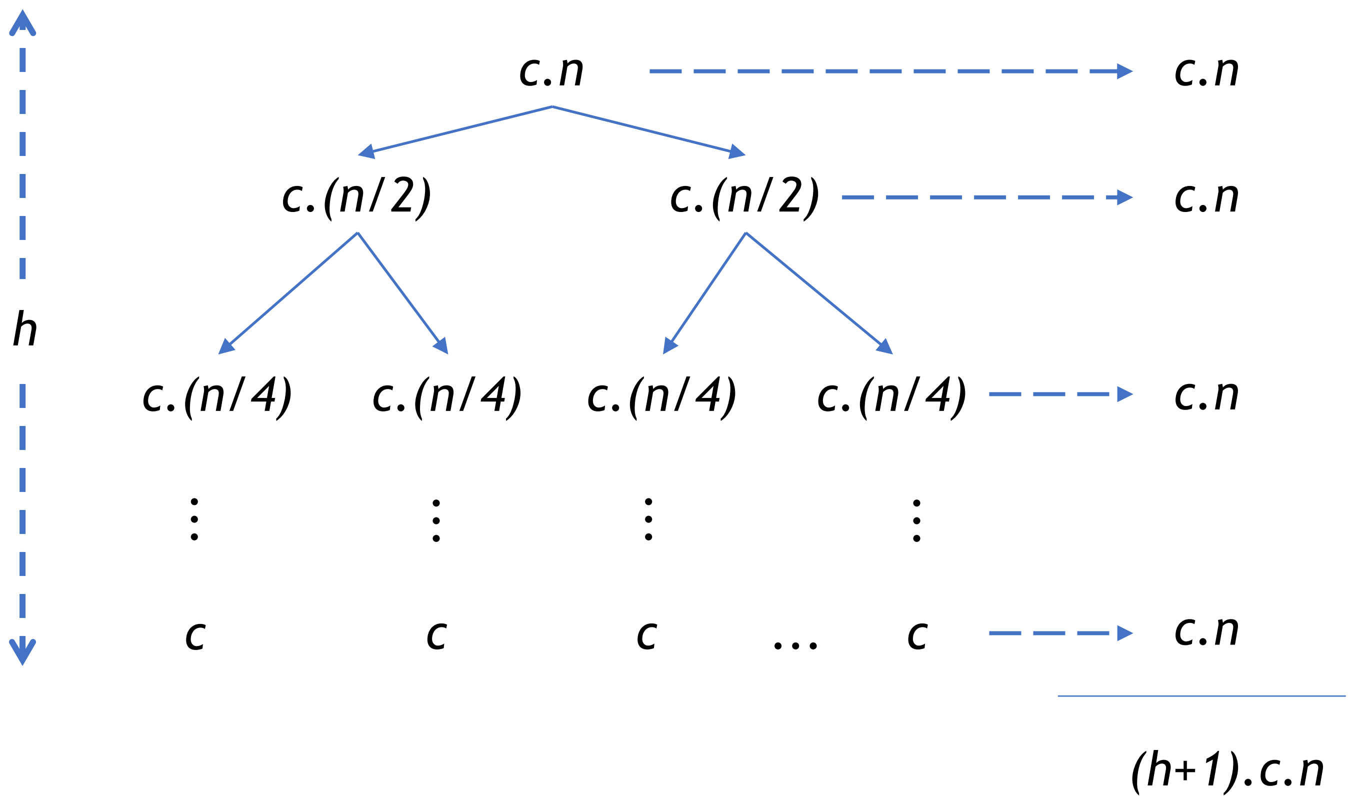


Custo do merge sort



Qual o valor da altura *h*?

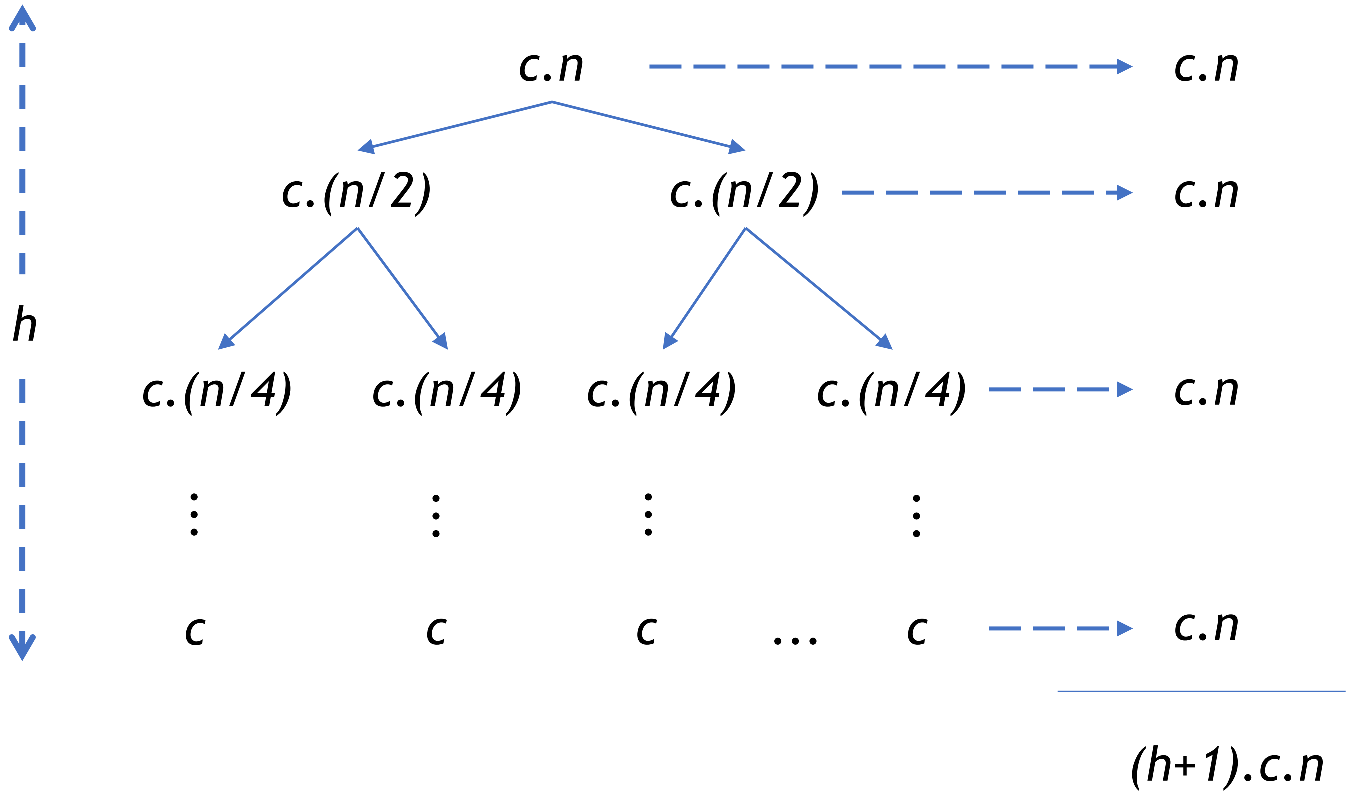
Custo do merge sort



Qual o valor da altura *h*?

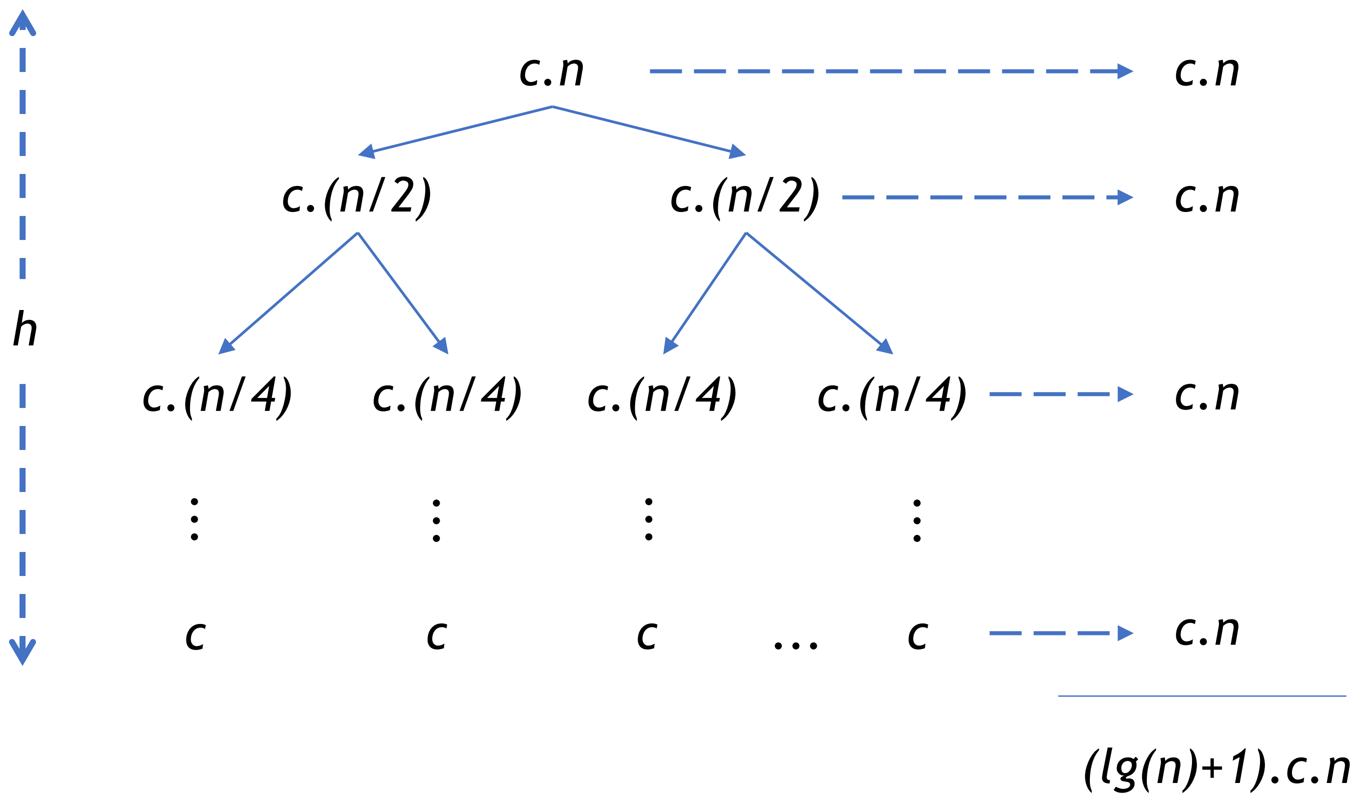
A altura da árvore representa quantas vezes podemos dividir *n* até chegar em 1.

Custo do merge sort



$$1 = \frac{n}{2^h} \Rightarrow 2^h = n \Rightarrow h = \lg(n)$$

Custo do merge sort



$$1 = \frac{n}{2^h} \Rightarrow 2^h = n \Rightarrow h = \lg(n)$$

Merge sort

- Portanto, chegamos que o custo (de tempo) do merge sort é $O(n.lg(n))$

Quick sort

Quick sort

- Algoritmo original proposto por C. A. R. Hoare em 1960 (publicado em 1962).

- Funcionamento do quick sort:
  - Escolhe pivô;
  - Particiona a lista em duas subsequências, de forma que:
    - O pivô é posicionado no índice correto (ordenação);
    - Cada elemento à esquerda tem valor menor ou igual ao pivô;
    - Cada elemento à direita tem valor maior ou igual ao pivô.
  - Ordena as subsequências (à esquerda e à direita do pivô) com quick sort.

Quick sort

- Algoritmo original proposto por C. A. R. Hoare em 1960 (publicado em 1962).

- Funcionamento do quick sort:
  - Escolhe pivô;
  - Particiona a lista em duas subsequências, de forma que:
    - O pivô é posicionado no índice correto (ordenação);
    - Cada elemento a esquerda tem valor menor que pivô;
    - Cada elemento a direita tem valor maior ou igual ao pivô.
  - Ordena as subsequências (à esquerda e à direita do pivô) com quick sort.

Forma que implementaremos o quick sort.

Quick sort

- Algoritmo original proposto por C. A. R. Hoare em 1960 (publicado em 1962).

- Funcionamento do quick sort:
  - Escolhe pivô;
  - Particiona a lista em duas subsequências, de forma que:
    - O pivô é posicionado no índice correto (ordenação);
    - Cada elemento a esquerda tem valor menor que pivô;
    - Cada elemento a direita tem valor maior ou igual ao pivô.
  - Ordena as subsequências (à esquerda e à direita do pivô) com quick sort.

Um ponto importante do quick sort é o algoritmo de particionamento.

Particionamento

Podemos implementar o particionamento em O(n)

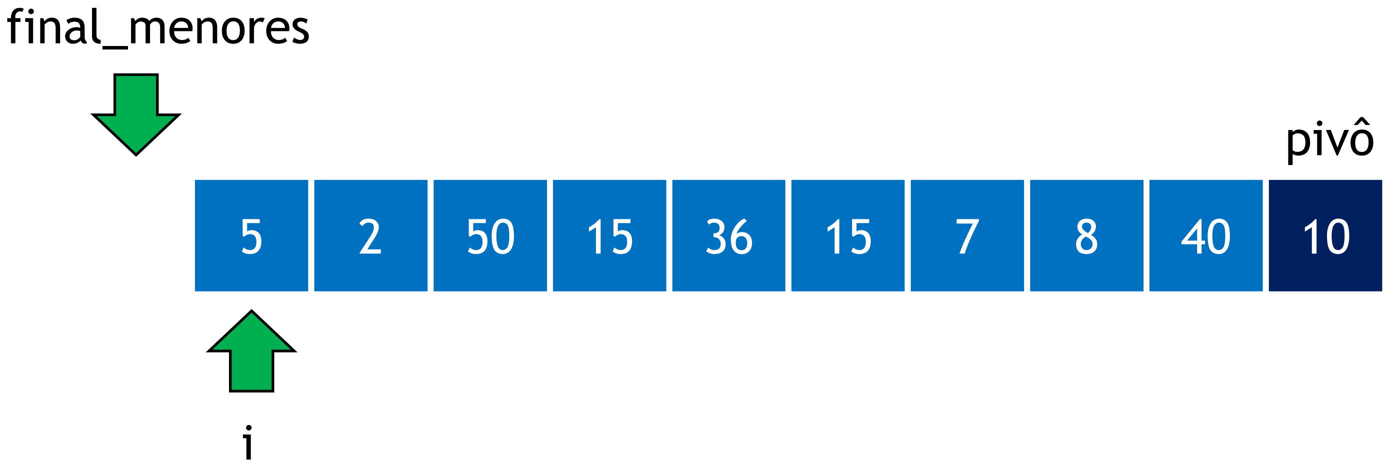
- Definição do problema: dado um vetor e um elemento deste vetor (o pivô), posicionar o pivô no índice correto (ordenação) e separar os demais elementos de modo que todos à esquerda são menores que o pivô e todos à direita são maiores ou iguais ao pivô.

5	2	50	15	36	15	7	8	40	10

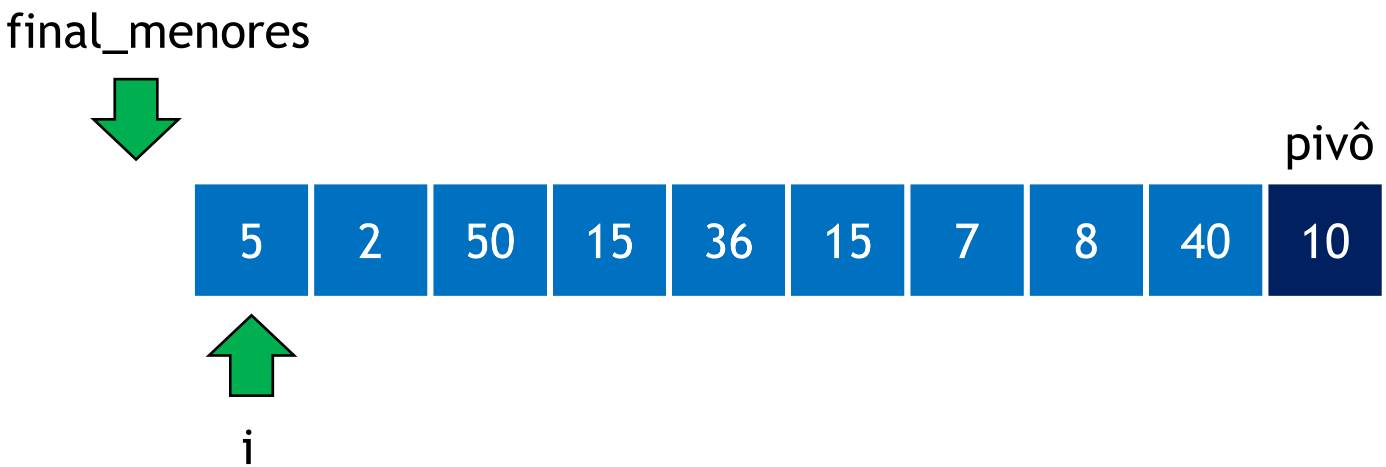
10	pivô

5	2	7	8	10	15	50	15	40	36

Particionamento

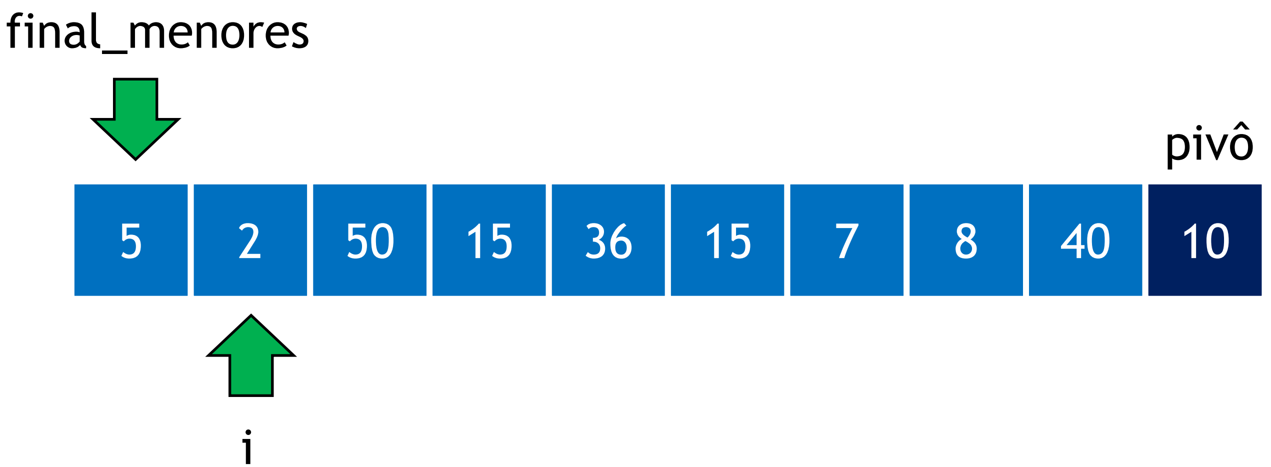


Particionamento



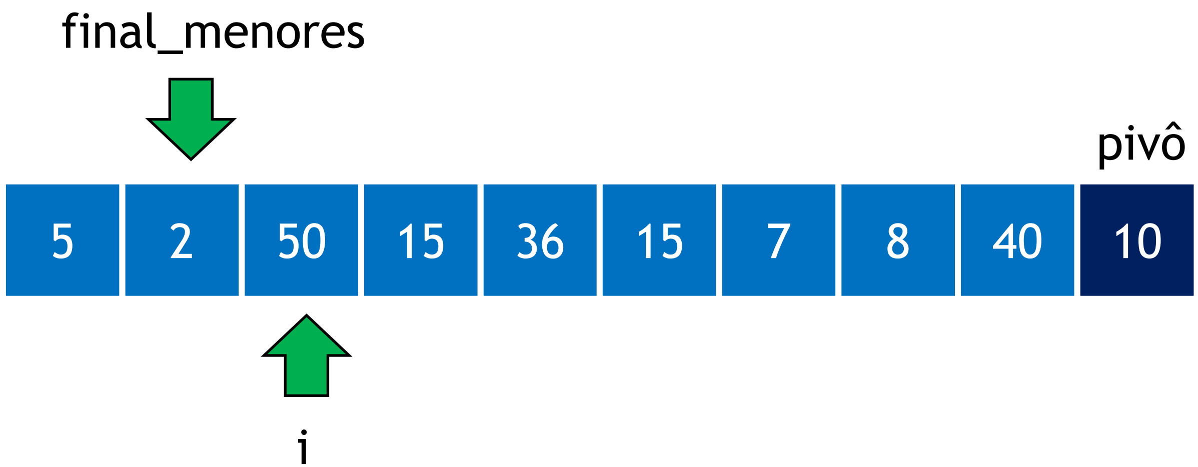
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



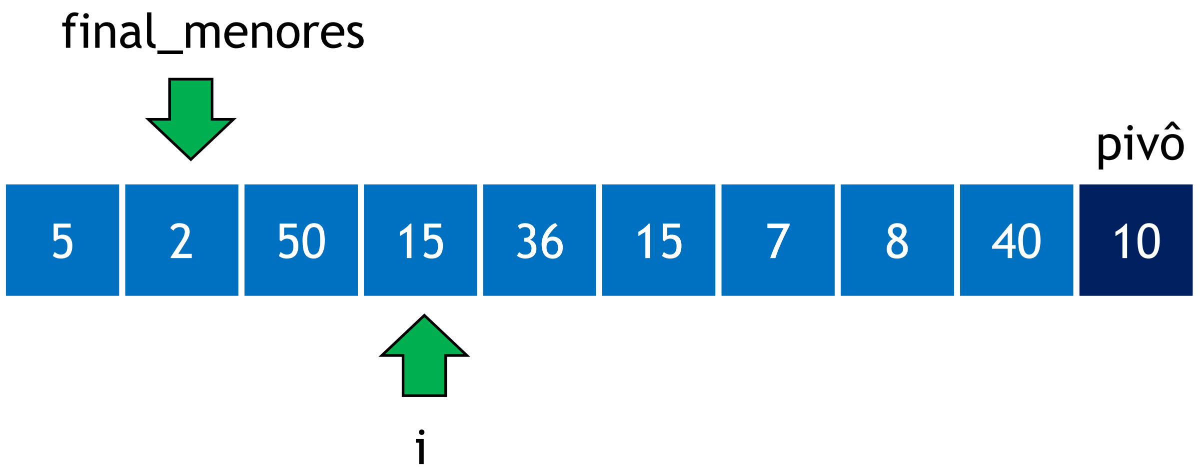
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



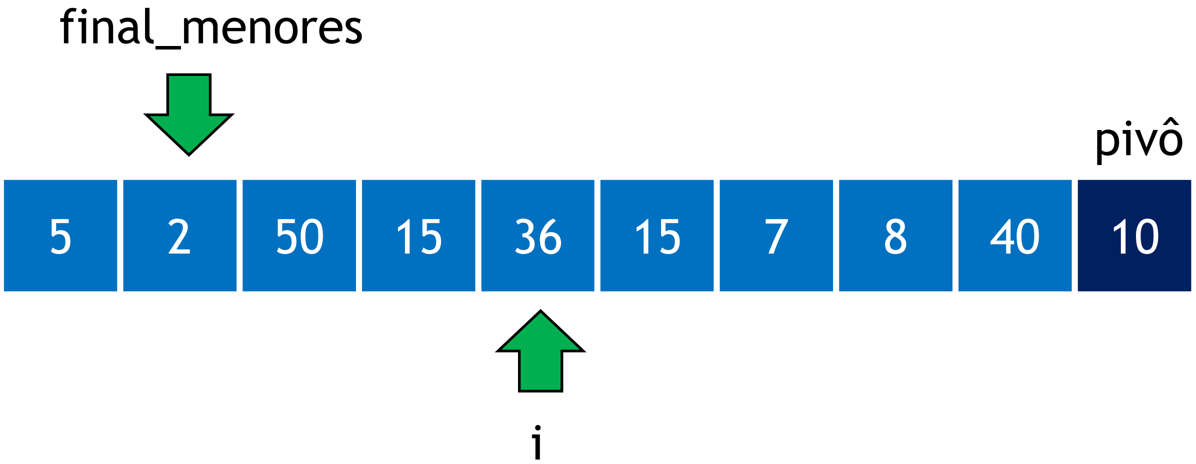
v[i] < pivô ? Não. Então não altera *final_menores*.

Particionamento



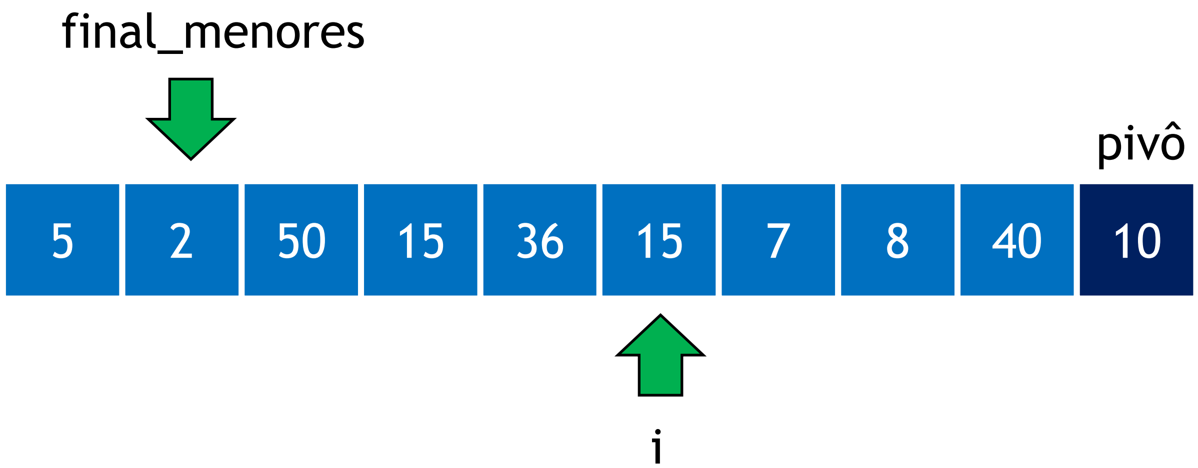
v[i] < pivô ? Não. Então não altera *final_menores*.

Particionamento



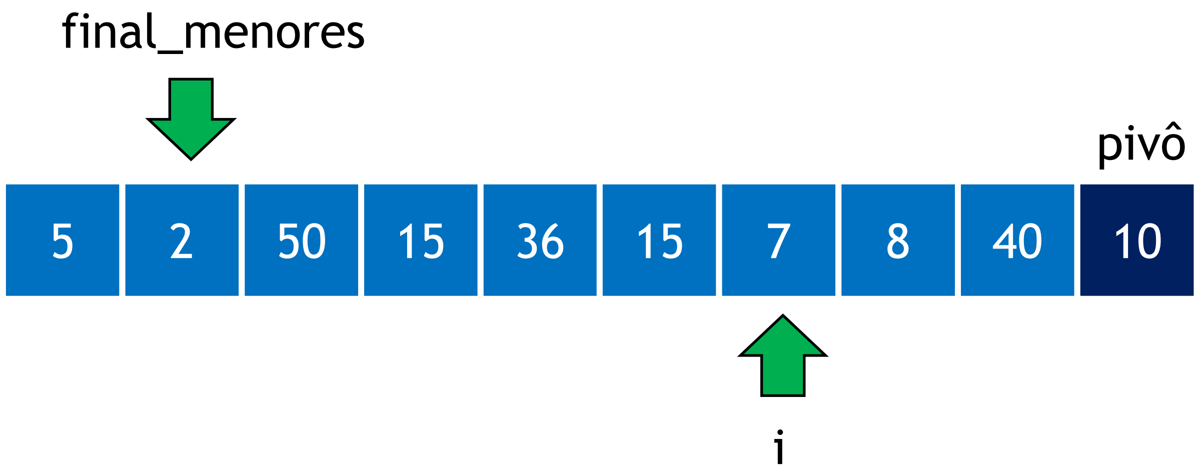
v[i] < pivô ? Não. Então não altera *final_menores*.

Particionamento



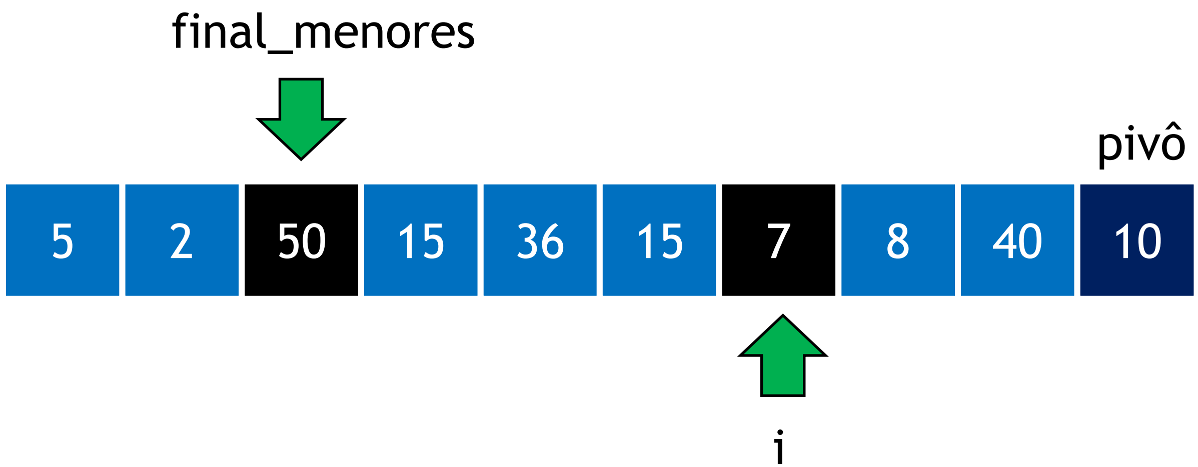
v[i] < pivô ? Não. Então não altera *final_menores*.

Particionamento



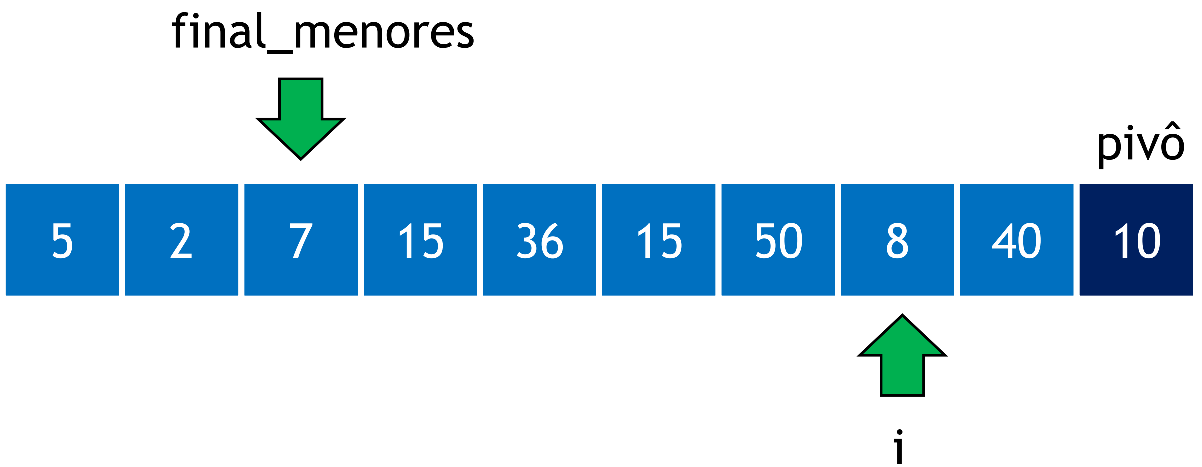
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



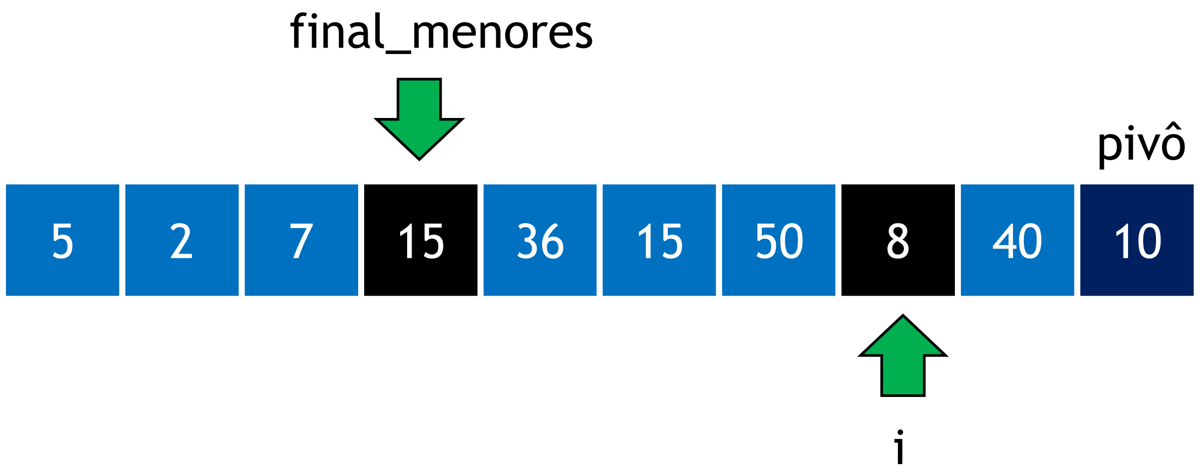
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



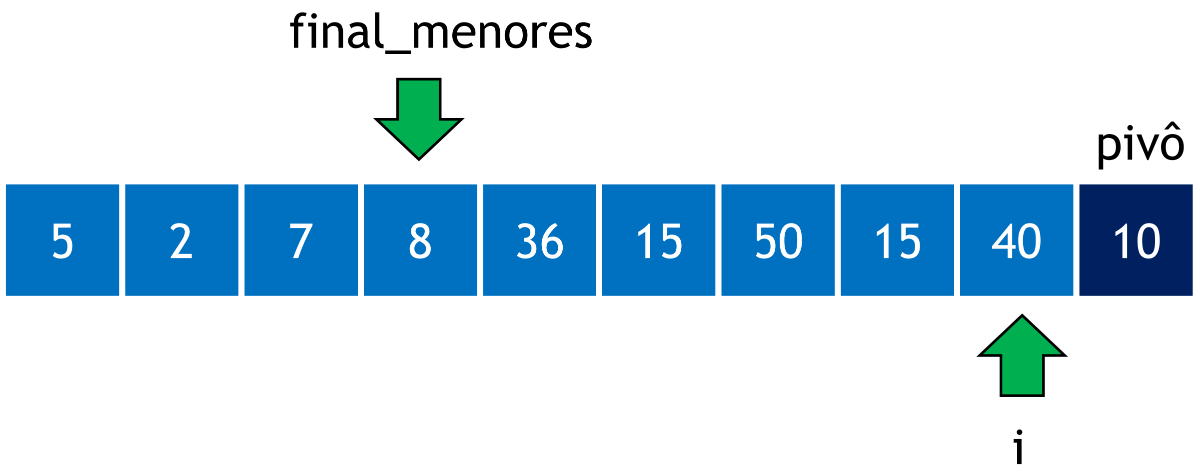
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



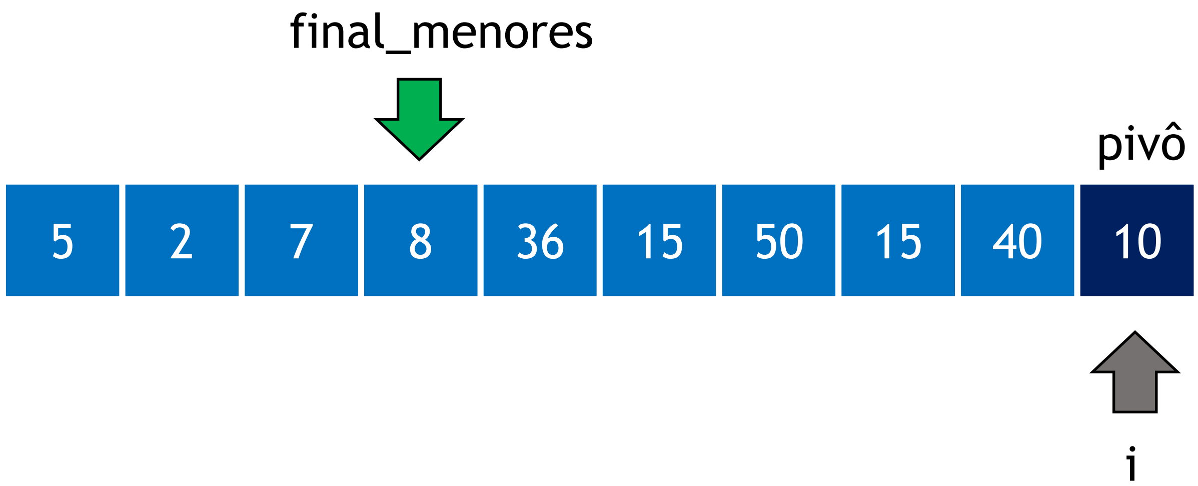
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



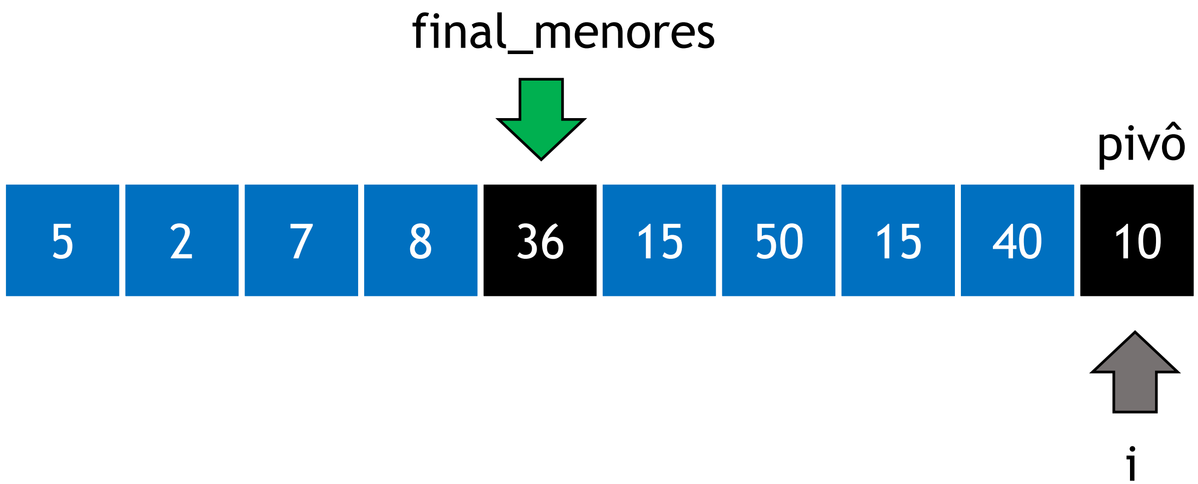
v[i] < pivô ? Sim.
Incrementa *final_menores* e troca v[i] por [final_menores].

Particionamento



Procedimento segue até o penúltimo elemento. Quando termina, incrementa final_menores e troca v[final_menores] com v[i] (pivô).

Particionamento



Procedimento segue até o penúltimo elemento. Quando termina, incrementa final_menores e troca v[final_menores] com v[i] (pivô).

Particionamento

				pivô					
5	2	7	8	10	15	50	15	40	36

Agora o particionamento está encerrado.

Observe que o pivô ficou no índice correto (ordenação) e todos os elementos à esquerda são menores e os à direita são maiores ou iguais ao pivô.

Particionamento

- Implementação em C

Chamada:
```
int indice_pivo = particiona(vetor, esq, dir);
```

Implementação

```
int particiona(int *v, int esq, int dir) {
    int pivo = v[dir];

    int i, ultimo_menores = esq-1;
    for (i = esq; i < dir; i++)
        if (v[i] < pivo) {
            ultimo_menores++;
            int tmp = v[i];
            v[i] = v[ultimo_menores];
            v[ultimo_menores] = tmp;
        }

    v[dir] = v[ultimo_menores+1];
    v[ultimo_menores+1] = pivo;
    return ultimo_menores+1;
}
```

Quick sort

- Escolhe pivô;
- Aplica particionamento;
- Ordena as subsequências à esquerda e à direita do pivô com quick sort.

Vetor inicial

2	8	4	6	7	3	1	5

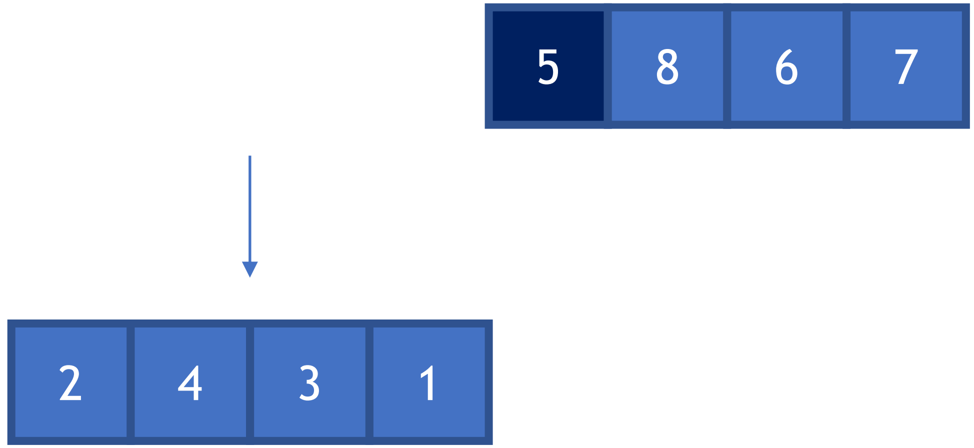
Escolhe pivô

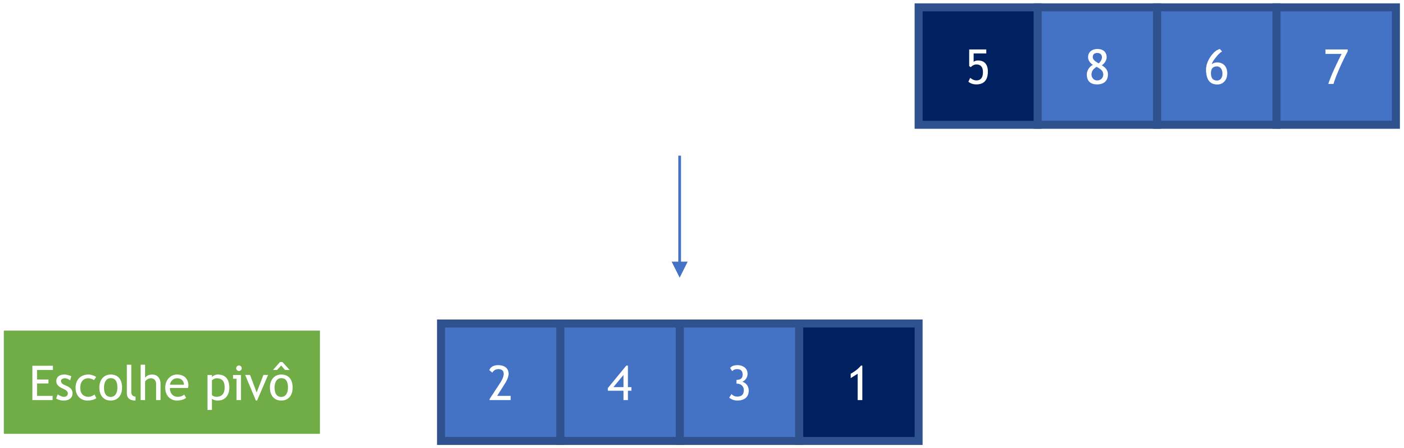
2	8	4	6	7	3	1	5

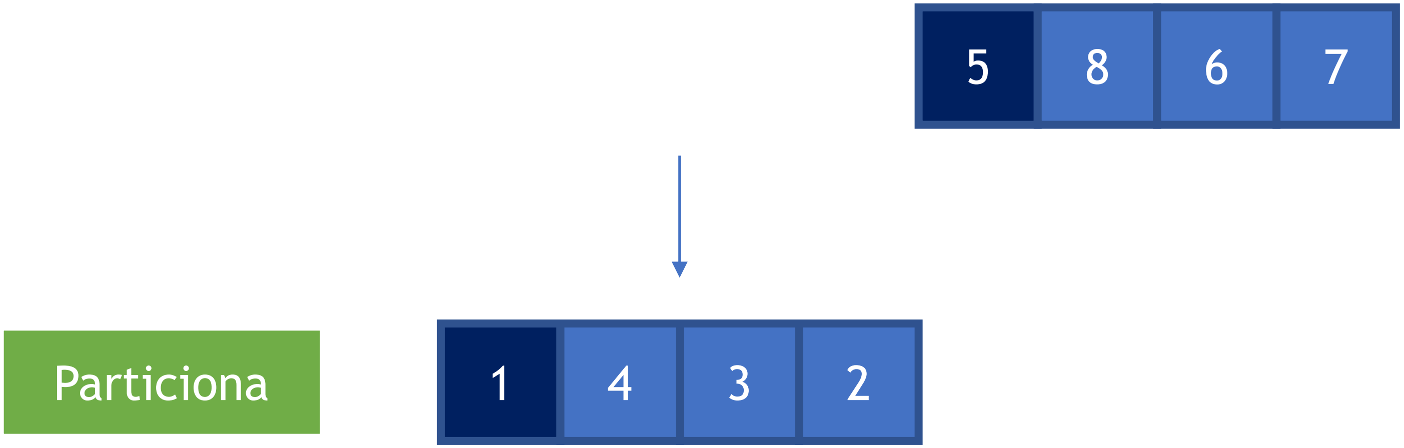
Particiona

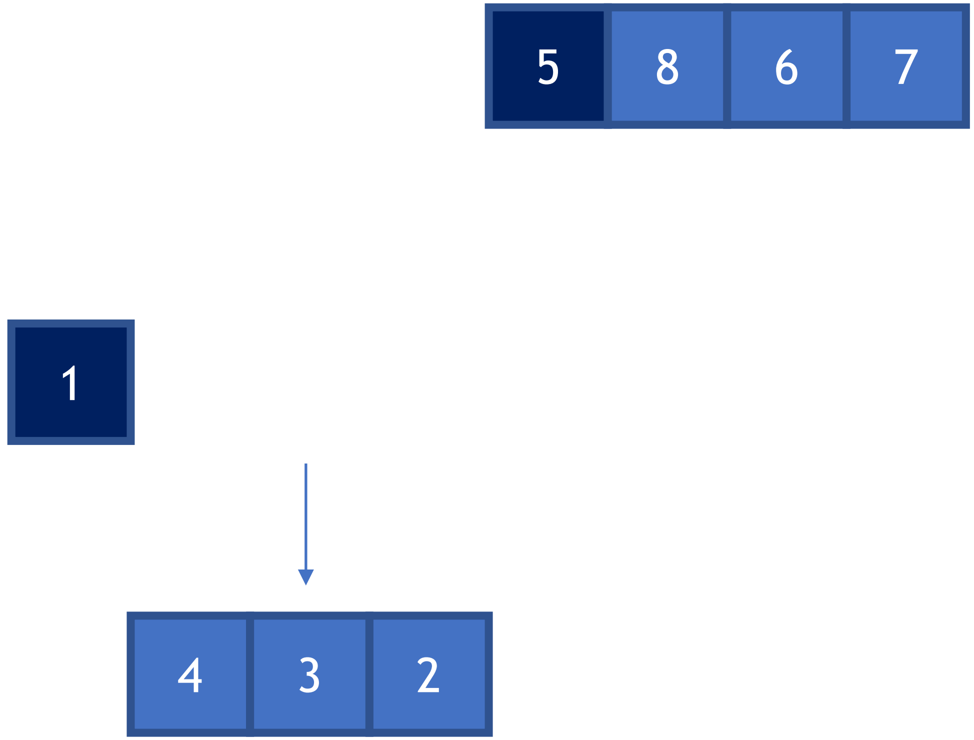
2	4	3	1	5	8	6	7

QS para as duas subsequências.

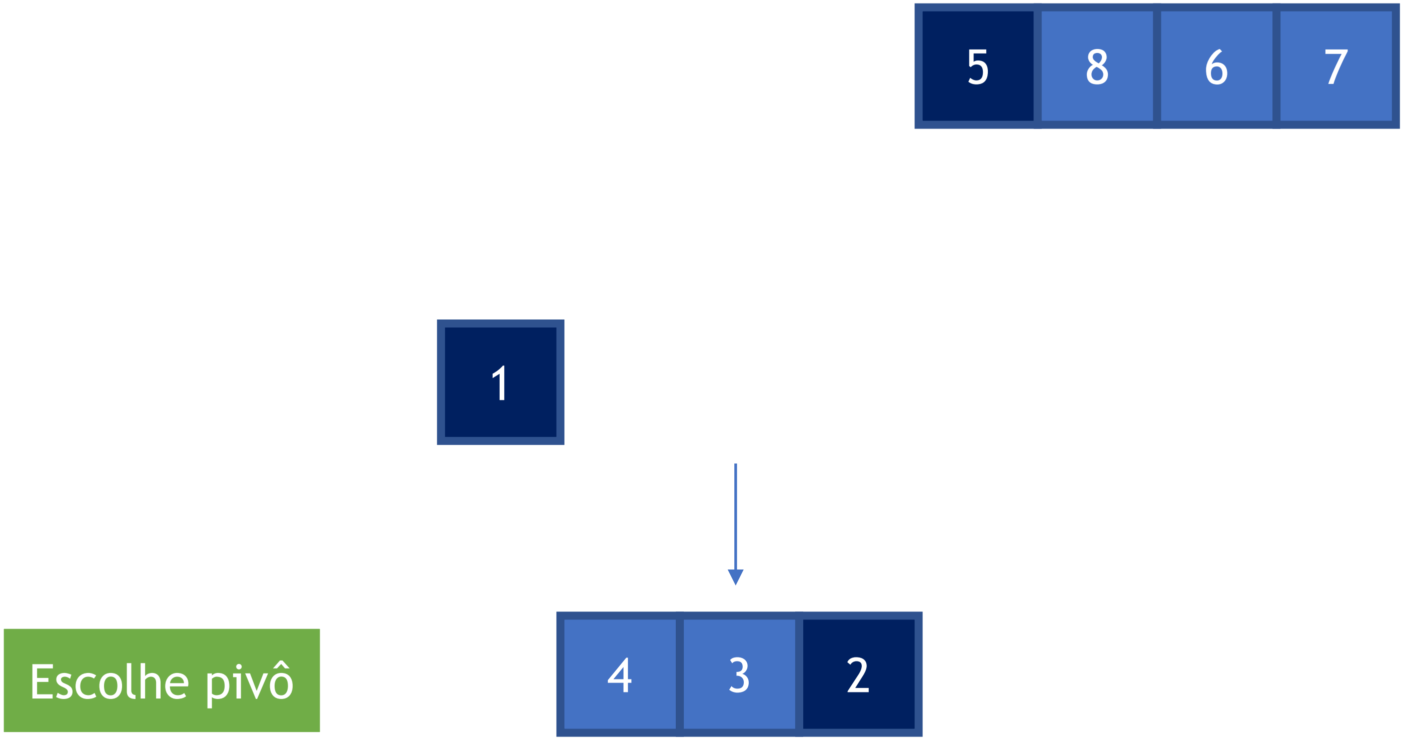


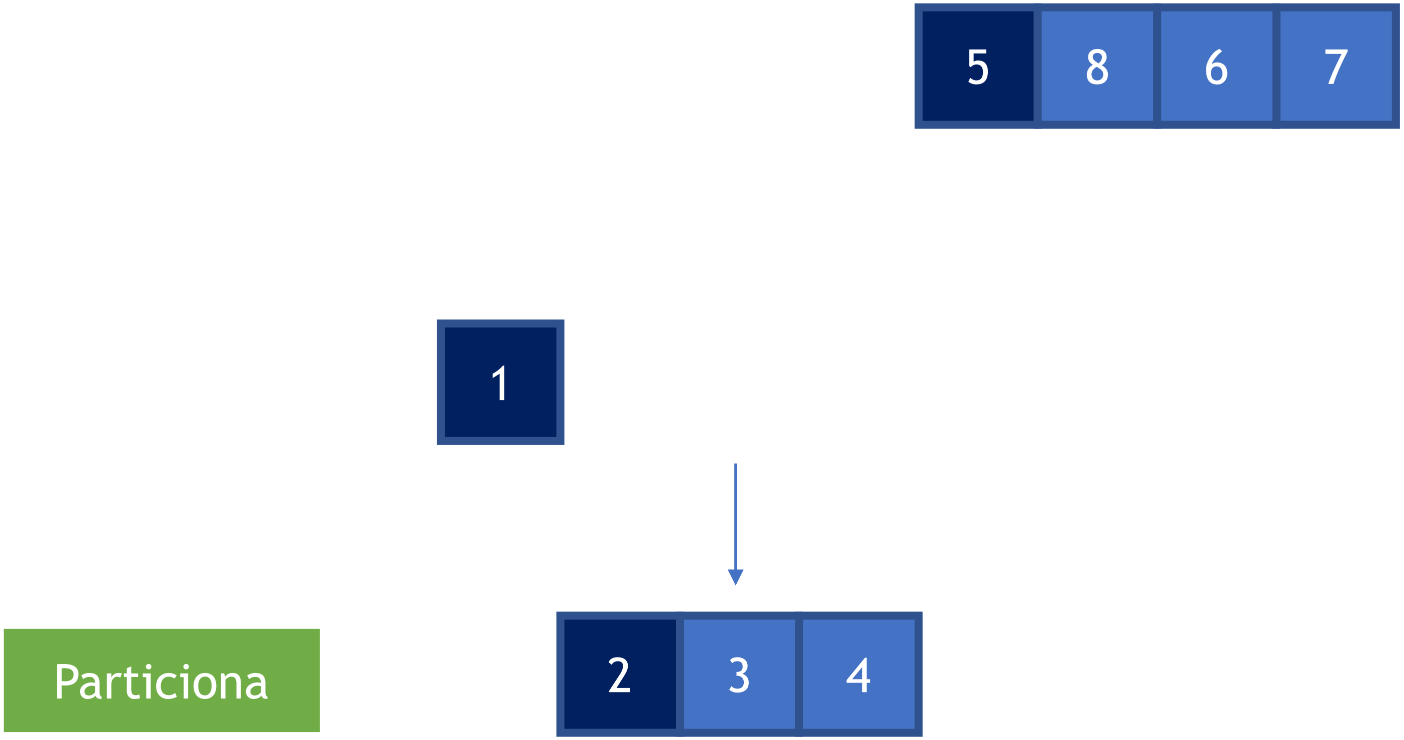

5 8 6 7
Escolhe pivô
2 4 3 1

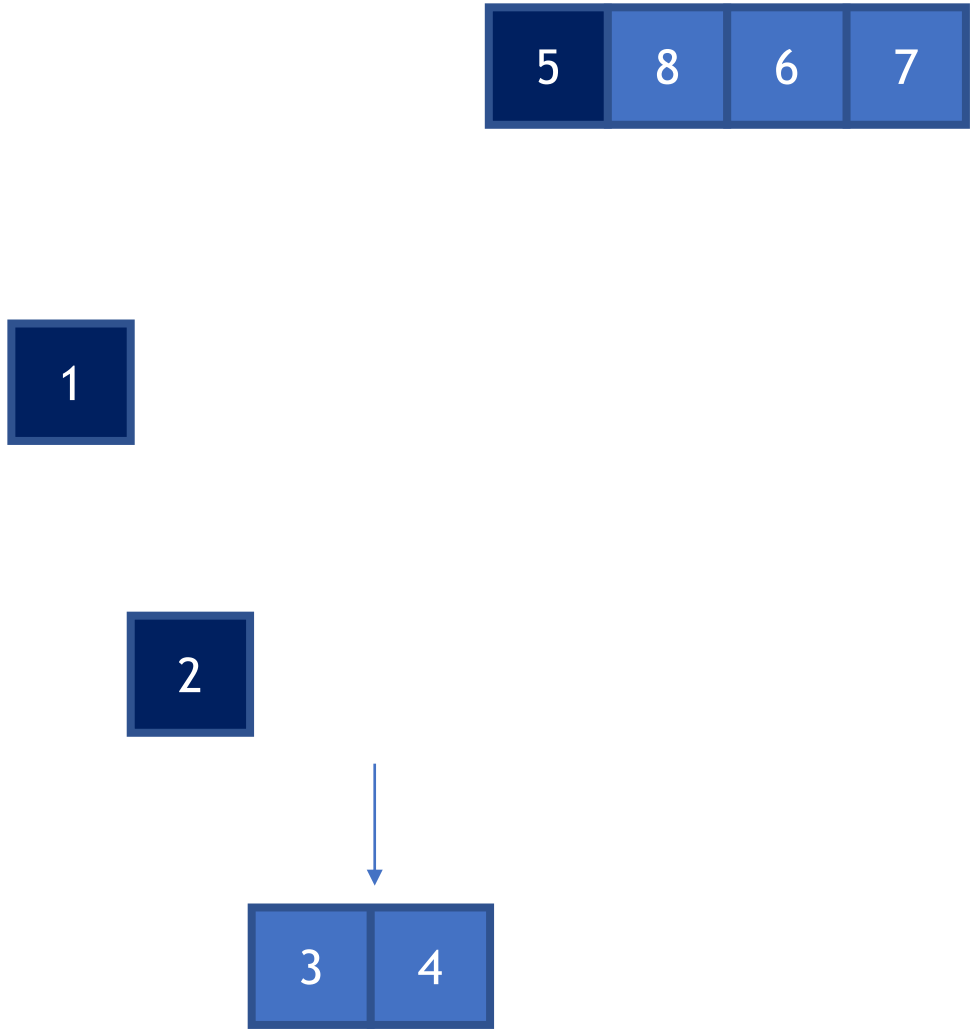

5 8 6 7
Particiona
1 4 3 2



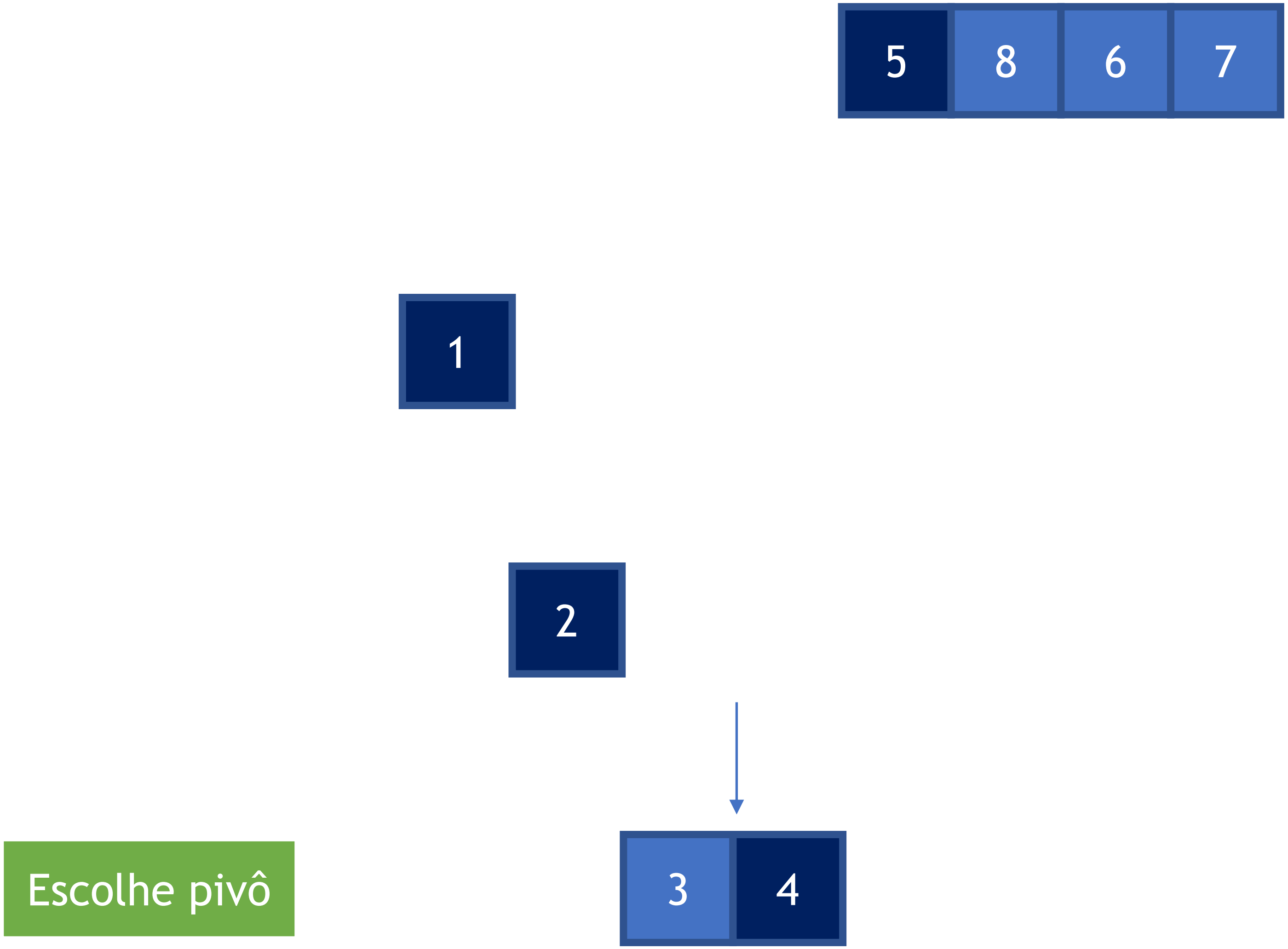
QS para as duas subsequências.

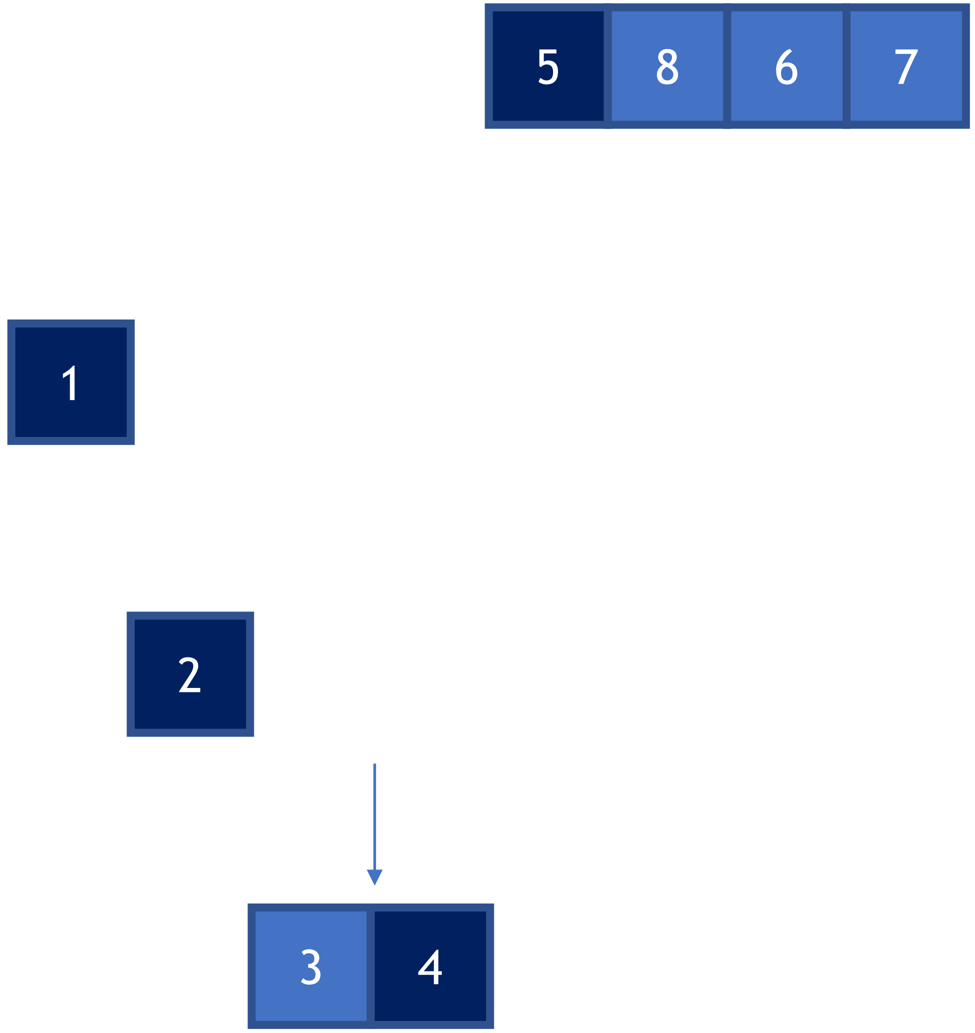

5
8
6
7
1
4
3
2
Escolhe pivô


5
8
6
7
1
Particiona
2
3
4

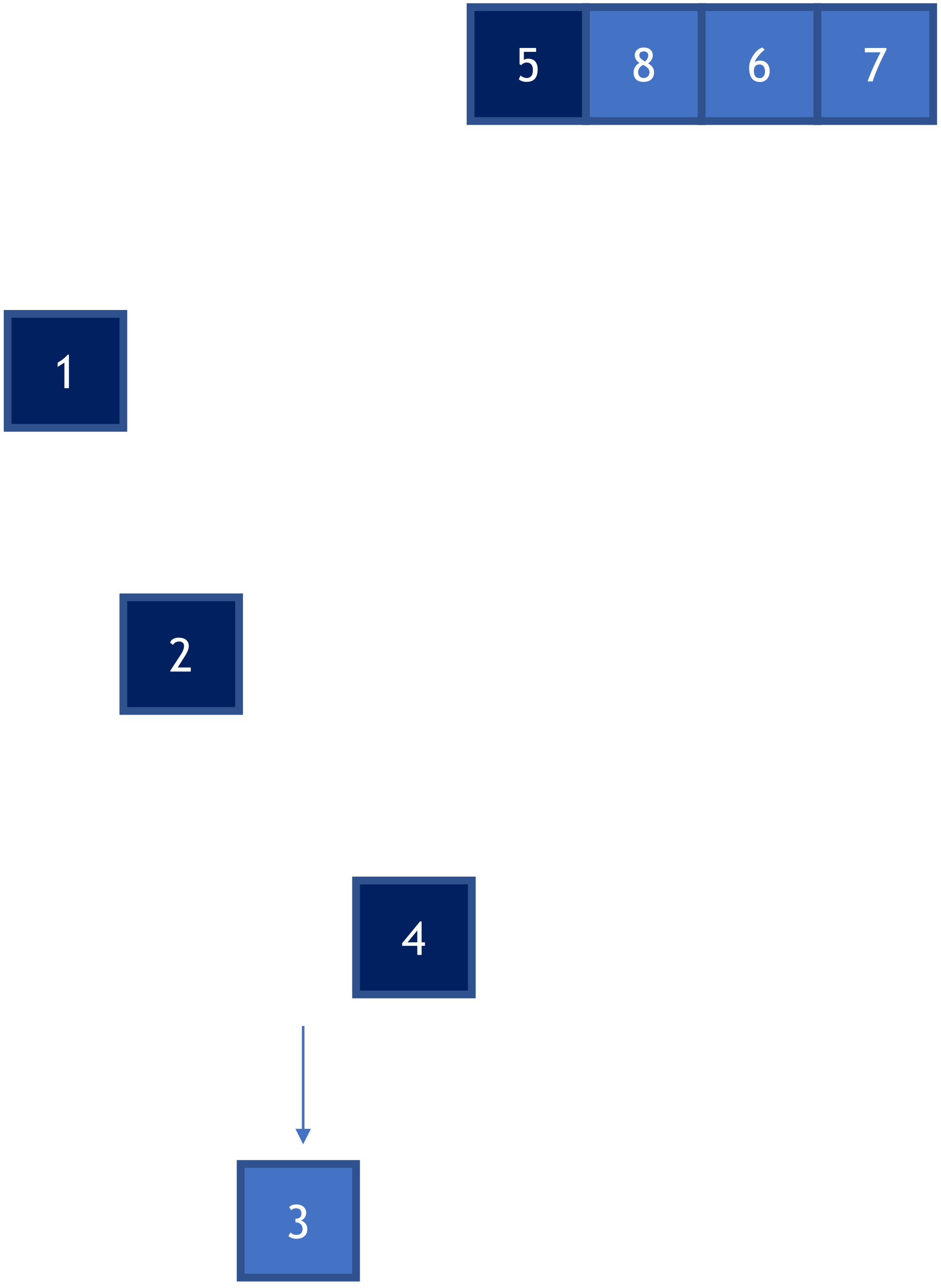


QS para as duas subsequências.

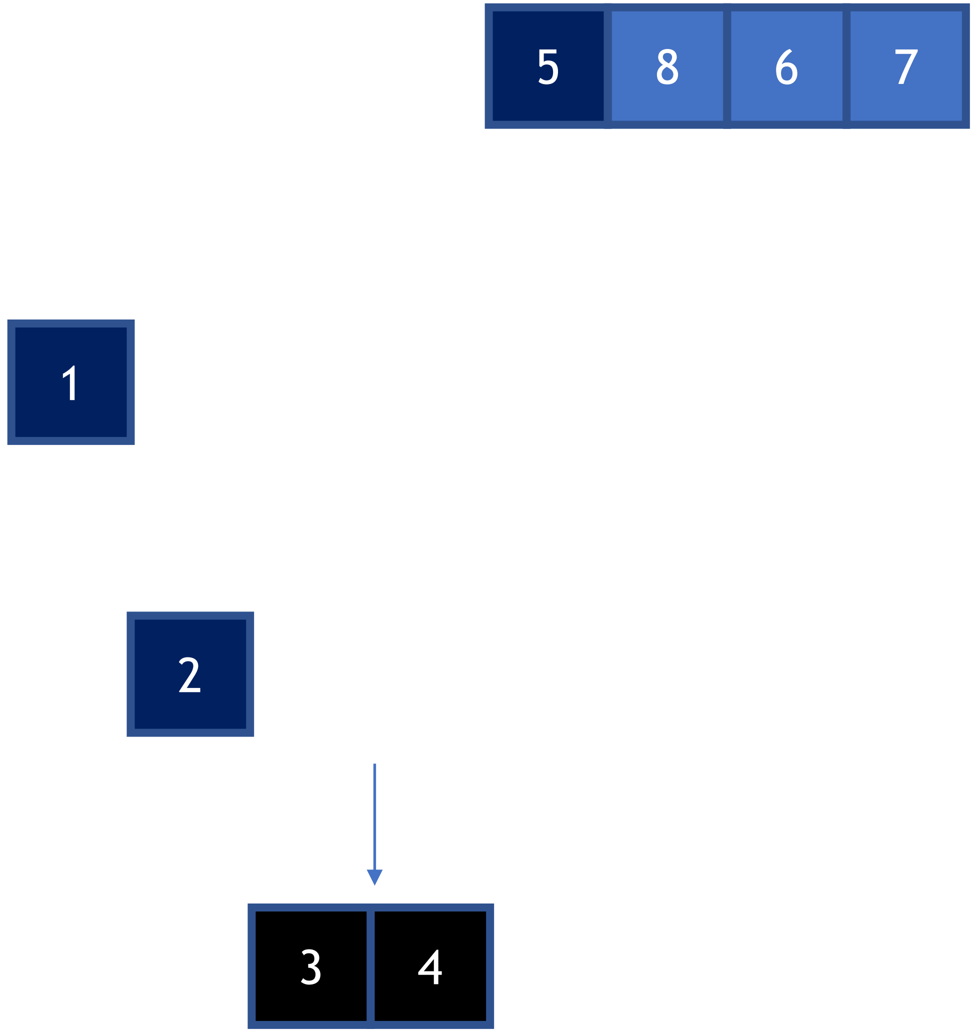

5
8
6
7
1
2
3
4
Escolhe pivô

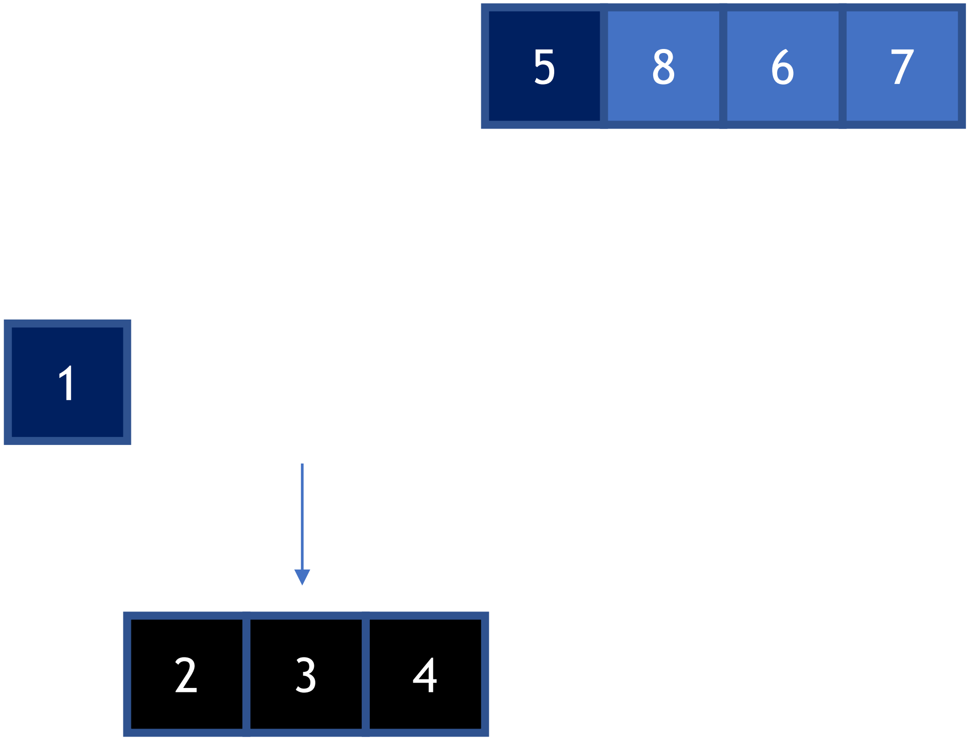


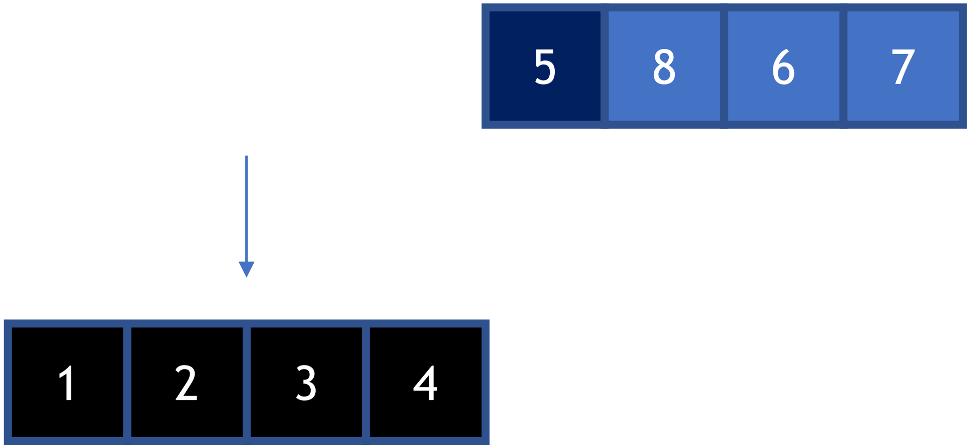
Particiona



QS para as duas subsequências.


5
8
6
7
1
2
3
4


5
8
6
7
1
2
3
4

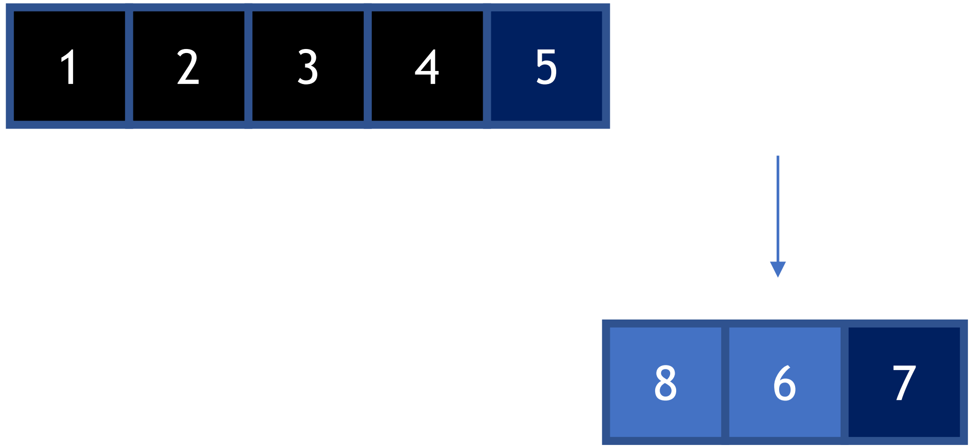

5
8
6
7
1
2
3
4

1	2	3	4	5	8	6	7

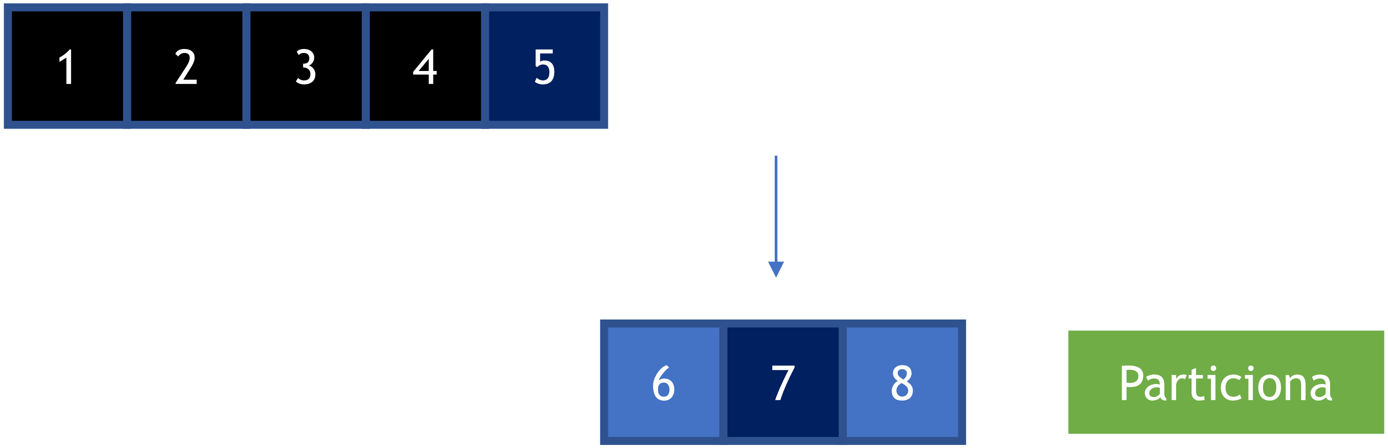
1	2	3	4	5

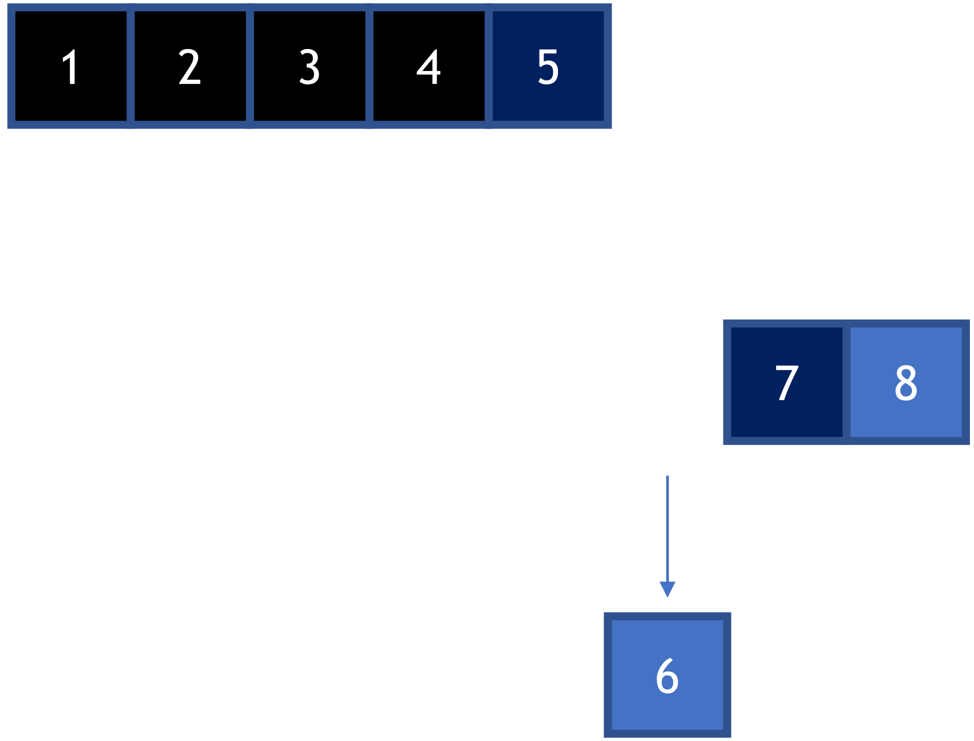
8	6	7

QS para as duas subsequências.

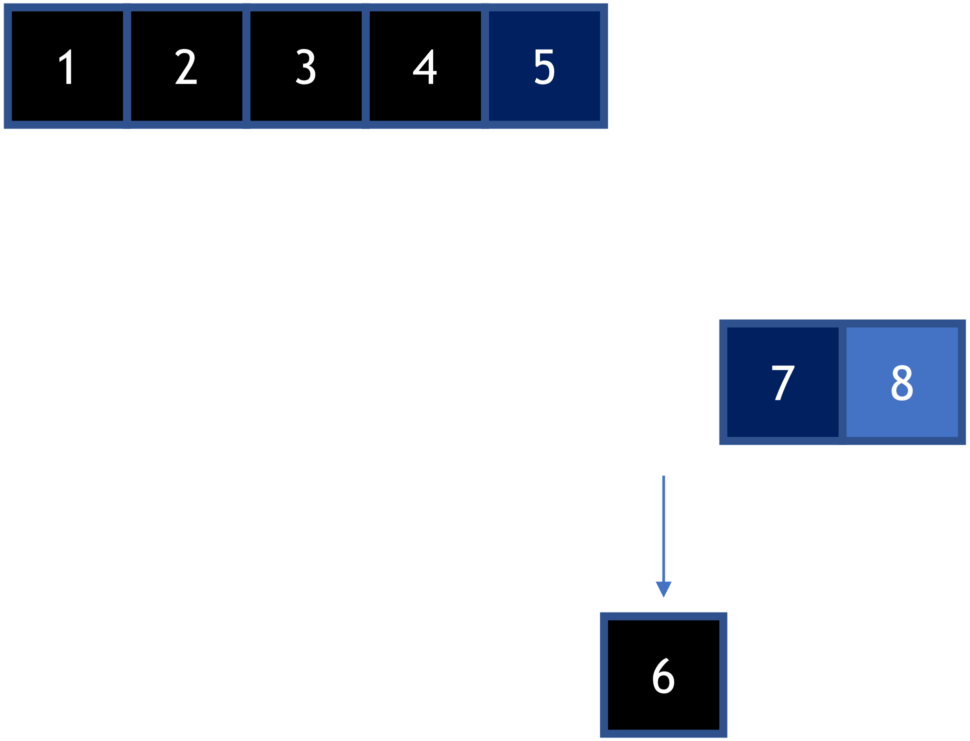


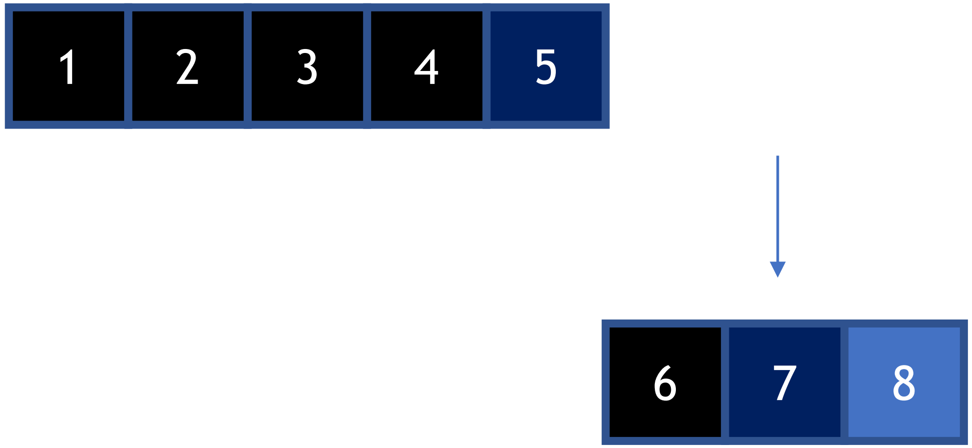
Escolhe pivô

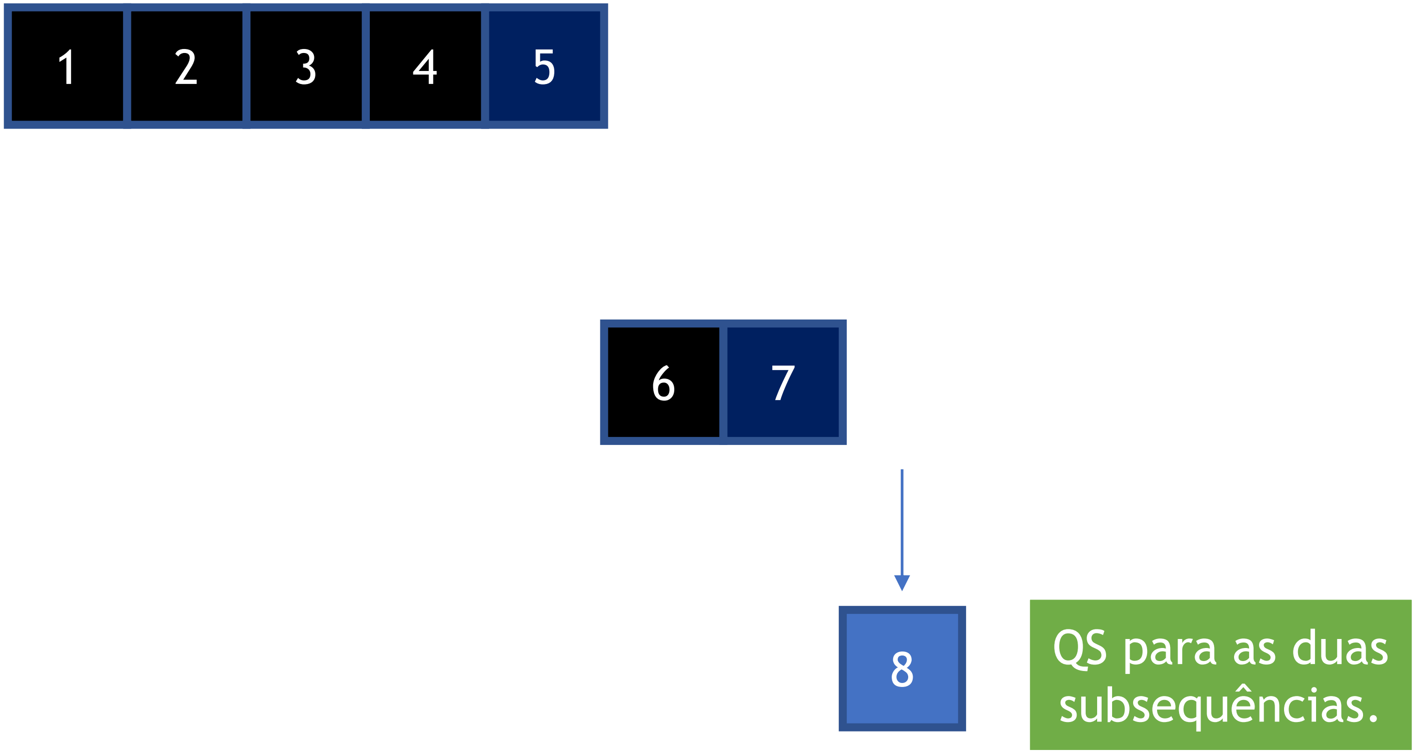

1
2
3
4
5
6
7
8
Particiona

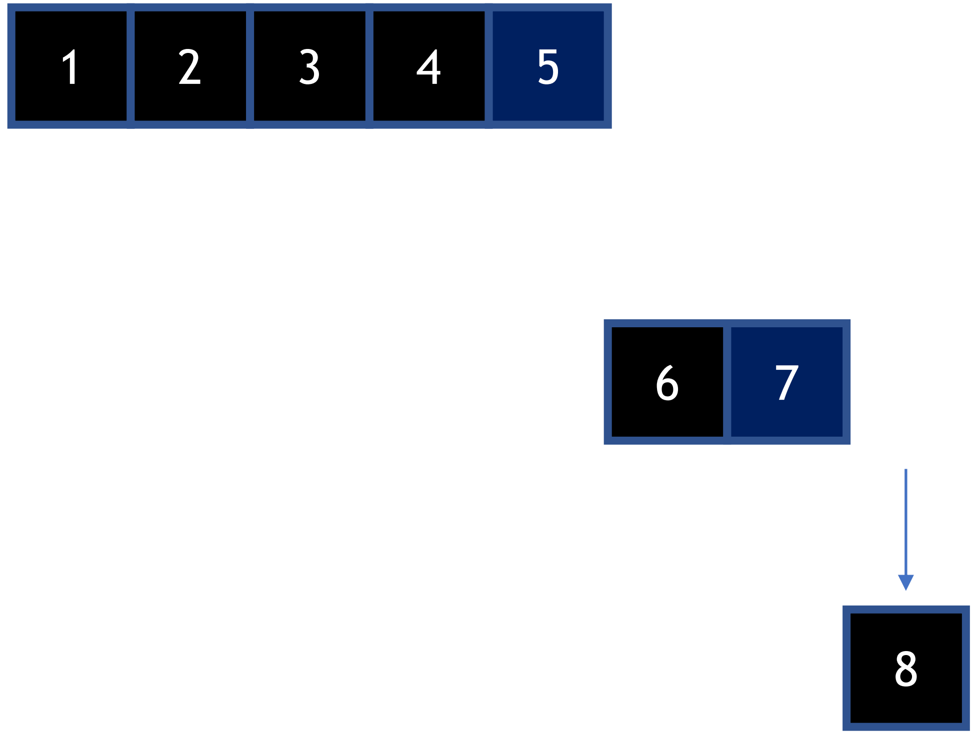


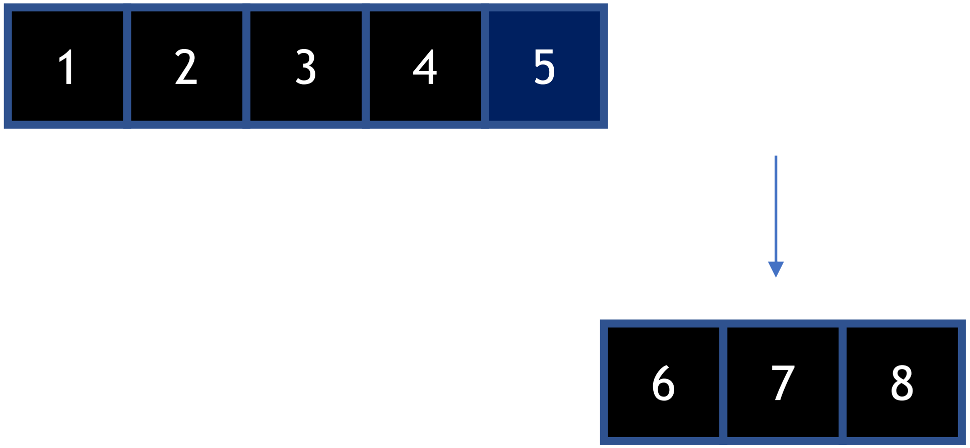
QS para as duas subsequências.


1
2
3
4
5
7
8
6


1
2
3
4
5
6
7
8


1
2
3
4
5
6
7
8
QS para as duas subsequências.


1
2
3
4
5
6
7
8


1
2
3
4
5
6
7
8

1	2	3	4	5	6	7	8

Ordenação finalizada.

Quick sort

- Implementação em C

Chamada:
```
quicksort(vetor, n);
```

Implementação

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}

void quicksort(int *v, int n) {
    quick_sort(v, 0, n - 1);
}
```

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$T(n)$

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$\rightarrow T(n)$

$\rightarrow O(n)$

$\rightarrow$?

$\rightarrow$?

Complexidade varia de acordo com o resultado do particiona

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$T(n)$

$O(n)$

?

?

Complexidade varia de acordo com o resultado do particiona

Pior caso: pivô fica na primeira ou última posição após o particionamento.

Melhor caso: pivô fica no meio do vetor após o particionamento.

Pior caso

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$T(n)$

$O(n)$

$$T(n) = \begin{cases} O(1), & n = 1 \\ T(n-1) + T(1) + O(n), & n > 1 \end{cases}$$

Pior caso

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$\longrightarrow T(n)$

$\longrightarrow O(n)$

$$T(n) = \begin{cases} O(1), & n = 1 \\ T(n-1) + O(n), & n > 1 \end{cases}$$

Pior caso

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$T(n)$

$O(n)$

$$T(n) = \begin{cases} c, & n = 1 \\ T(n-1) + c.n, & n > 1 \end{cases}$$

Pior caso

Custo do quick sort

$$T(n) = \begin{cases} c, & n = 1 \\ T(n-1) + c.n, & n > 1 \end{cases}$$

$$T(n) = T(n-1) + cn$$

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n) = T(n-2) + c(n-1) + cn$$

$$T(n-2) = T(n-3) + c(n-2)$$

$$T(n) = T(n-3) + c(n-2) + c(n-1) + cn$$

$$\vdots$$

$$T(n) = T(1) + \cdots + c(n-2) + c(n-1) + cn$$

$$T(n) = 1.c + \cdots + c(n-2) + c(n-1) + cn$$

$$T(n) = \frac{n.(1+n)}{2} = \frac{n^2+n}{2} \longrightarrow O(n^2)$$

Melhor caso

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$\rightarrow T(n)$

$\rightarrow O(n)$

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2.T(n/2) + O(n), & n > 1 \end{cases}$$

Melhor caso

Custo do quick sort

```
void quick_sort(int *v, int esq, int dir) {
    if (esq < dir) {
        int indice_pivo = particiona(v, esq, dir);
        quick_sort(v, esq, indice_pivo - 1);
        quick_sort(v, indice_pivo + 1, dir);
    }
}
```

$T(n)$

$O(n)$

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2.T(n/2) + O(n), & n > 1 \end{cases}$$

Já fizemos a análise de uma função similar no merge sort e sabemos que chegamos ao custo O(n.lg(n))

Custo do quick sort

- Portanto, chegamos ao seguinte custo (de tempo) para o quick sort:
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n.\lg(n))$
  - O caso médio também é $O(n.\lg(n))$

Mais detalhes em:

SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3a edição. Rio de Janeiro, RJ: LTC, 2012.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 3ª edição. Rio de Janeiro, RJ: Elsevier, 2012.

Resumo dos algoritmos

	Selection sort	Insertion sort	Merge sort	Quick sort
Pior caso	$O(n^2)$	$O(n^2)$	$O(n.\lg(n))$	$O(n^2)$
Caso médio	$O(n^2)$	$O(n^2)$	$O(n.\lg(n))$	$O(n.\lg(n))$
Melhor caso	$O(n^2)$	$O(n)$	$O(n.\lg(n))$	$O(n.\lg(n))$

Referências

- Slides do Prof. Monael Pinheiro Riberio:
  - https://sites.google.com/site/aed2019q1/
- Slides da Profa. Mirtha Lina Fernández Venero
  - Algoritmos e Estruturas de Dados I – 2019
- Slides do Prof. Fabrício Olivetti de França
  - https://folivetti.github.io/courses/AEDI/

Bibliografia básica

- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3ª edição. São Paulo, SP: Prentice Hall, 2005.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2ª edição. Rio de Janeiro, RJ: Campus, 2002.

Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3a edição. Rio de Janeiro, RJ: LTC, 1994.
- TEWNENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.