MC-202 — Aula 7
Divisão e Conquista:
algoritmos de ordenação recursivos

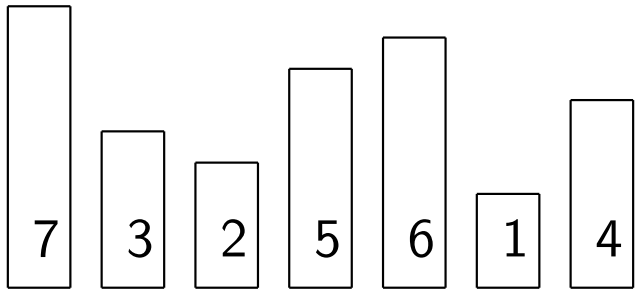
Lehilton Pedrosa

Instituto de Computação – Unicamp

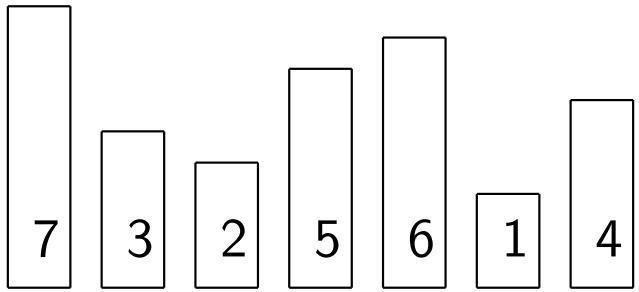
Segundo Semestre de 2015

Roteiro

Relembrando um algoritmo simples: ordenação por seleção



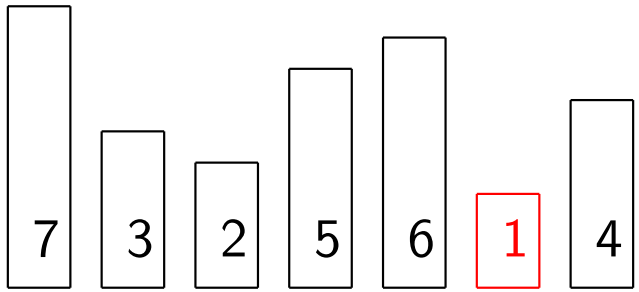
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:

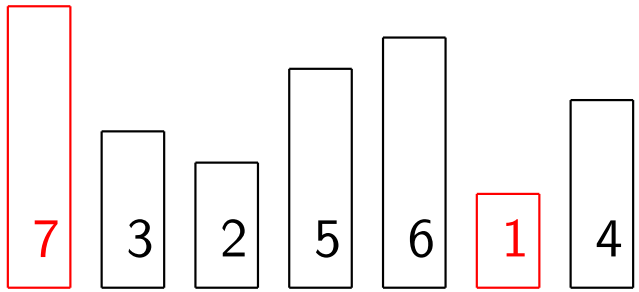
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento

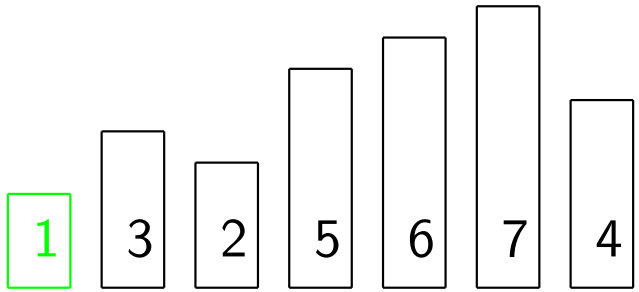
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista

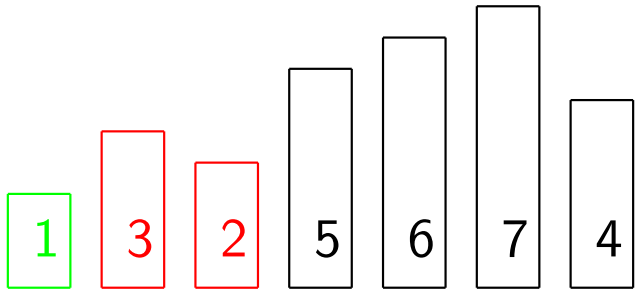
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

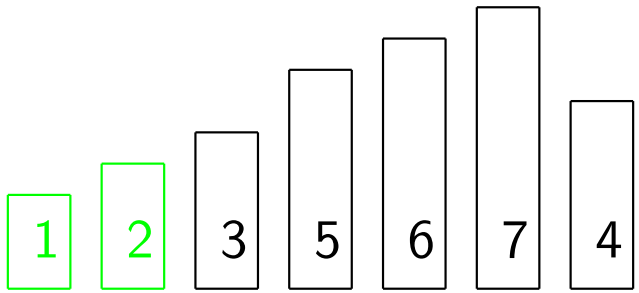
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

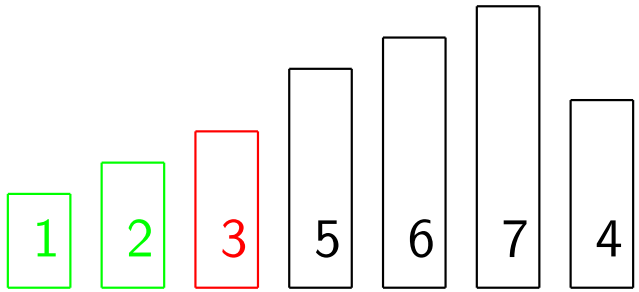
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

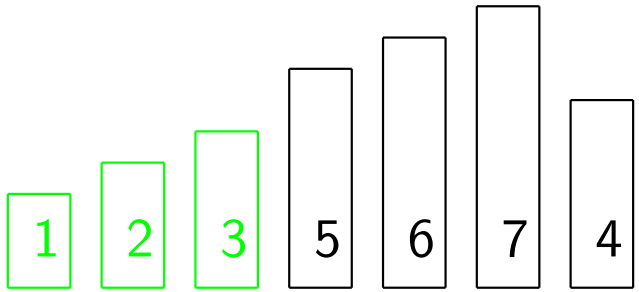
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

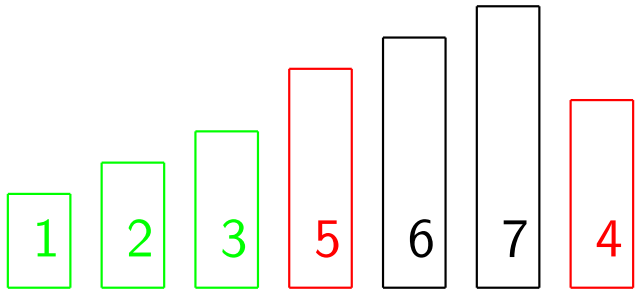
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

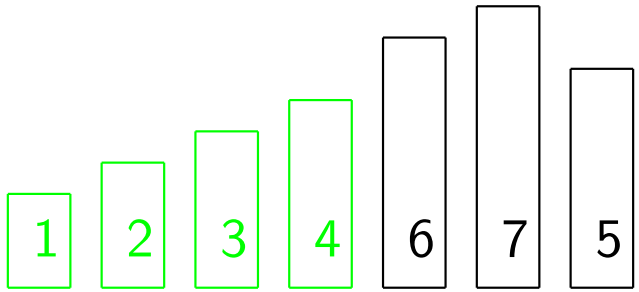
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

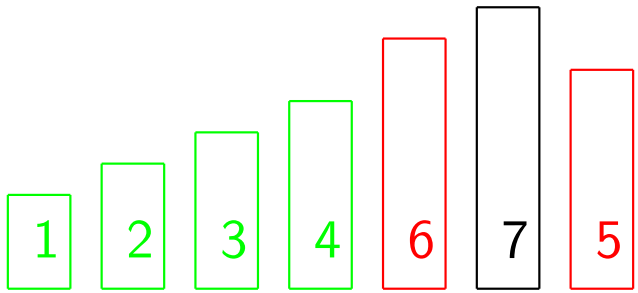
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

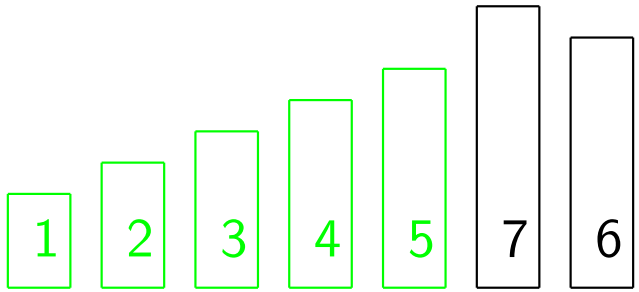
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

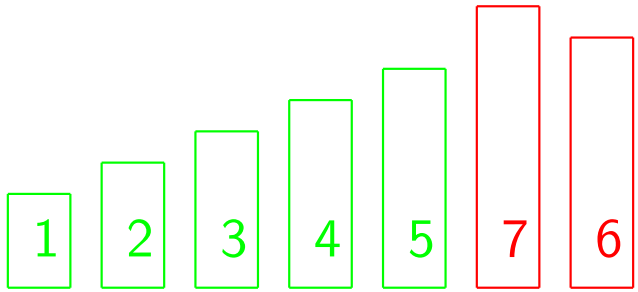
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

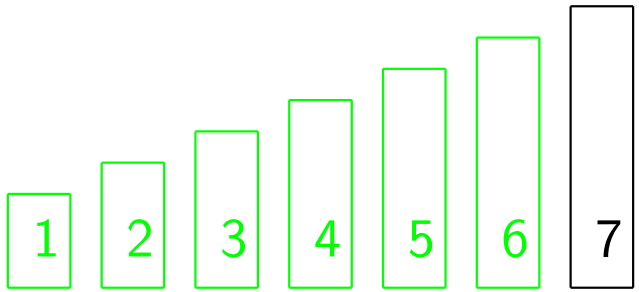
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

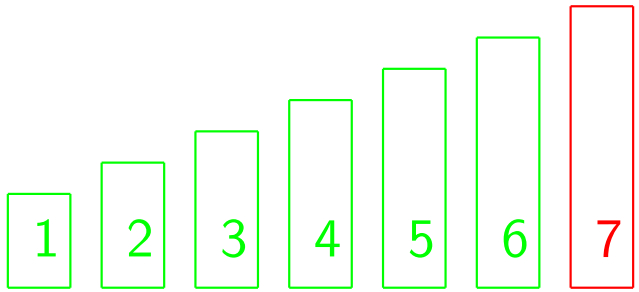
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

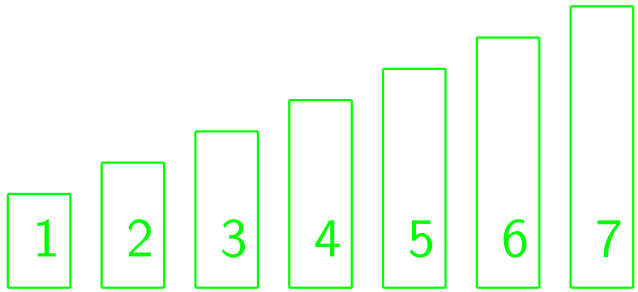
Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

Relembrando um algoritmo simples: ordenação por seleção



Ideia

- Repetimos n vezes:
  1. Selecionar o menor elemento
  2. Trocar com o primeiro elemento da lista
  3. Continuar com a lista restante (preta)

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Quantas comparações?

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Quantas comparações?

- chamamos `menor_elemento` *n* **vezes**

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Quantas comparações?

- chamamos `menor_elemento` n **vezes**
- cada chamada faz $n - i - 1$ **comparações**

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Quantas comparações?

- chamamos `menor_elemento` n **vezes**
- cada chamada faz $n - i - 1$ **comparações**

TOTAL: $\frac{n(n-1)}{2} = O(n^2)$

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

Quantas comparações?

- chamamos `menor_elemento` n **vezes**
- cada chamada faz $n - i - 1$ **comparações**

TOTAL: $\frac{n(n-1)}{2} = O(n^2)$

Isso é rápido?

Verificando a eficiência

Ordenação por seleção

```
int ordenar_selecao(int vetor[], int n) {
    int i, menor;
    for (i = 0; i < n; i++) {
        menor = menor_elemento(vetor, i, n);
        trocar(&vetor[i], &vetor[menor]);
    }
}
```

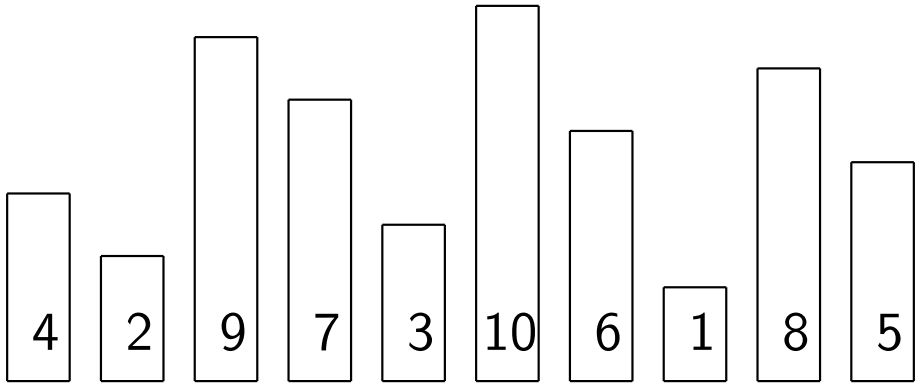
Quantas comparações?

- chamamos `menor_elemento` n **vezes**
- cada chamada faz $n - i - 1$ **comparações**

TOTAL: $\frac{n(n-1)}{2} = O(n^2)$

Isso é rápido? Se n for 1.000.000?

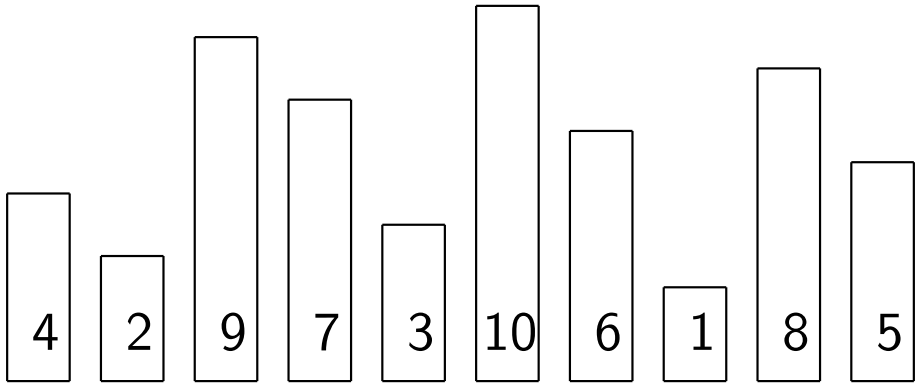
Outra estratégia: recursão



Problema 1

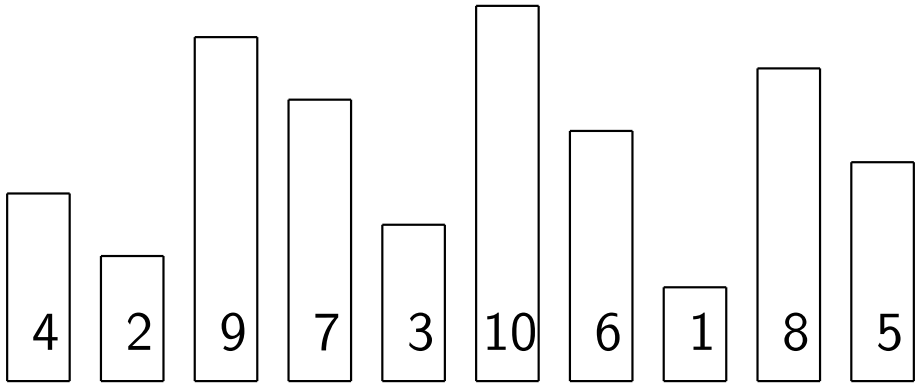
Suponha que temos um vetor desordenado com 10 números. Como ordenar a primeira metade da lista números?

Ordenando a primeira parte



Suponha que

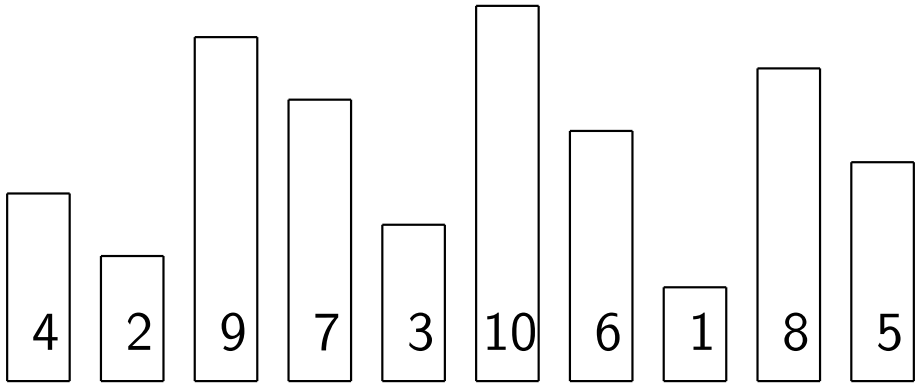
Ordenando a primeira parte



Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`

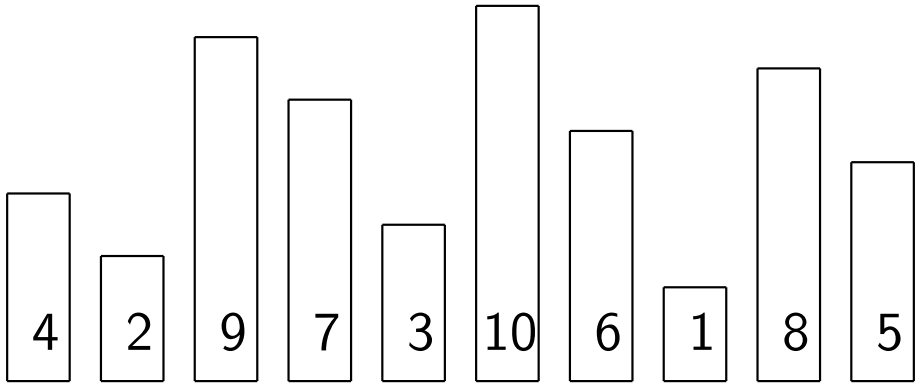
Ordenando a primeira parte



Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`
- ela ordena o vetor da posição *ini* até *fim*

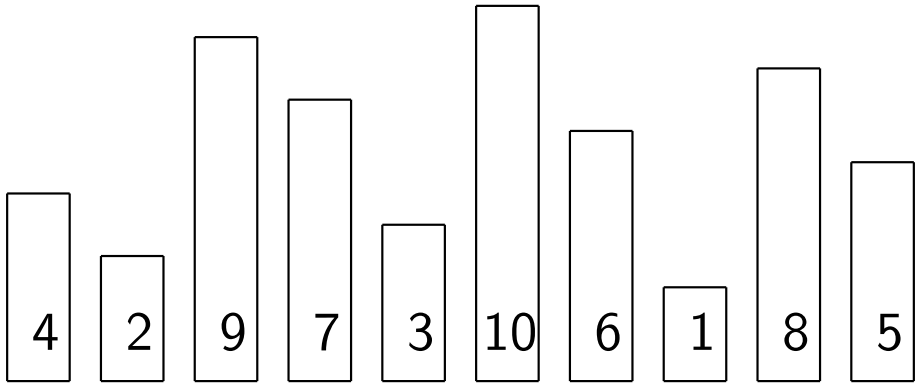
Ordenando a primeira parte



Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`
- ela ordena o vetor da posição *ini* até *fim*
- o vetor é indexado da posição 1 até 10

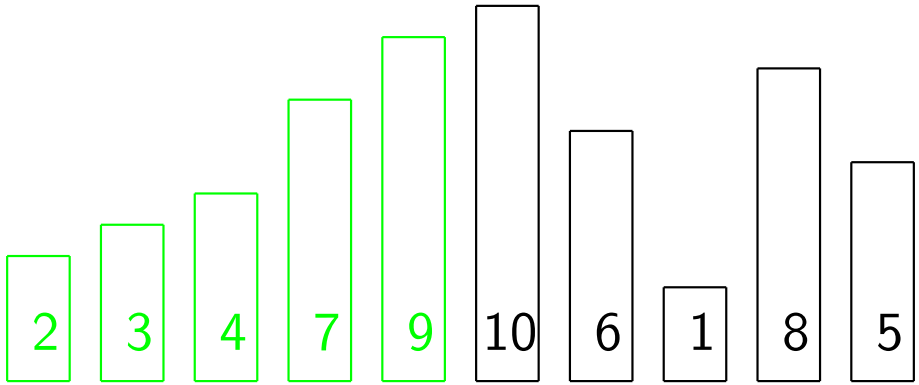
Ordenando a primeira parte



Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`
- ela ordena o vetor da posição *ini* até *fim*
- o vetor é indexado da posição 1 até 10
- executamos `ordenar(vetor, 1, 5);`

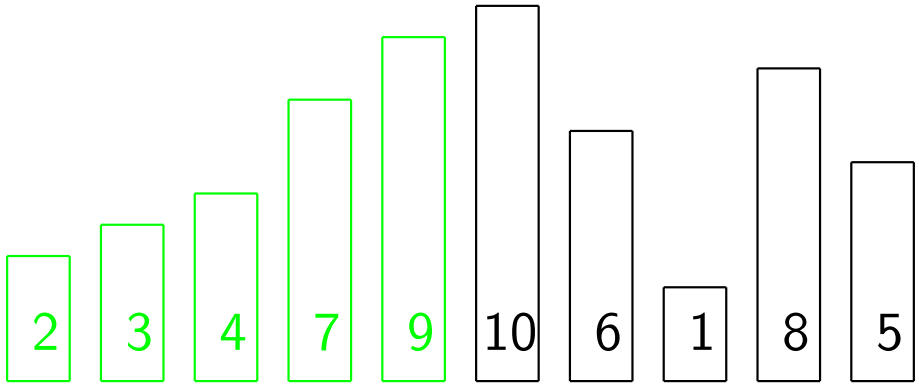
Ordenando a primeira parte



Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`
- ela ordena o vetor da posição *ini* até *fim*
- o vetor é indexado da posição 1 até 10
- executamos `ordenar(vetor, 1, 5);`

Ordenando a primeira parte

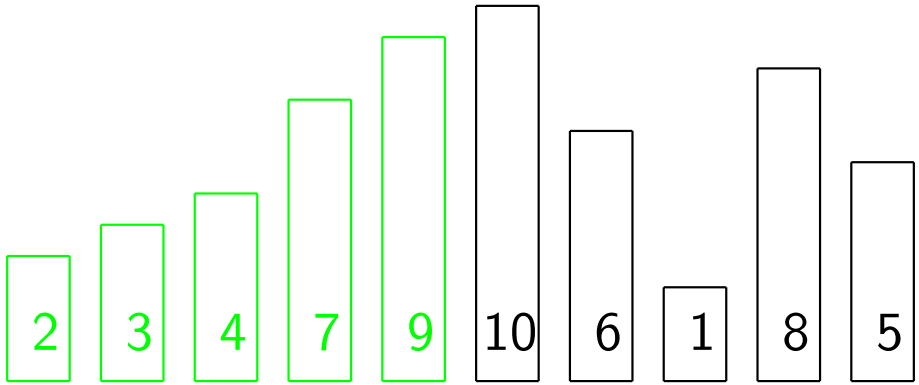


Suponha que

- temos uma função `ordenar(int vetor[], int ini, int fim)`
- ela ordena o vetor da posição *ini* até *fim*
- o vetor é indexado da posição 1 até 10
- executamos `ordenar(vetor, 1, 5);`

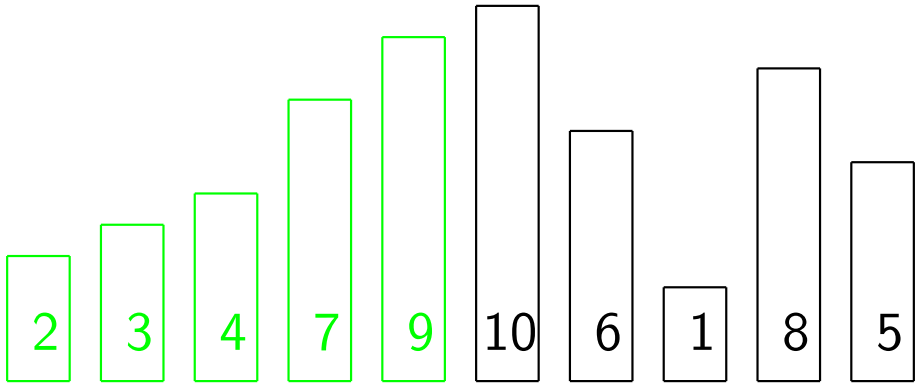
E se quiséssemos ordenar a segunda parte?

...e a segunda parte



Suponha que

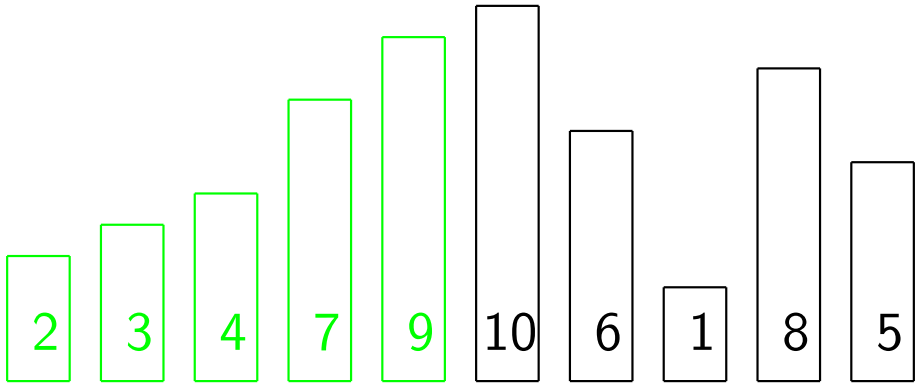
...e a segunda parte



Suponha que

- agora queremos ordenar a segunda parte

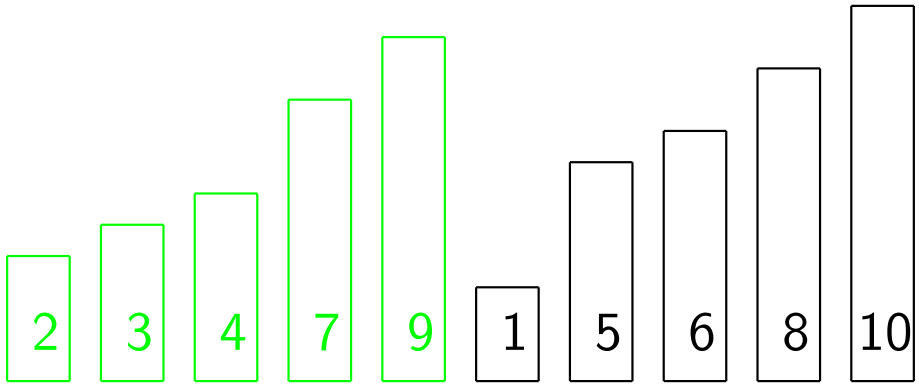
...e a segunda parte



Suponha que

- agora queremos ordenar a segunda parte
- executamos `ordenar(vetor, 6, 10);`

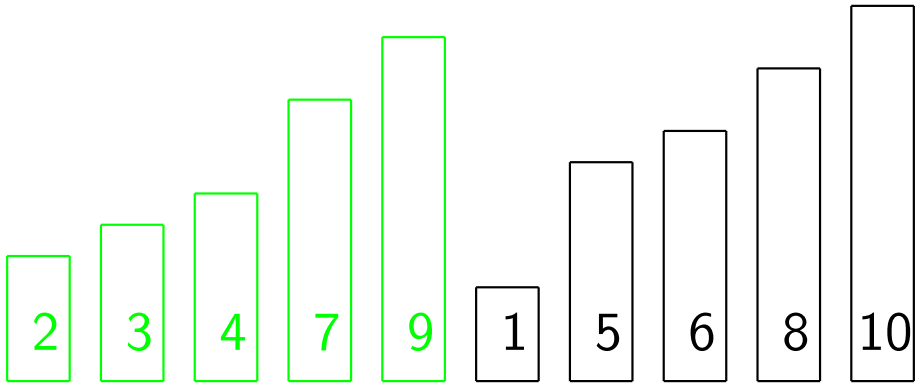
...e a segunda parte



Suponha que

- agora queremos ordenar a segunda parte
- executamos `ordenar(vetor, 6, 10);`

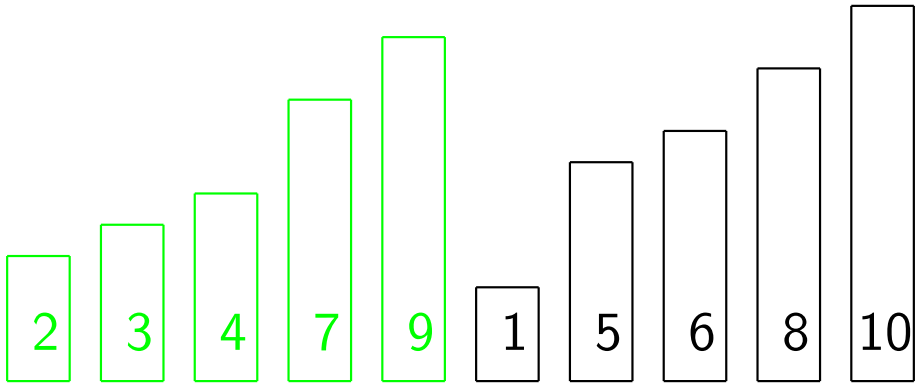
Ordenando tudo



Problema

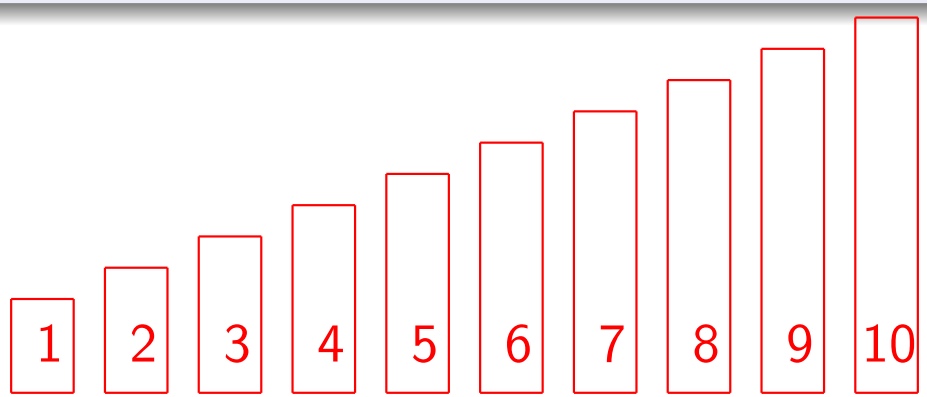
Suponha que temos um vetor de 10 números com as duas metades já ordenadas. Como criar um novo vetor com todos os elementos ordenados?

Ordenando tudo

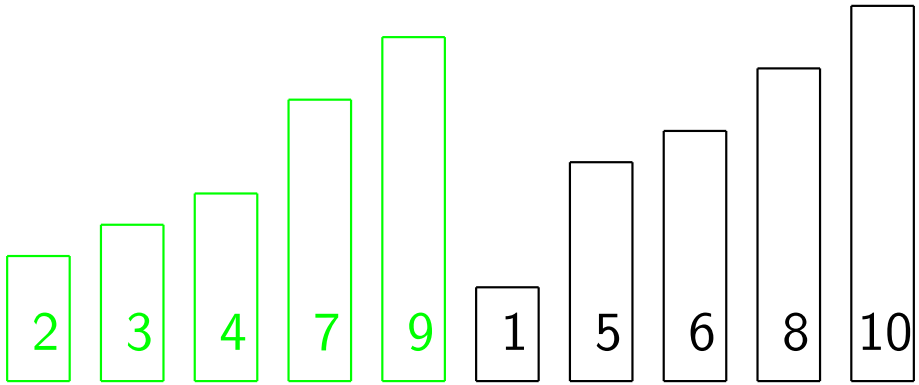


Problema

Suponha que temos um vetor de 10 números com as duas metades já ordenadas. Como criar um novo vetor com todos os elementos ordenados?

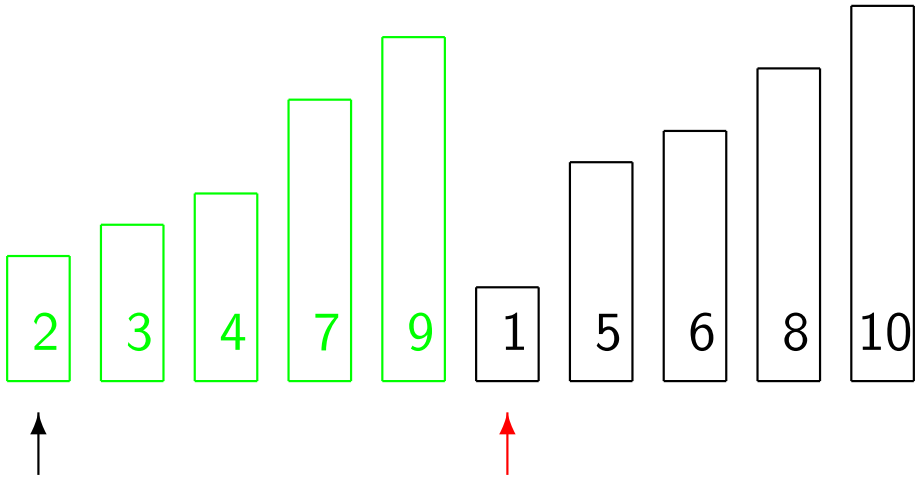


Intercalando



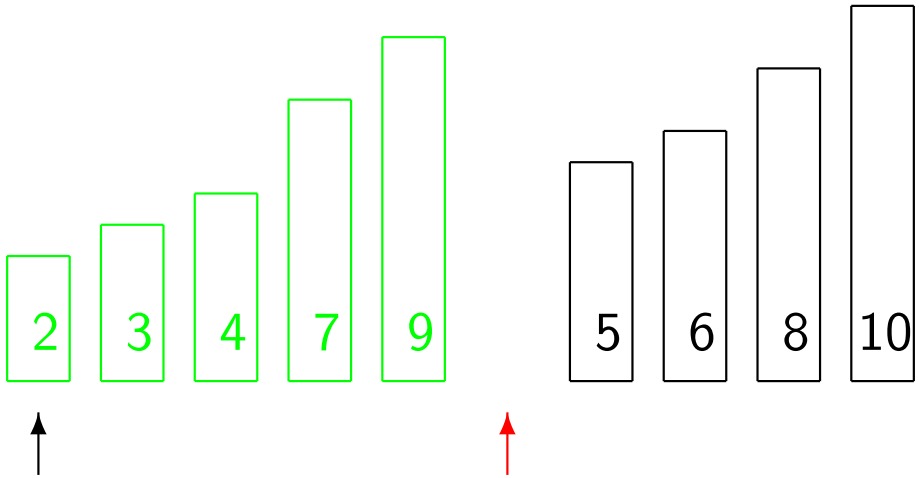
- Percorremos os dois subvetores,

Intercalando



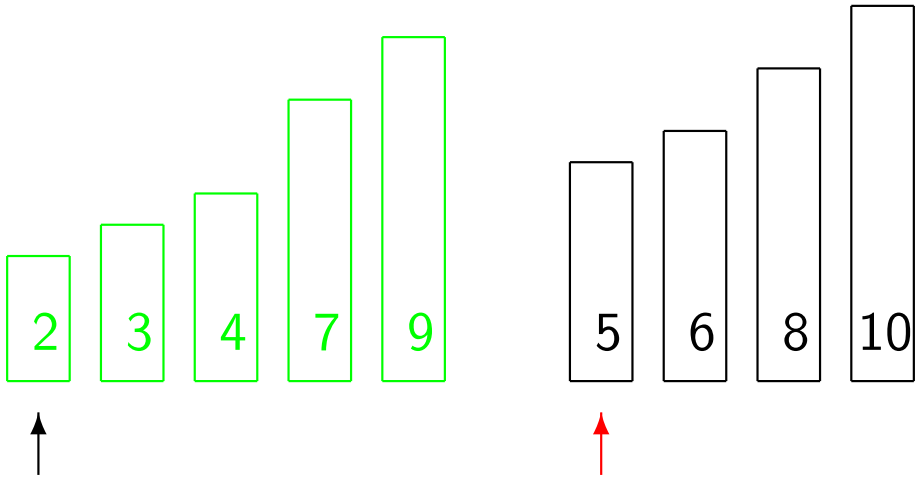
- Percorremos os dois subvetores,

Intercalando



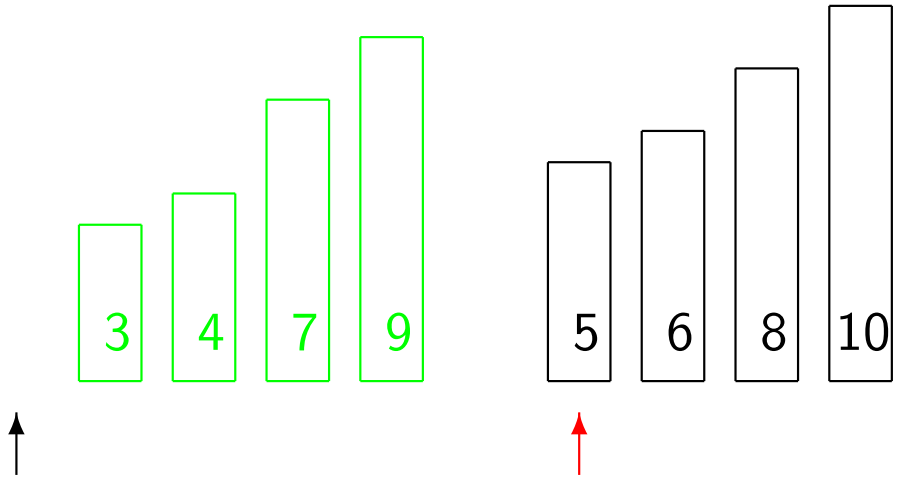
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor

Intercalando

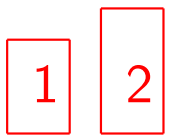


- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos

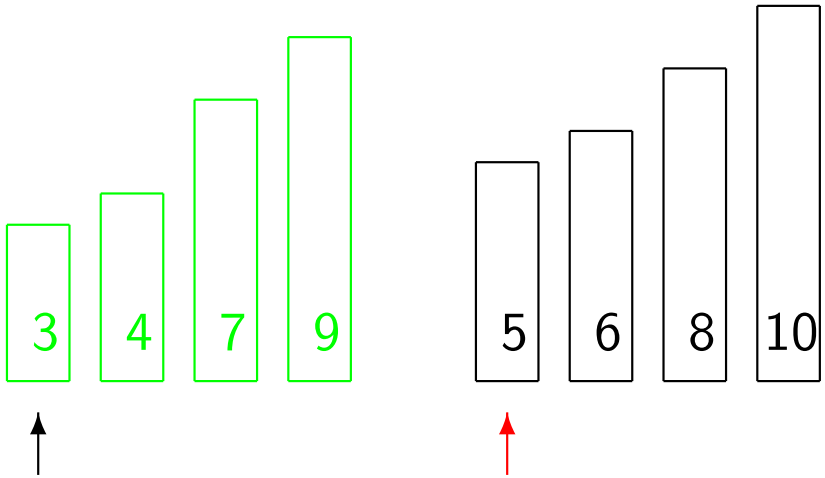
Intercalando



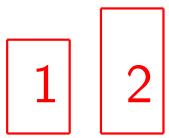
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



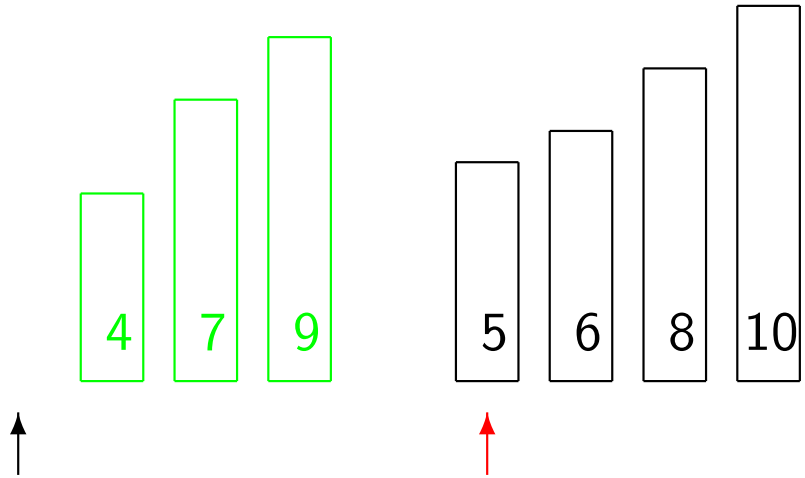
Intercalando



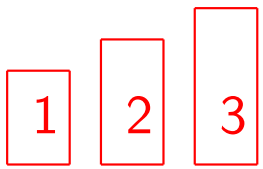
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



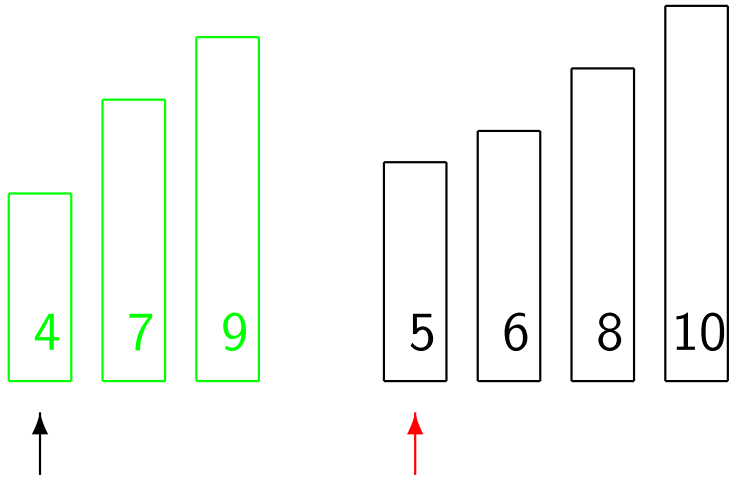
Intercalando



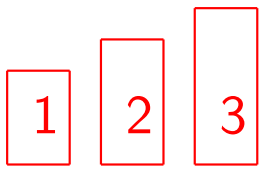
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



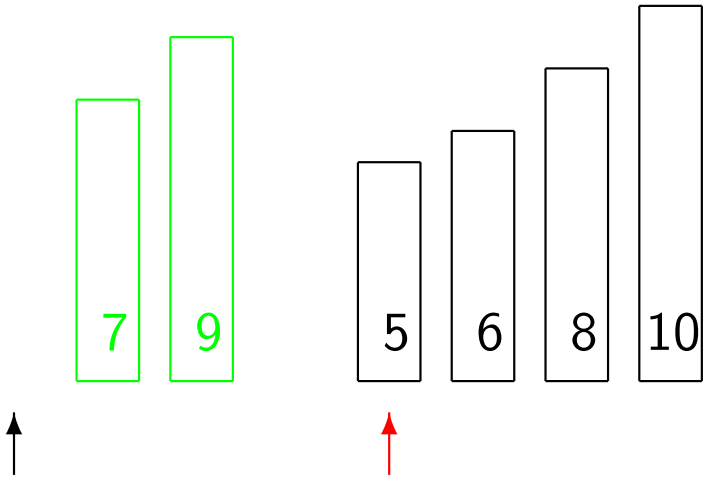
Intercalando



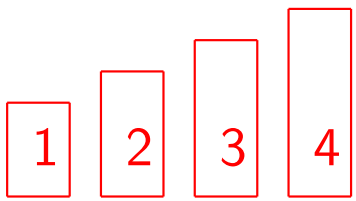
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



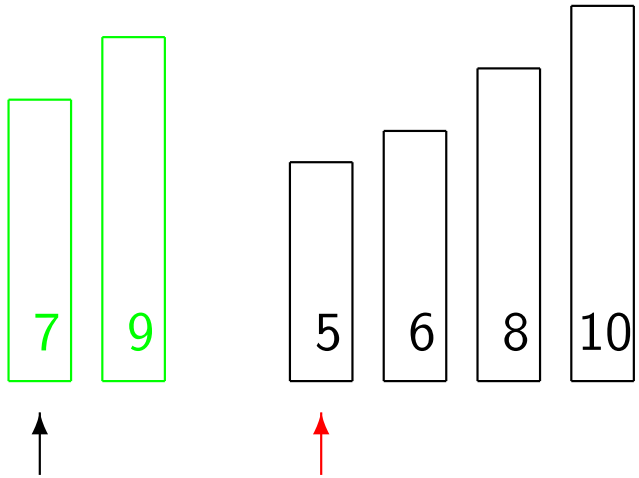
Intercalando



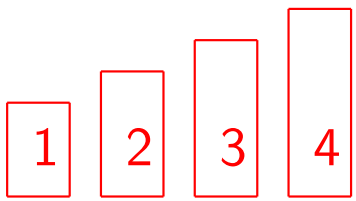
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



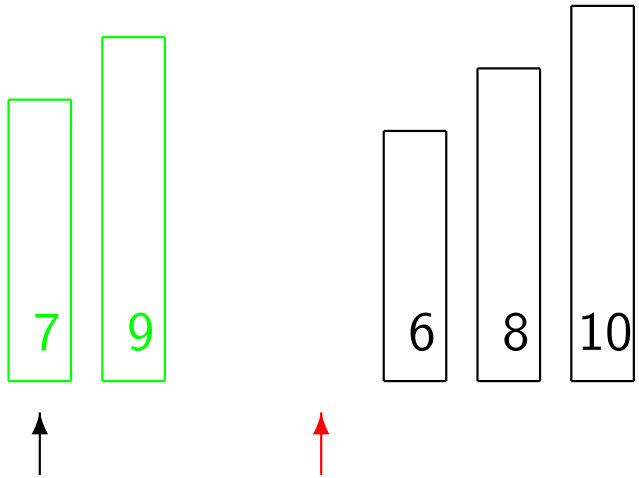
Intercalando



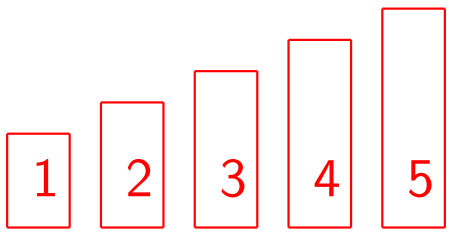
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



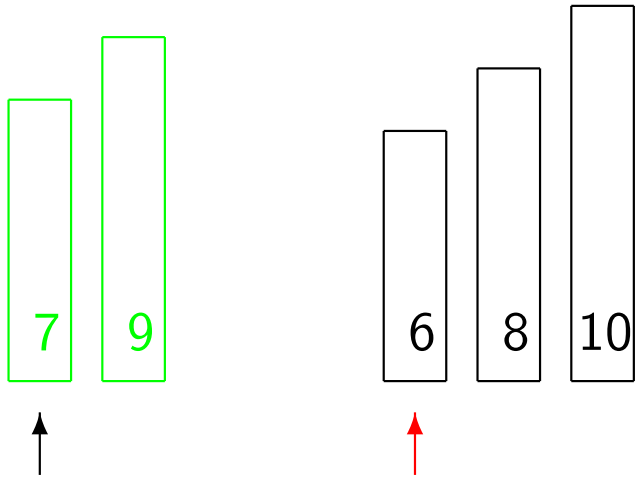
Intercalando



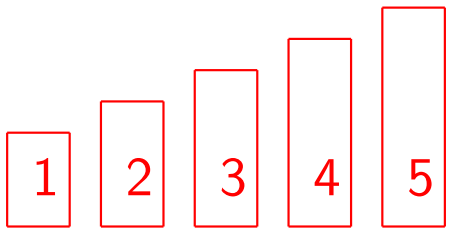
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



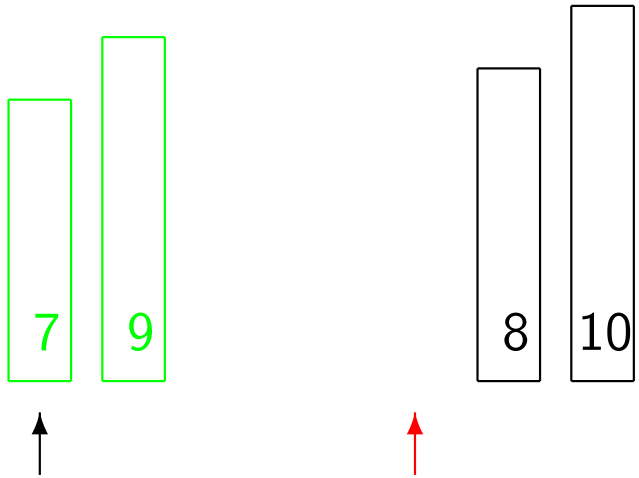
Intercalando



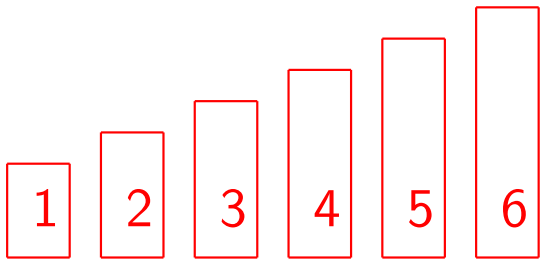
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



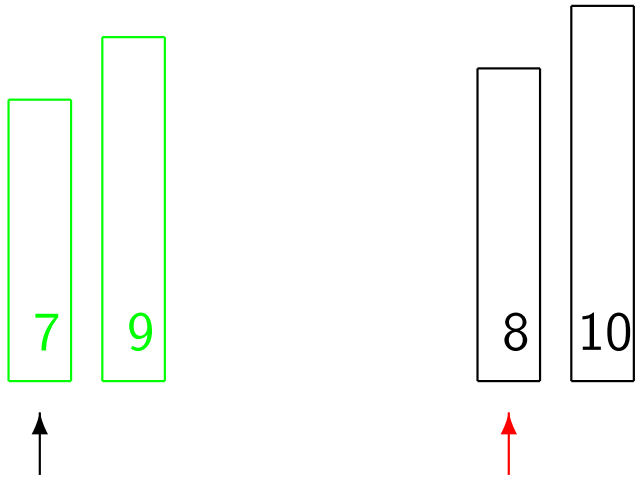
Intercalando



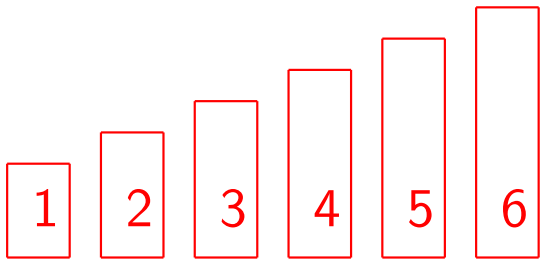
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



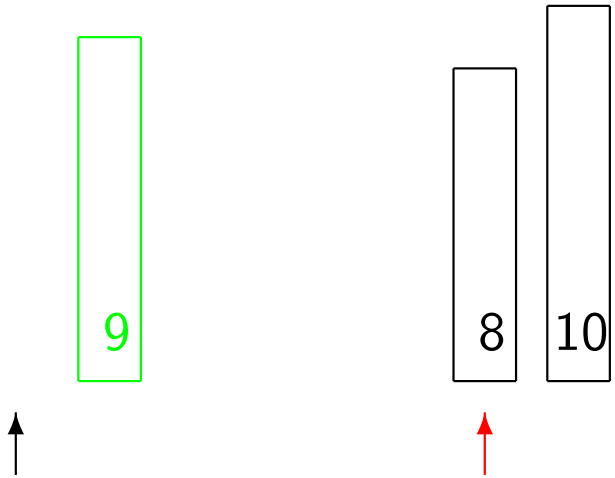
Intercalando



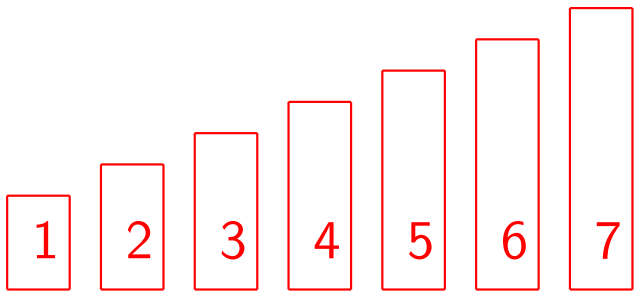
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



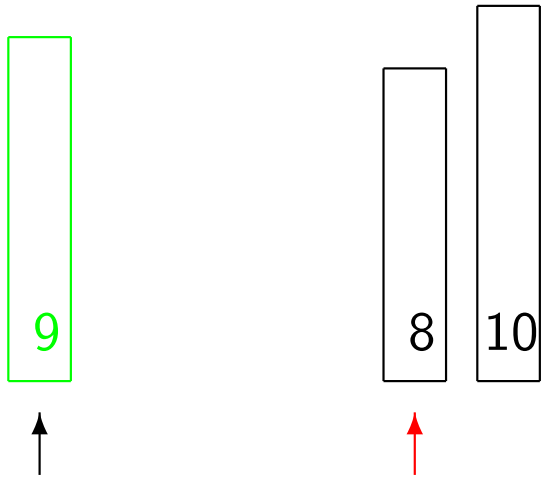
Intercalando



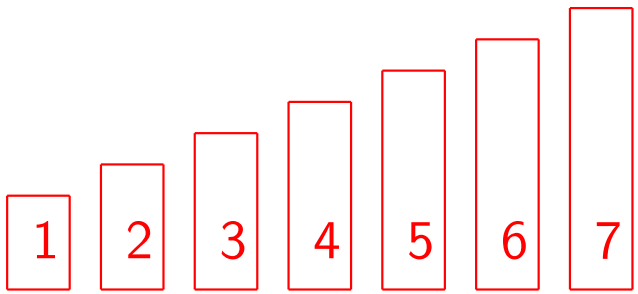
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



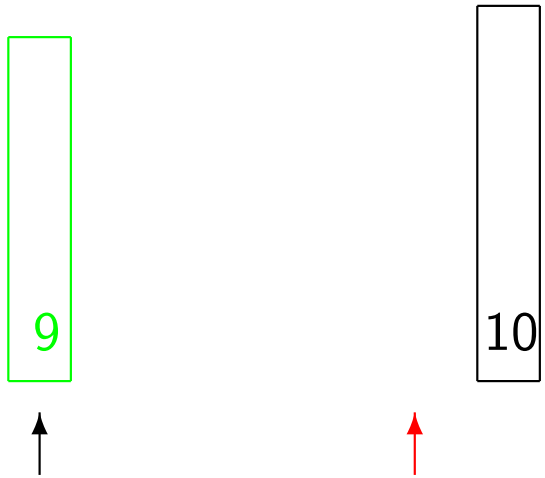
Intercalando



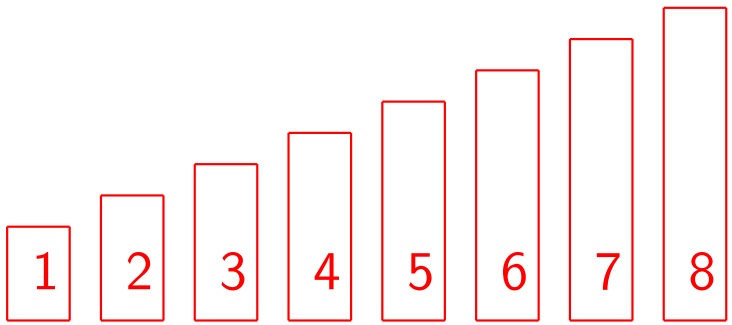
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



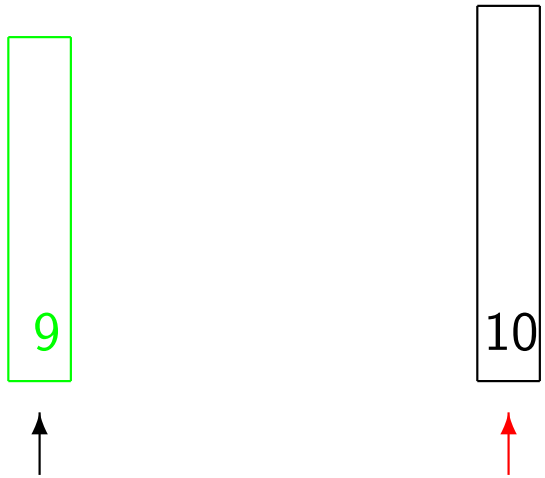
Intercalando



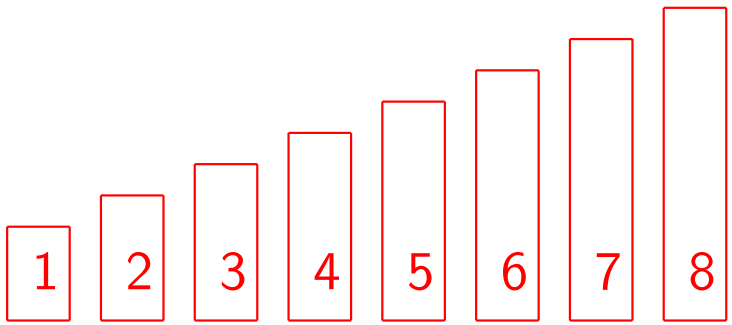
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



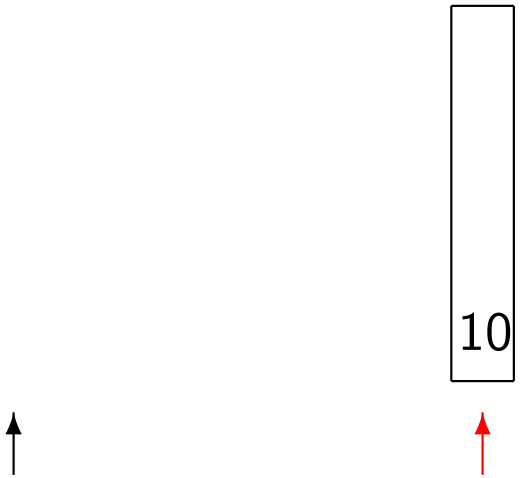
Intercalando



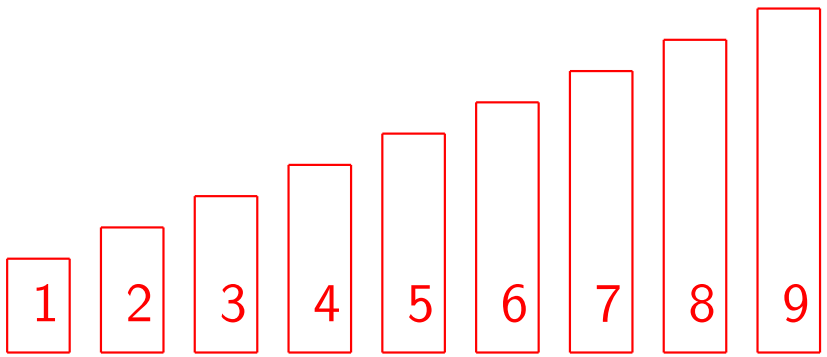
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



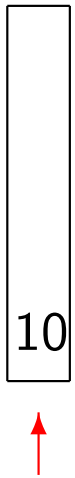
Intercalando



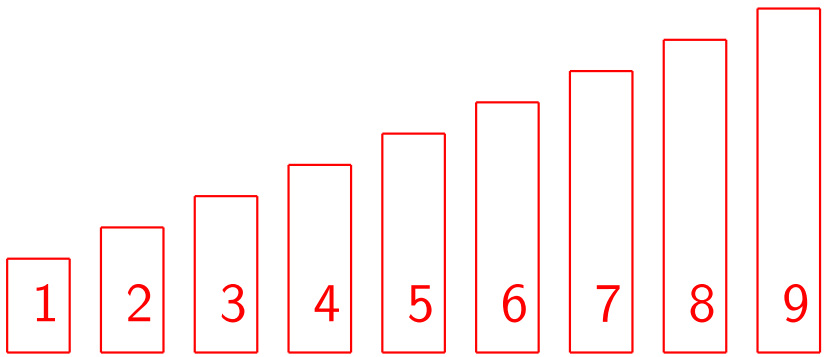
- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos



Intercalando

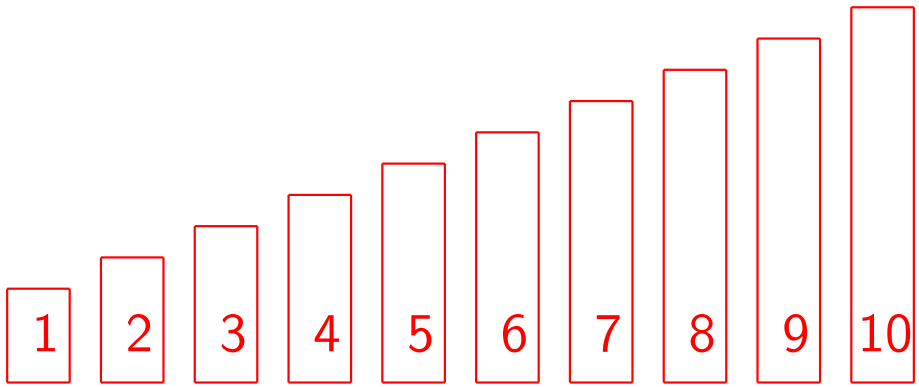


- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos
- Depois movemos o resto.



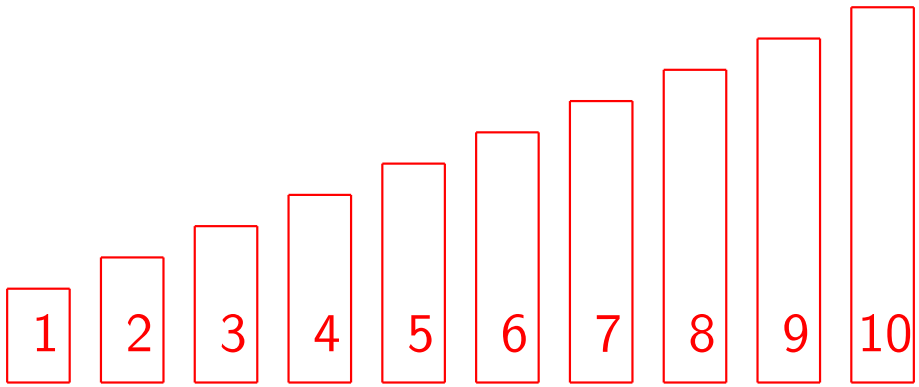
Intercalando

- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos
- Depois movemos o resto.



Intercalando

- Percorremos os dois subvetores,
- pegamos o menor e inserimos no novo vetor e continuamos
- Depois movemos o resto.



Divisão e conquista

Observação

Divisão e conquista

Observação

- A recursão parte do princípio que é mais fácil resolver problemas menores

Divisão e conquista

Observação

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista

Observação

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista

É a técnica para resolver problemas baseada em:

Divisão e conquista

Observação

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista

É a técnica para resolver problemas baseada em:

- **Divisão:** Quebramos um problema em vários subproblemas menores

Divisão e conquista

Observação

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista

É a técnica para resolver problemas baseada em:

- **Divisão:** Quebramos um problema em vários subproblemas menores
- **Conquista:** Combinamos a solução dos problemas menores

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:
  - O primeiro nas posições de `ini` até `meio`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:
  - O primeiro nas posições de `ini` até `meio`
  - O segundo nas posições de `meio + 1` até `fim`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:
  - O primeiro nas posições de `ini` até `meio`
  - O segundo nas posições de `meio + 1` até `fim`
- Precisamos de um vetor auxiliar do tamanho do vetor

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:
  - O primeiro nas posições de `ini` até `meio`
  - O segundo nas posições de `meio + 1` até `fim`
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho `MAX`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para a intercalação

- Os dois subvetores estão armazenados em `vetor`:
  - O primeiro nas posições de `ini` até `meio`
  - O segundo nas posições de `meio + 1` até `fim`
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho `MAX`
  - Exemplo `#define MAX 100`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Intercalar subvetores

```
int intercalar(int vetor[], int ini, int meio, int fim) {
    int auxiliar[MAX];                  // vetor auxiliar
    int i = ini, j = meio + 1, k = 0; // índices dos vetores

    // intercala
    while(i <= meio && j <= fim) {
        if (vetor[i] <= vetor[j])
            auxiliar[k++] = vetor[i++];
        else
            auxiliar[k++] = vetor[j++];
    }
    // copia resto de cada subvetor
    while (i <= meio) auxiliar[k++] = vetor[i++];
    while (j <= fim)  auxiliar[k++] = vetor[j++];

    // copia de auxiliar para vetor
    for (i = ini, k=0; i <= fim; i++, k++)
        vetor[i] = auxiliar[k];
}
```

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`
  - O vetor termina na posição `vetor[fim]`

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`
  - O vetor termina na posição `vetor[fim]`
- Dividimos o vetor em dois subvetores de tamanho $\frac{n}{2}$.

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`
  - O vetor termina na posição `vetor[fim]`
- Dividimos o vetor em dois subvetores de tamanho $\frac{n}{2}$.
- O caso base é

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`
  - O vetor termina na posição `vetor[fim]`
- Dividimos o vetor em dois subvetores de tamanho $\frac{n}{2}$.
- O caso base é um vetor de tamanho 0 ou 1.

Algoritmo de ordenação por intercalação (*Merge-Sort*)

Convenções para ordenação

- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[ini]`
  - O vetor termina na posição `vetor[fim]`
- Dividimos o vetor em dois subvetores de tamanho $\frac{n}{2}$.
- O caso base é um vetor de tamanho 0 ou 1.

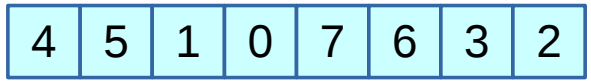
Ordenação por intercalação

```
void ordenar_intercalacao(int vetor[], int ini, int fim) {
    int meio;

    if (ini < fim) {
        meio = (ini + fim) / 2;
        ordenar_intercalacao(vetor, ini, meio);
        ordenar_intercalacao(vetor, meio + 1, fim);
        intercalar(vetor, ini, meio, fim);
    }
}
```

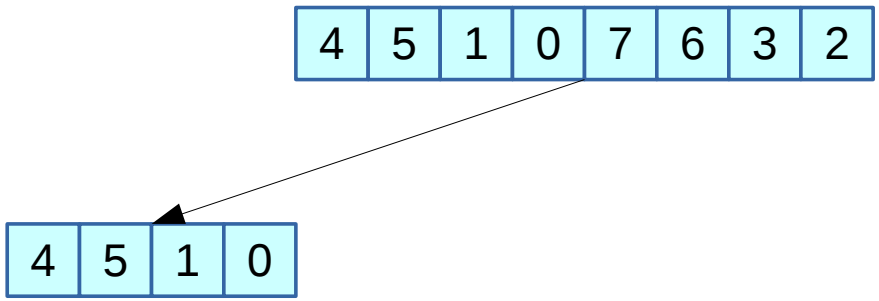
Ordenação por intercalação - Exemplo

```
#include "stdio.h"

#define MAX 100

void intercalar(int vetor[], int ini, int meio, int fim);
void ordenar_intercalacao(int vetor[], int ini, int fim);

int main() {
    int i;
    int vetor[] = { 4, 5, 1, 0, 7, 6, 3, 2 };

    ordenar_intercalacao(vetor, 0, 7);

    for (i = 0; i < 8; i++)
        printf("%d\n", vetor[i]);
}
```

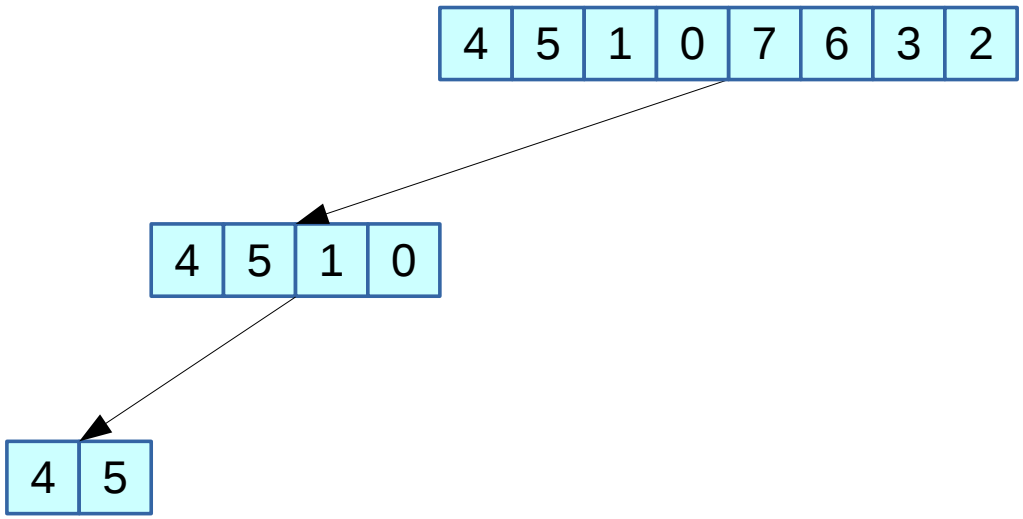
Ordenação por intercalação - simulando



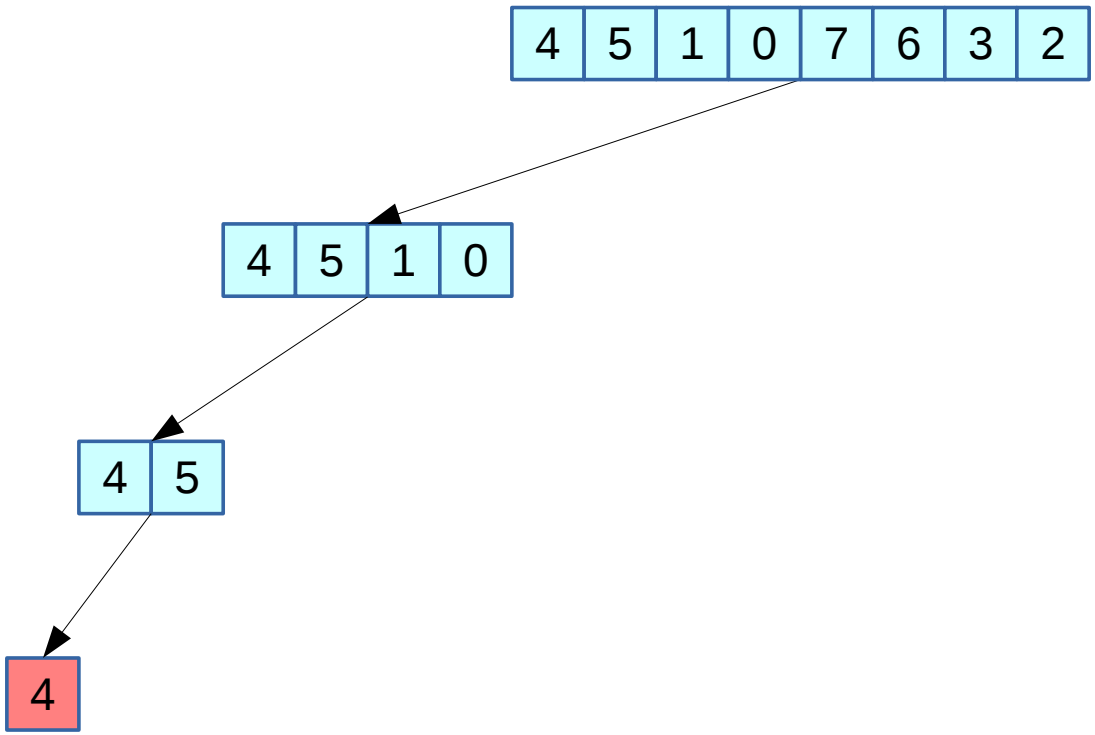
Ordenação por intercalação - simulando



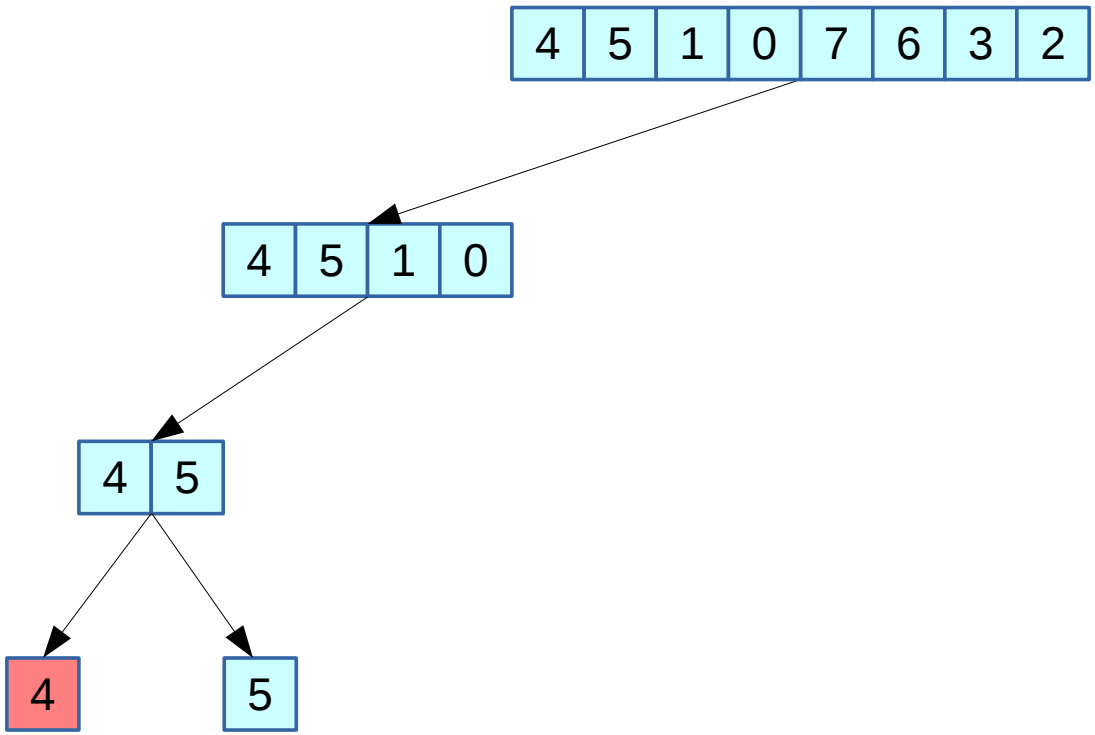
Ordenação por intercalação - simulando



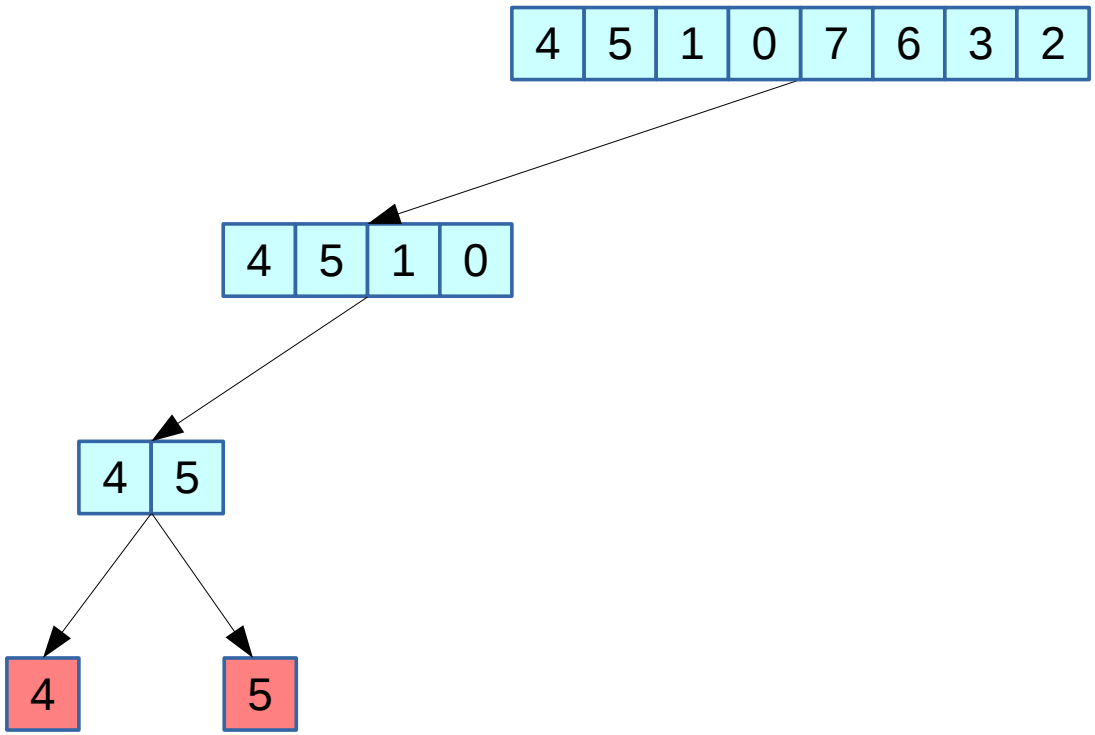
Ordenação por intercalação - simulando



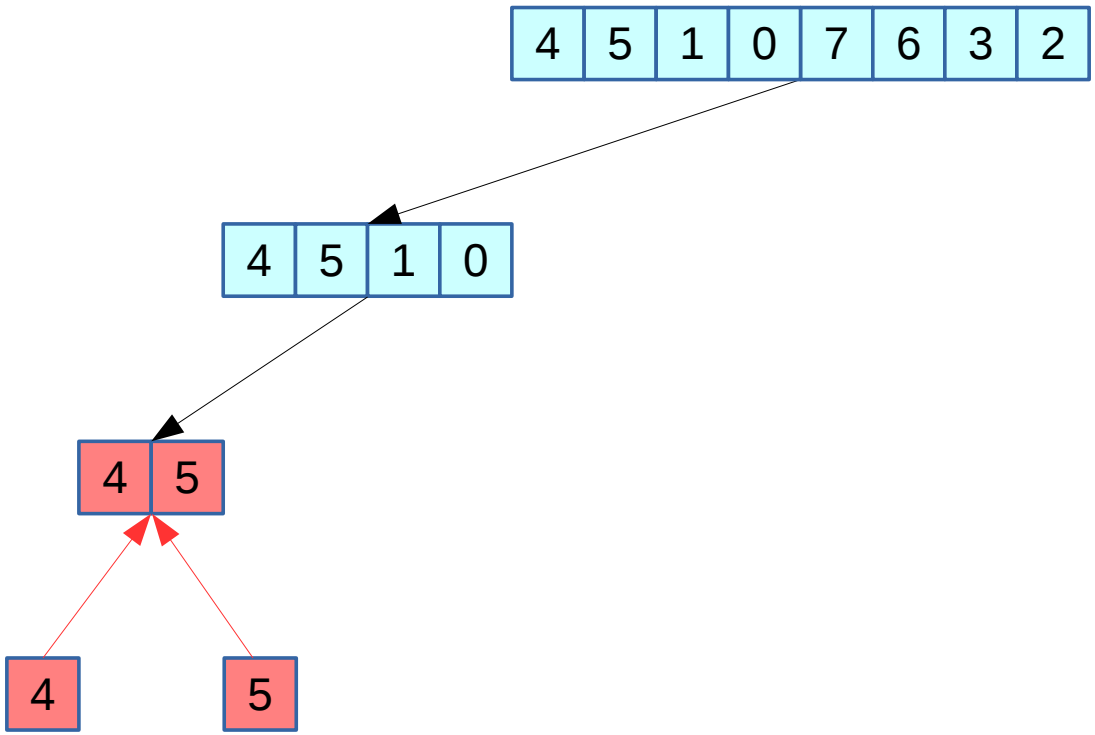
Ordenação por intercalação - simulando



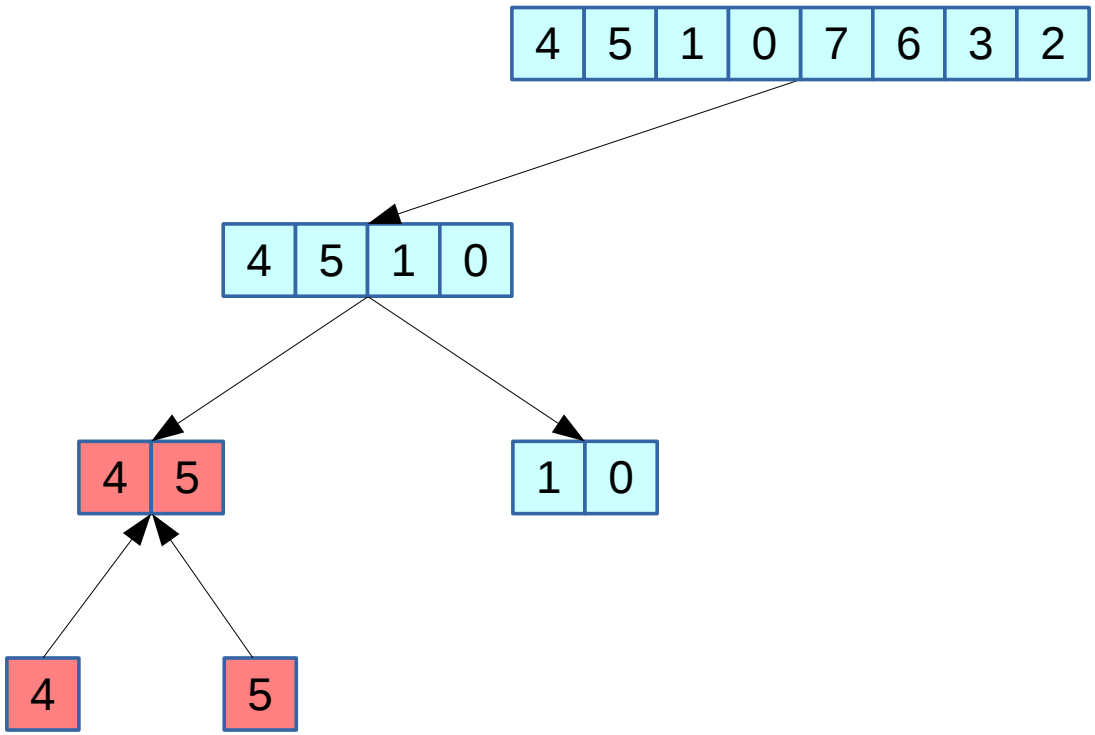
Ordenação por intercalação - simulando



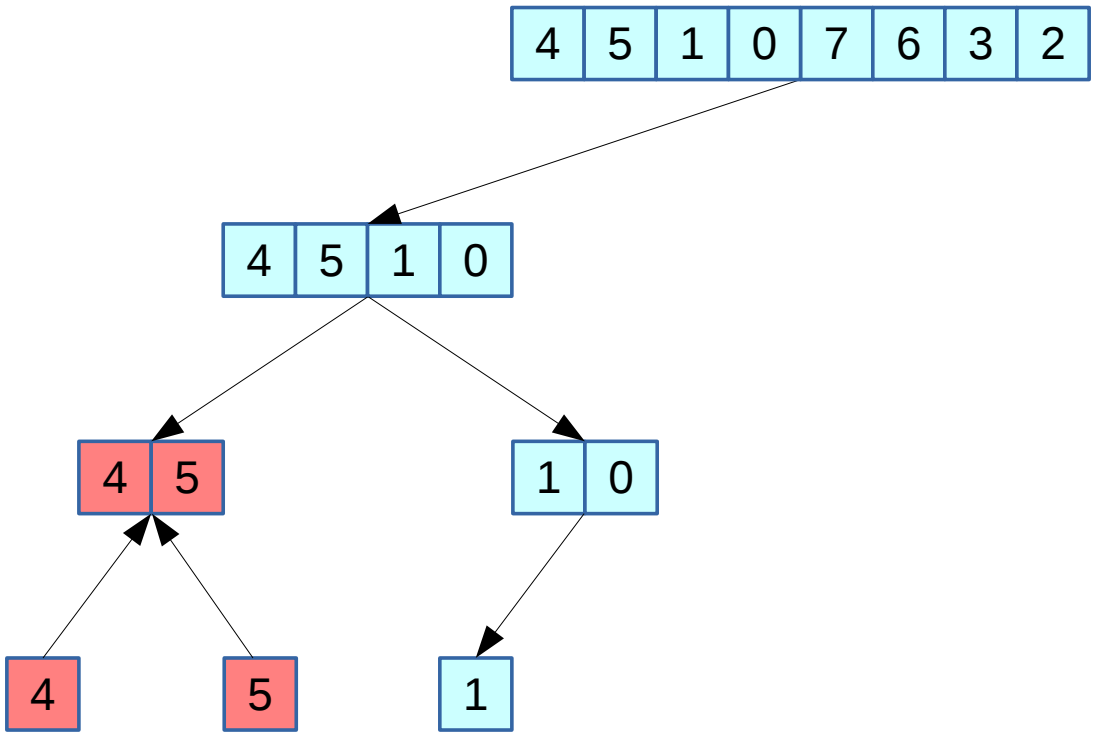
Ordenação por intercalação - simulando



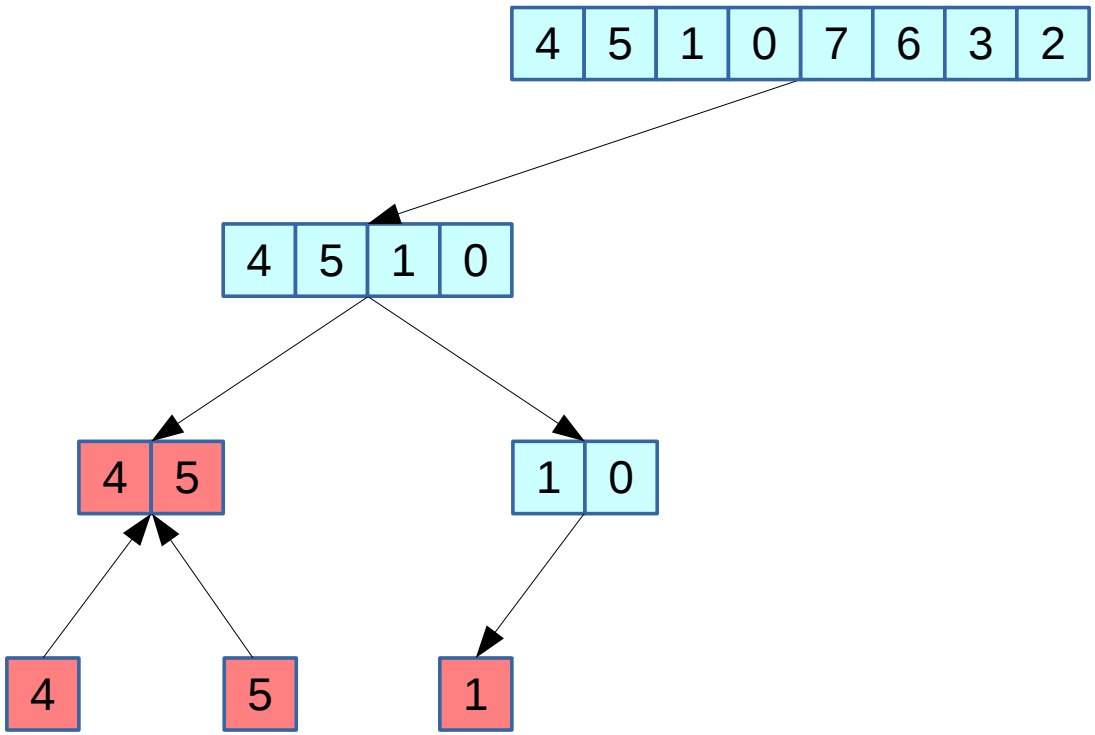
Ordenação por intercalação - simulando



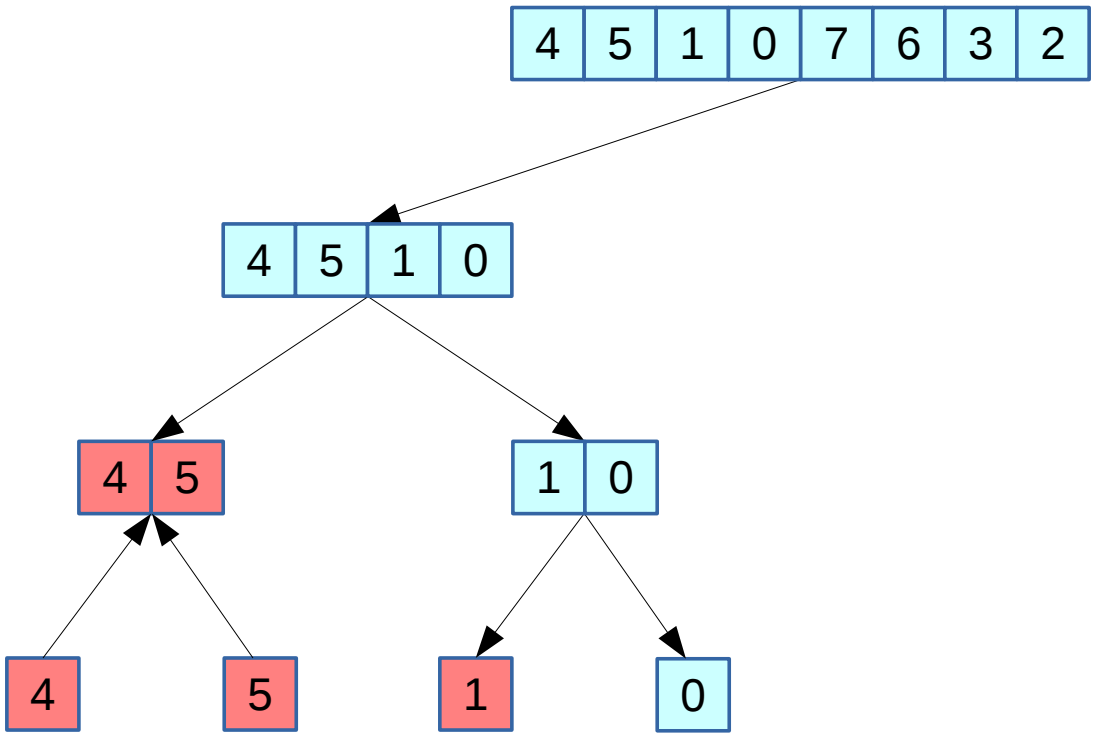
Ordenação por intercalação - simulando



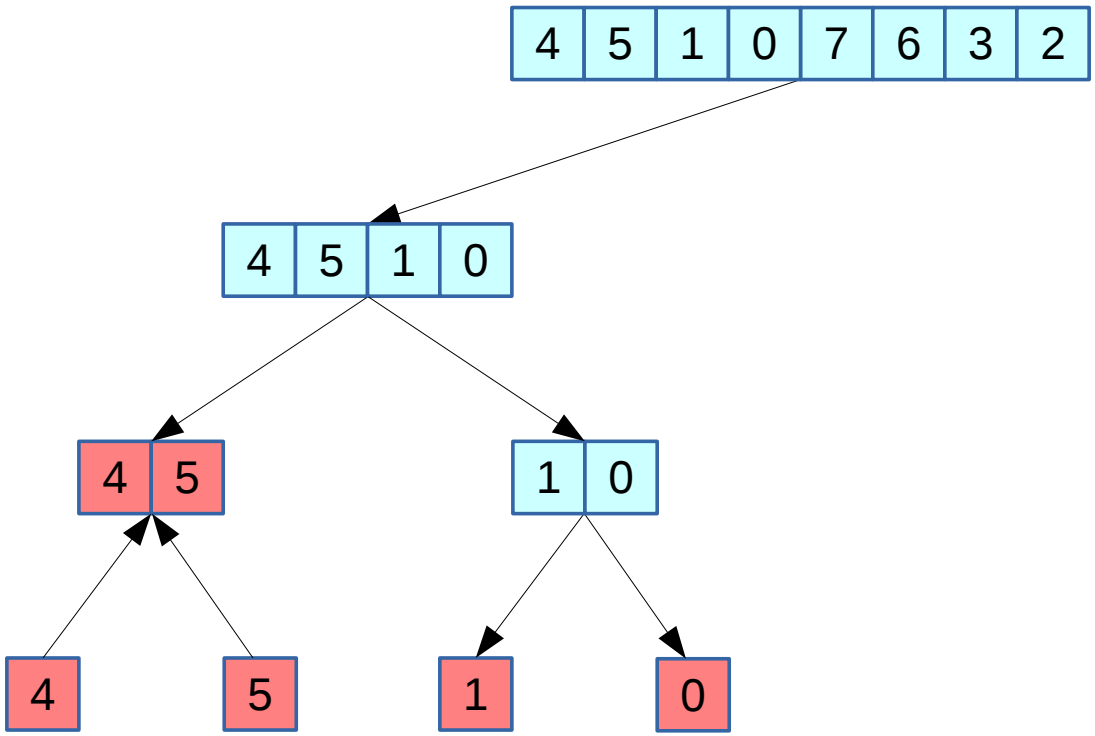
Ordenação por intercalação - simulando



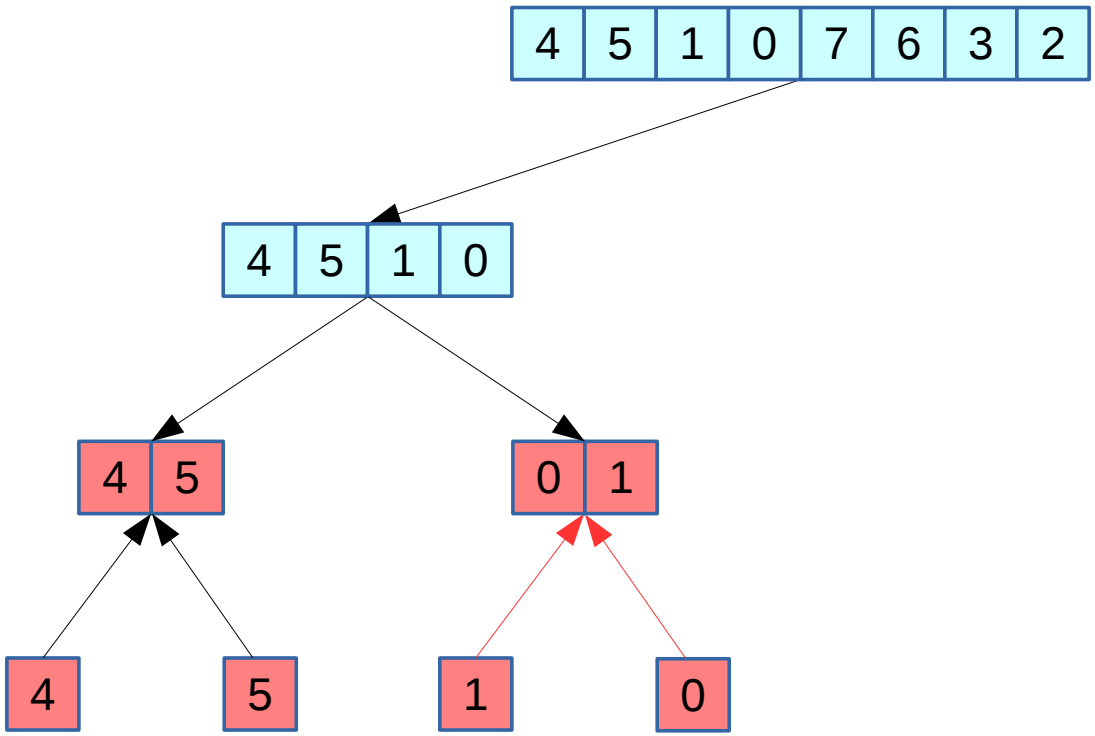
Ordenação por intercalação - simulando



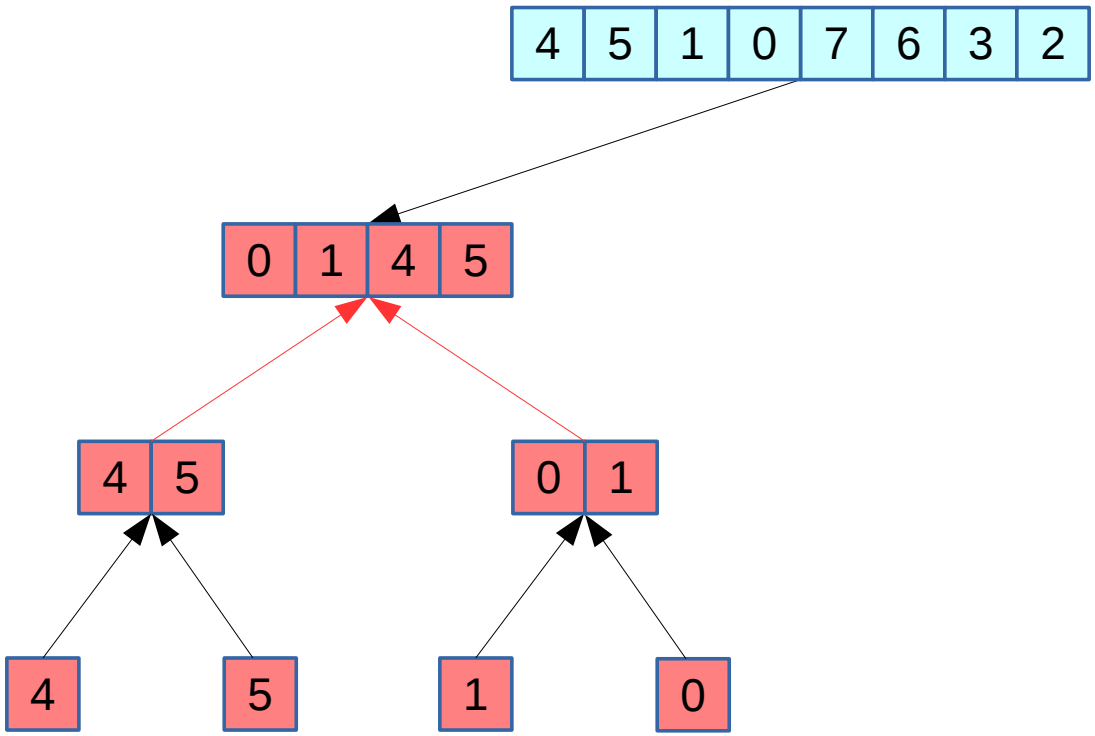
Ordenação por intercalação - simulando



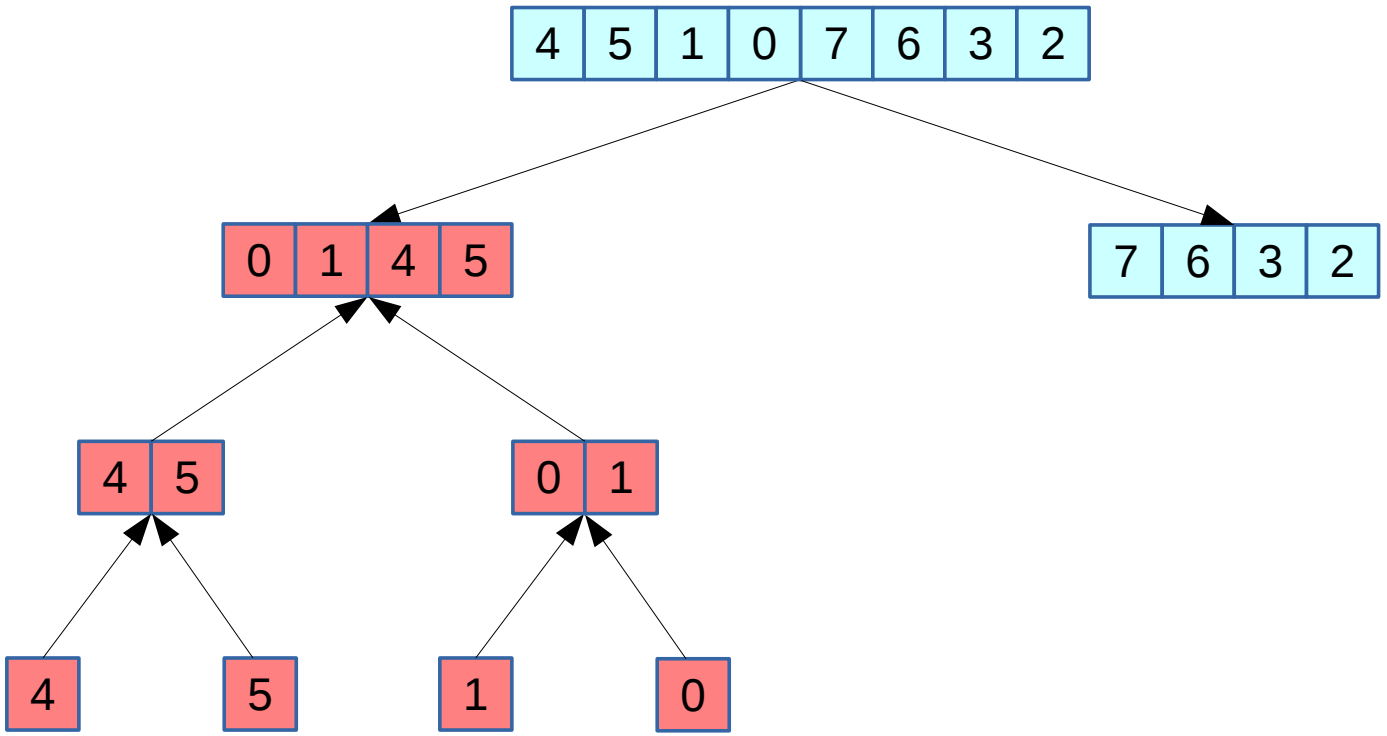
Ordenação por intercalação - simulando



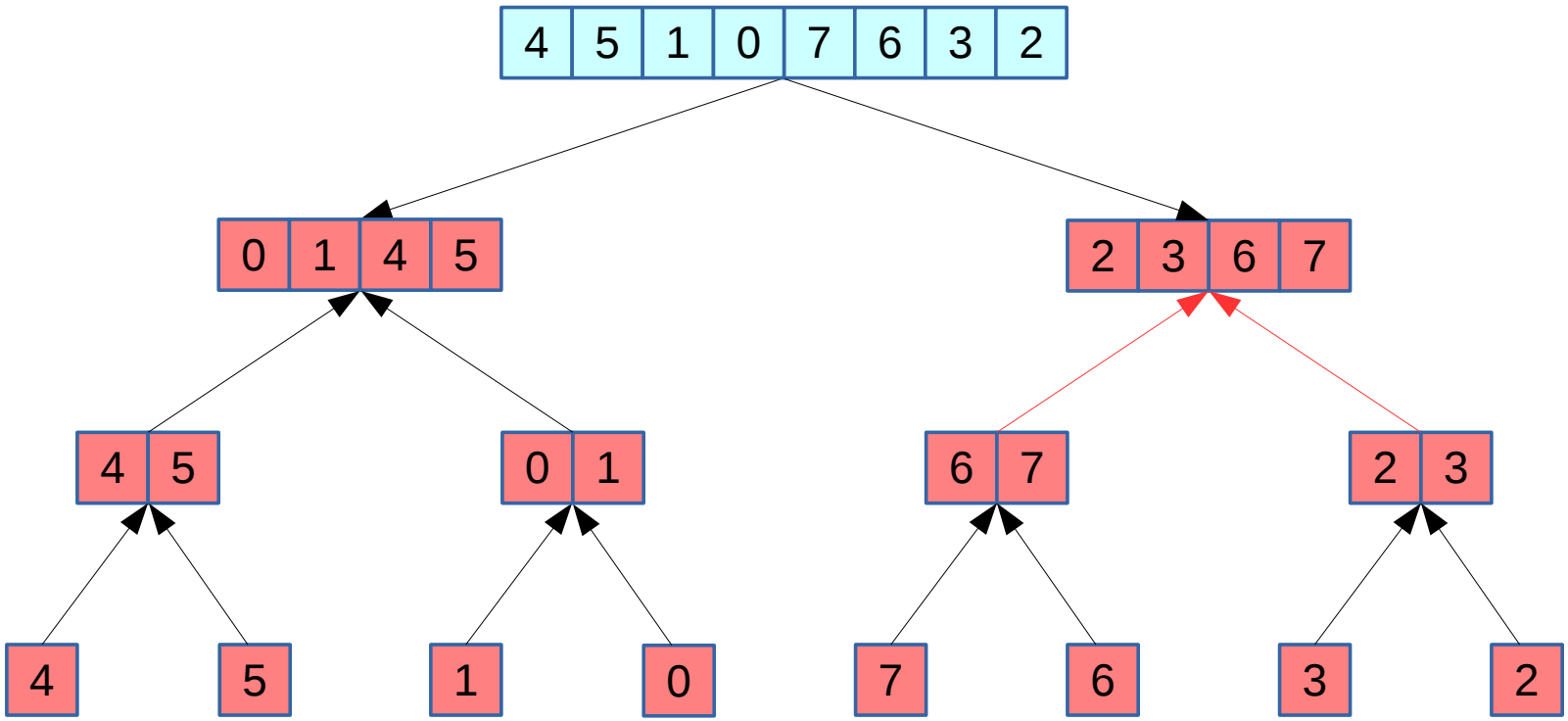
Ordenação por intercalação - simulando



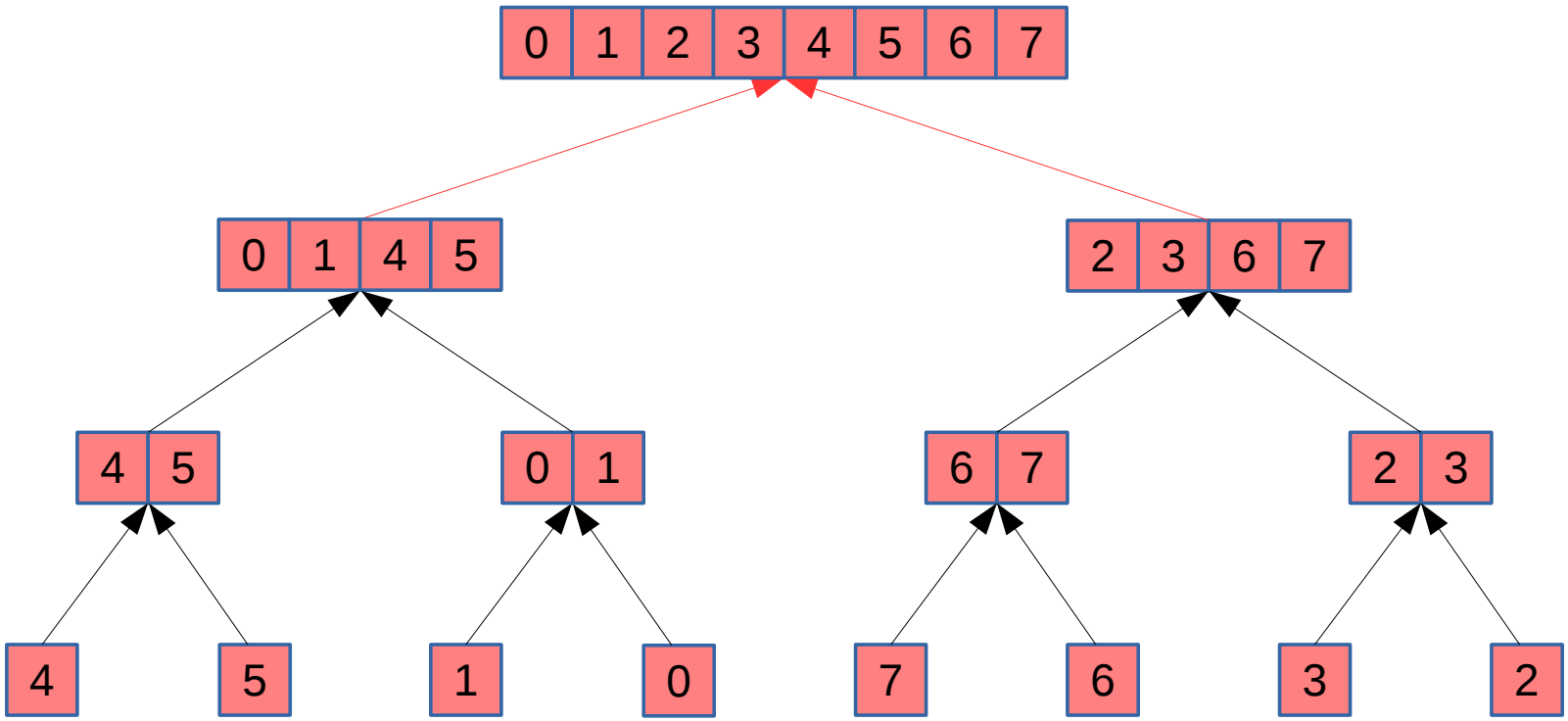
Ordenação por intercalação - simulando



Ordenação por intercalação - simulando



Ordenação por intercalação - simulando



Eficiência da ordenação por intercalação

Quantas comparações?

Eficiência da ordenação por intercalação

Quantas comparações?

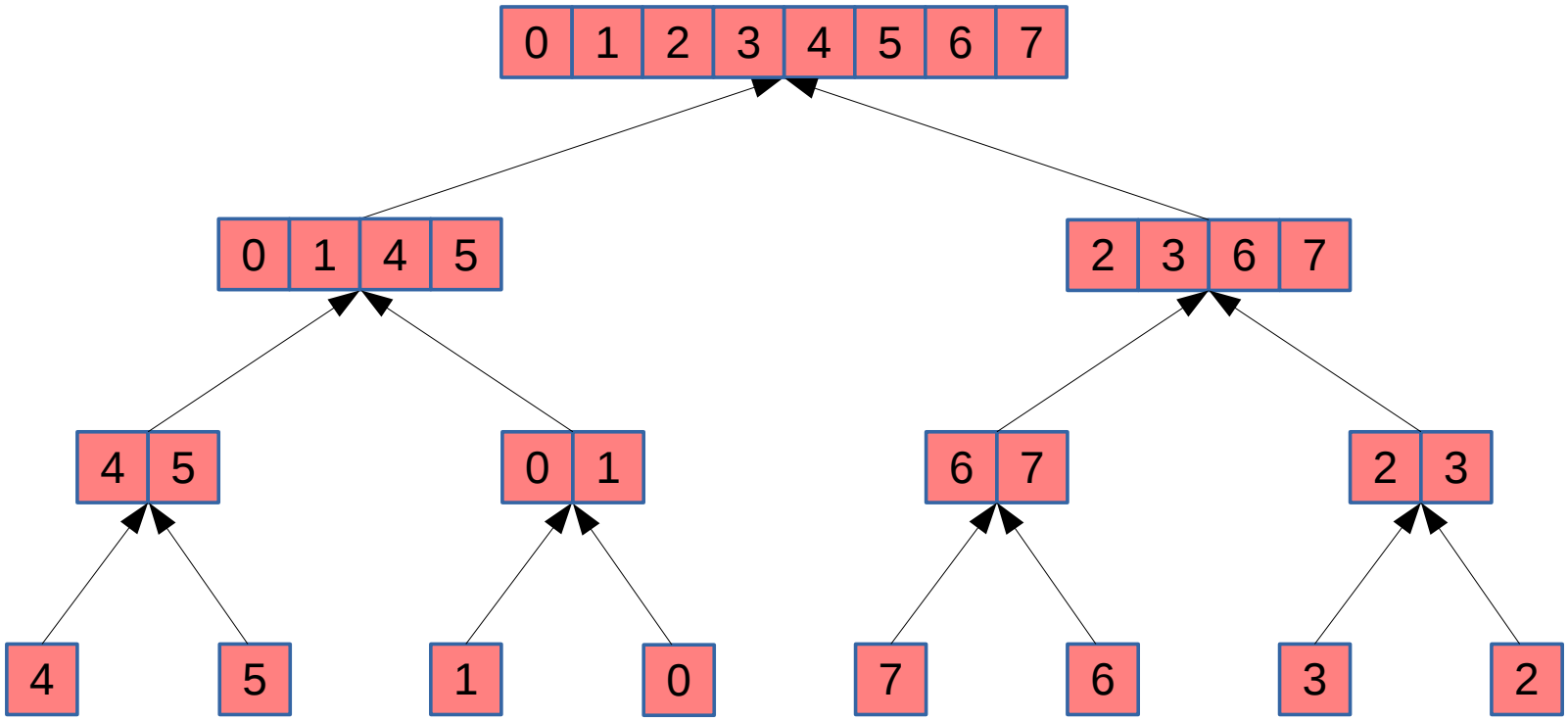
- somente `intercalar` compara elementos

Eficiência da ordenação por intercalação

Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar

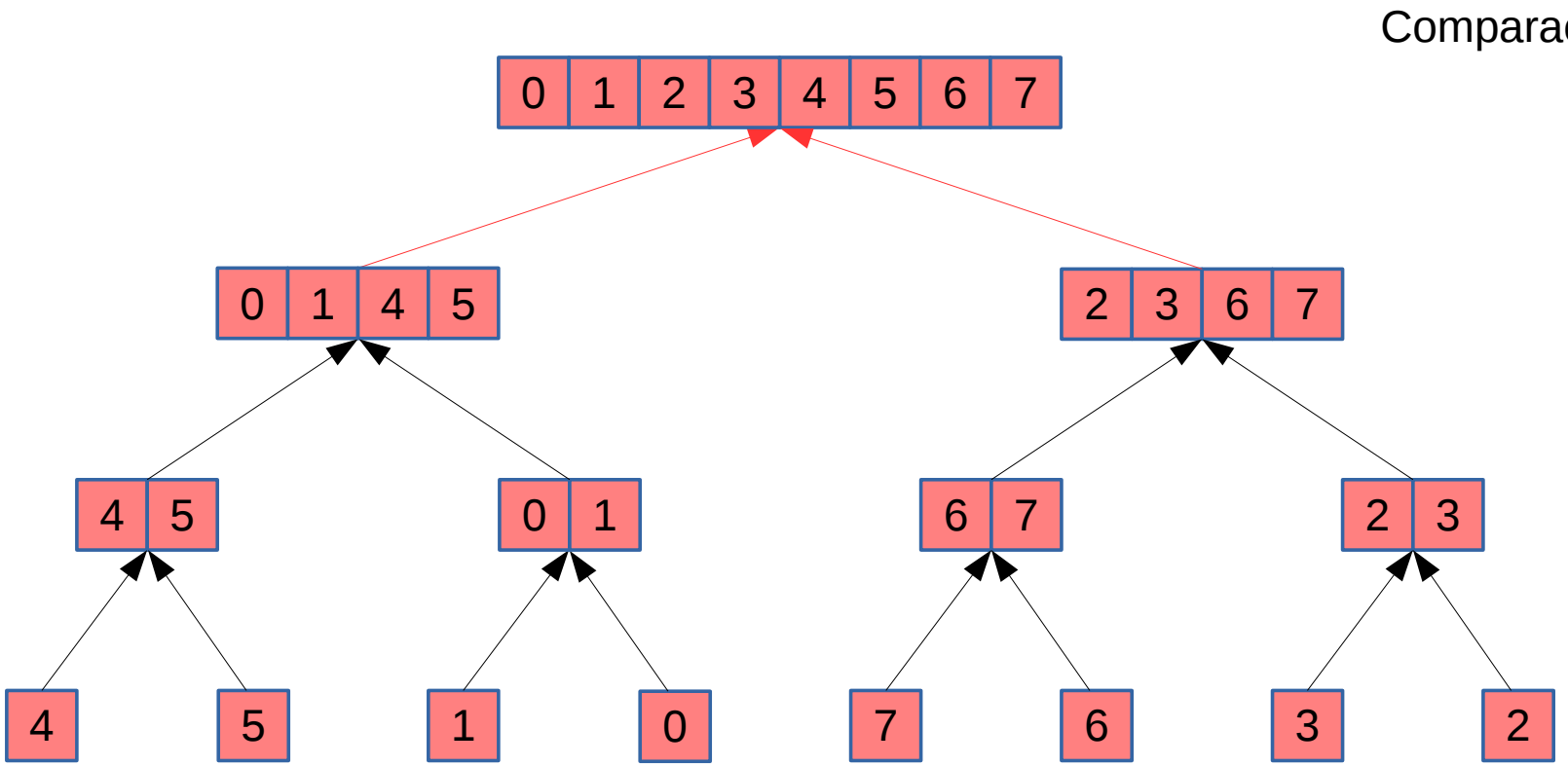
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

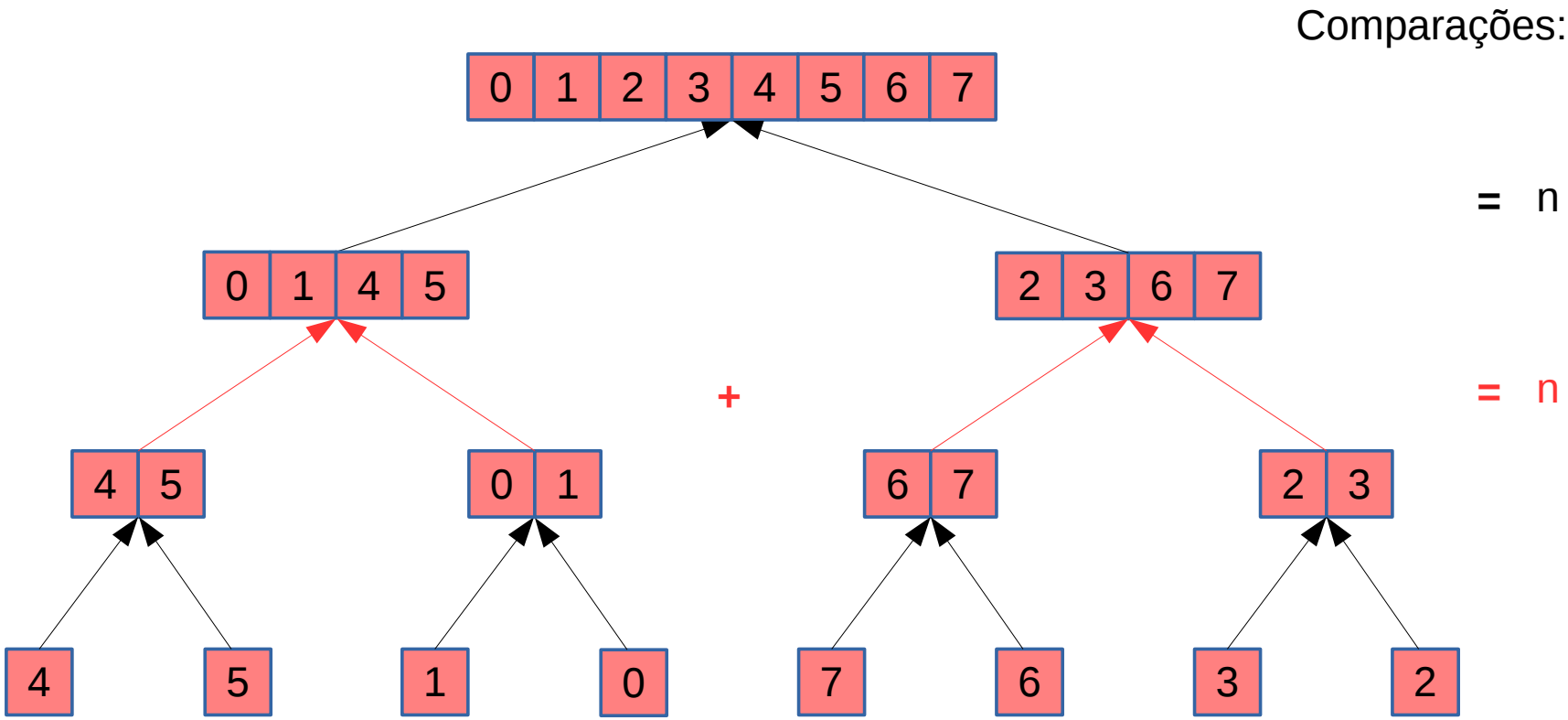
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

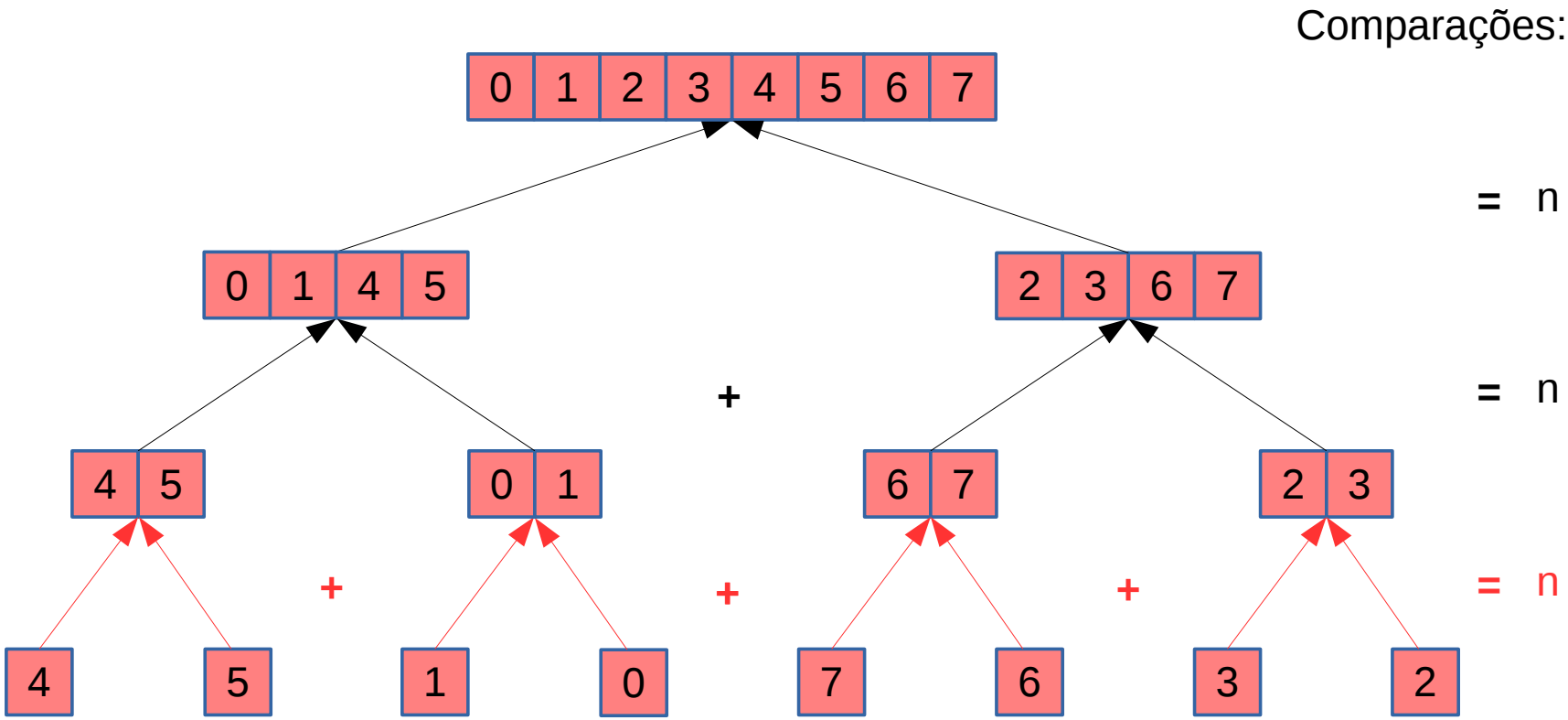
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

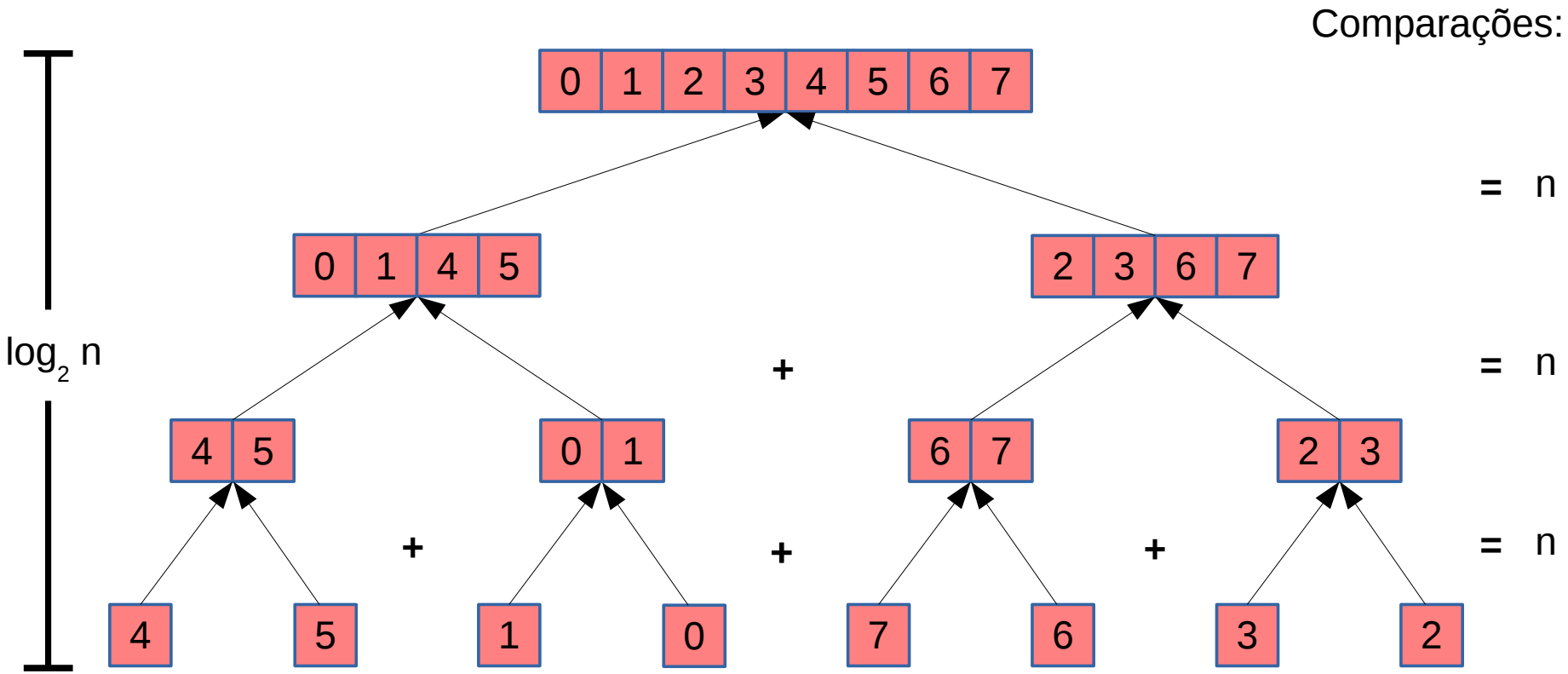
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

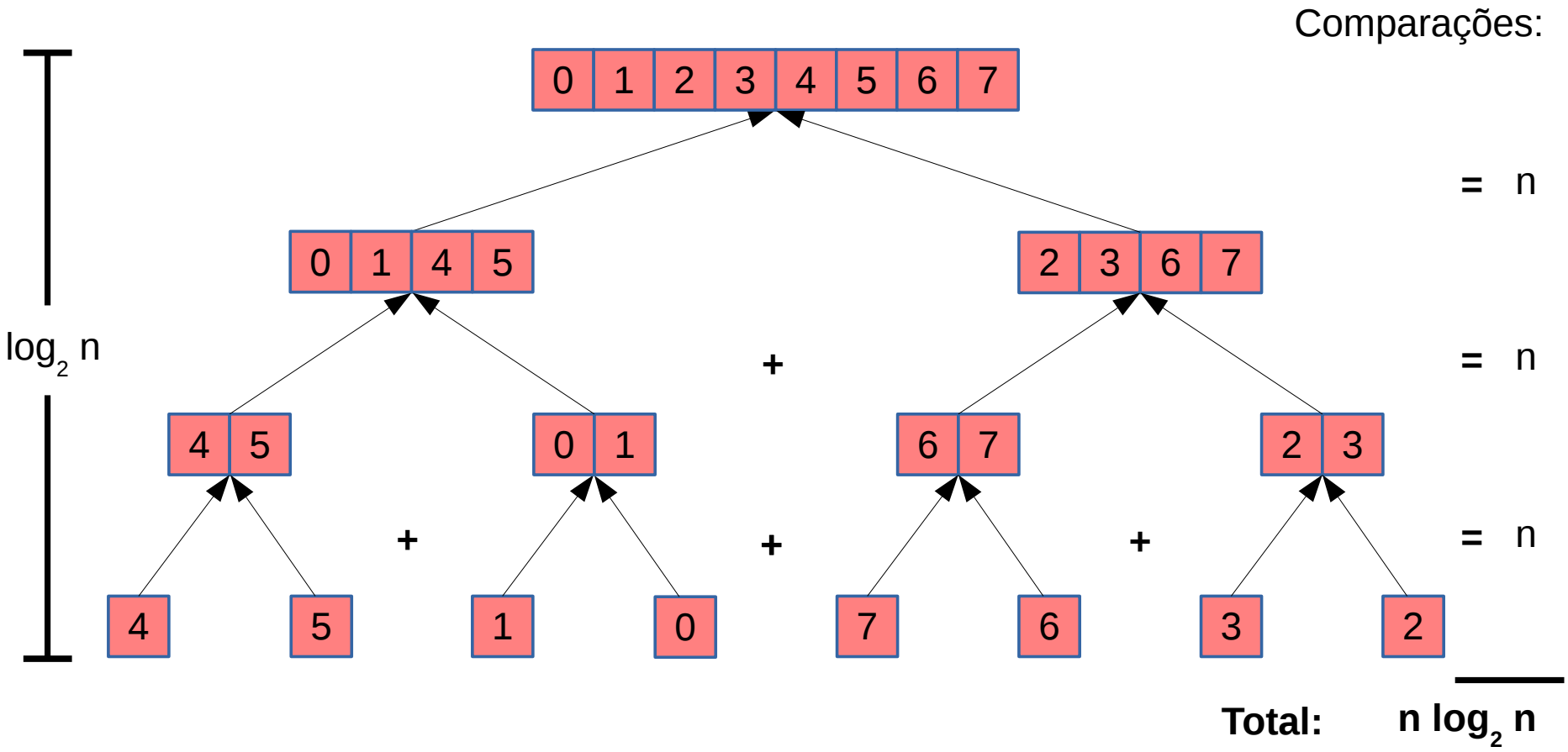
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

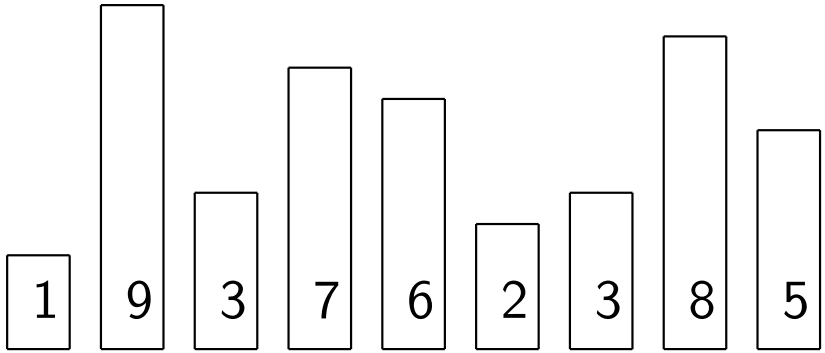
Eficiência da ordenação por intercalação



Quantas comparações?

- somente `intercalar` compara elementos
- vamos somar para todas as chamadas de intercalar (com ajuda da árvore)

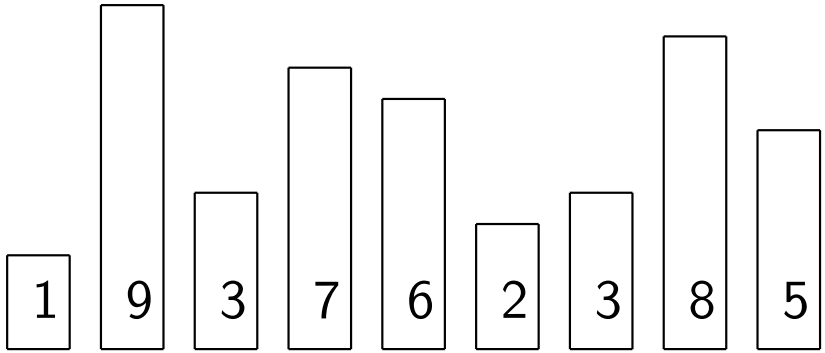
Outra estratégia para ordenação



Problema 1

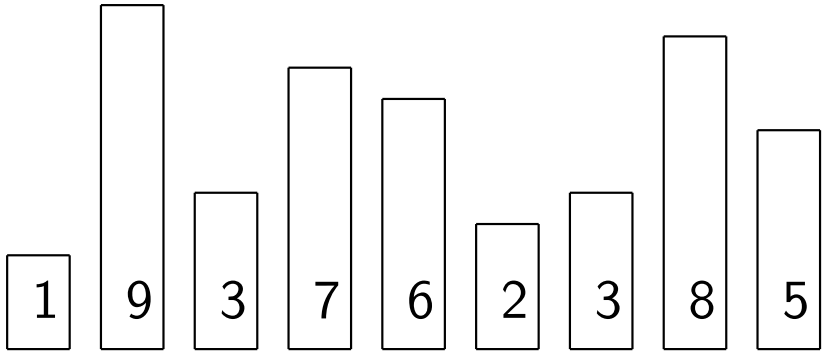
Suponha que temos um vetor desordenado com 10 números. Como fazer com que números *pequenos* (menores que 5) fiquem antes dos números *grandes* (maiores que 5)?

Considere a função

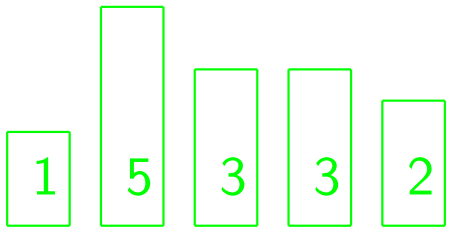


- `int particionar(int vetor[], int ini, int fim)`

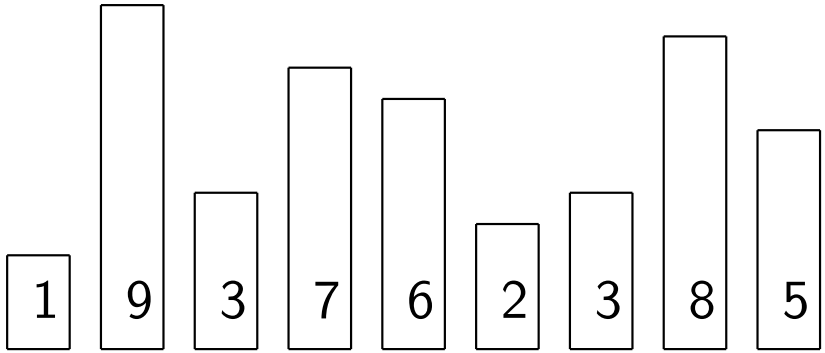
Considere a função



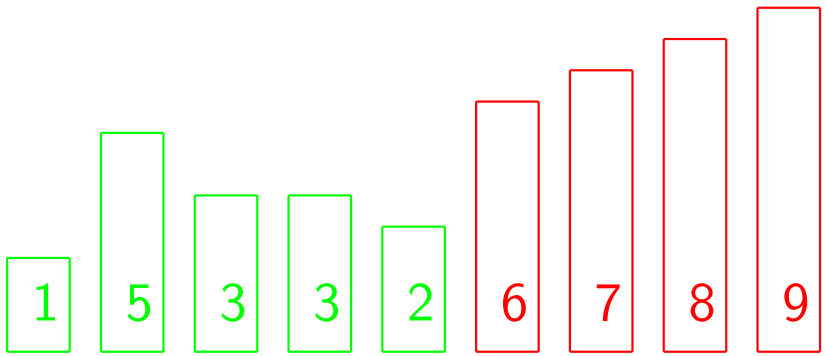
- `int particionar(int vetor[], int ini, int fim)`
  - a primeira parte do vetor contém elementos **"pequenos"**



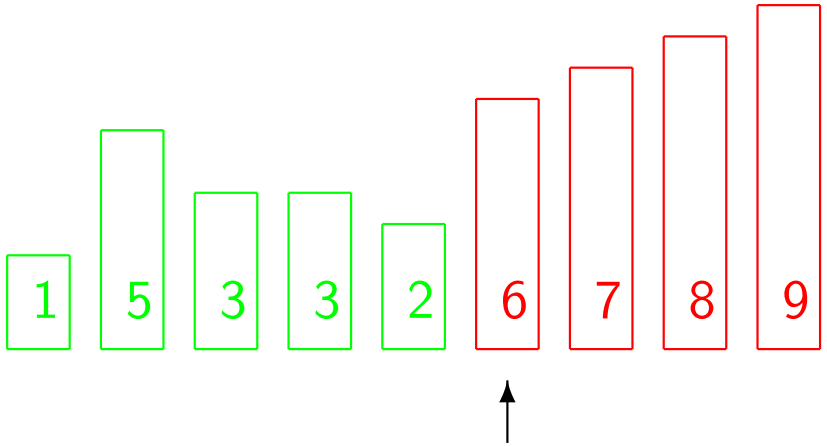
Considere a função



- `int particionar(int vetor[], int ini, int fim)`
  - a primeira parte do vetor contém elementos **"pequenos"**
  - a segunda parte do vetor contém elementos **"grandes"**



Combinando



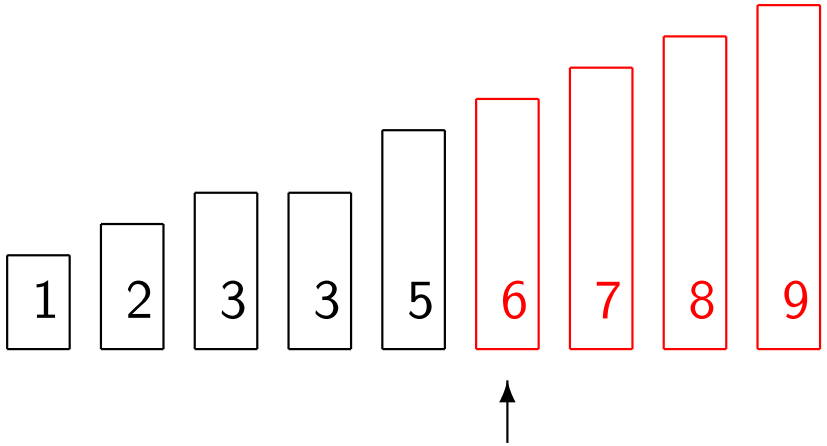
Problema 2

Suponha que o subvetor

- da posição **pos** a **fim**: contenha apenas elementos grandes
- da posição **ini** a **pos - 1**: contenha apenas elementos pequenos

Como ordenar?

Combinando



Problema 2

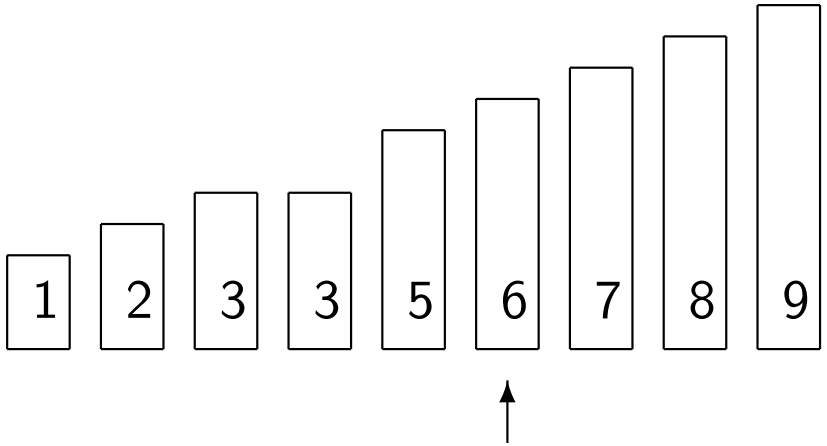
Suponha que o subvetor

- da posição **pos** a **fim**: contenha apenas elementos grandes
- da posição **ini** a **pos - 1**: contenha apenas elementos pequenos

Como ordenar?

- Ordenamos recursivamente o primeiro subvetor

Combinando



Problema 2

Suponha que o subvetor

- da posição **pos** a **fim**: contenha apenas elementos grandes
- da posição **ini** a **pos - 1**: contenha apenas elementos pequenos

Como ordenar?

- Ordenamos recursivamente o primeiro subvetor
- Depois o segundo subvetor

Divisão e conquista novamente

Quick Sort

Divisão e conquista novamente

Quick Sort

- **Divisão:** Separamos elementos pequenos e grandes

Divisão e conquista novamente

Quick Sort

- **Divisão:** Separamos elementos pequenos e grandes
- **Conquista:** Ordenamos cada subvetor

Divisão e conquista novamente

Quick Sort

- **Divisão:** Separamos elementos pequenos e grandes
- **Conquista:** Ordenamos cada subvetor

QuickSort

```
void quick_sort(int vetor[], int ini, int fim) {
    int pos;

    if (ini < fim){
        pos = particionar(vetor, ini, fim);

        quick_sort(vetor, ini, pos - 1);
        quick_sort(vetor, pos, fim);
    }
}
```

Como particionar um vetor?

Ideia

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô
- Separamos o vetor em duas partes:
  1. **primeira**: apenas elementos menores ou iguais ao pivô
  2. **segunda**: apenas elementos maiores que o pivô

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô
- Separamos o vetor em duas partes:
  1. **primeira**: apenas elementos menores ou iguais ao pivô
  2. **segunda**: apenas elementos maiores que o pivô

Algoritmo

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô
- Separamos o vetor em duas partes:
  1. **primeira**: apenas elementos menores ou iguais ao pivô
  2. **segunda**: apenas elementos maiores que o pivô

Algoritmo

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

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô
- Separamos o vetor em duas partes:
  1. **primeira**: apenas elementos menores ou iguais ao pivô
  2. **segunda**: apenas elementos maiores que o pivô

Algoritmo

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

Como particionar um vetor?

Ideia

- Escolhemos um valor pivô
- Separamos o vetor em duas partes:
  1. **primeira**: apenas elementos menores ou iguais ao pivô
  2. **segunda**: apenas elementos maiores que o pivô

Algoritmo

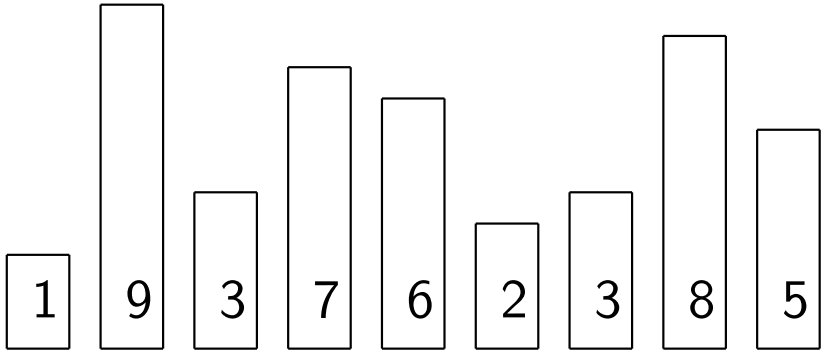
1. Obtemos o valor do pivô:
   - escolhemos sempre o valor do último elemento
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas

Algoritmo de particionamento

Particionar vetor

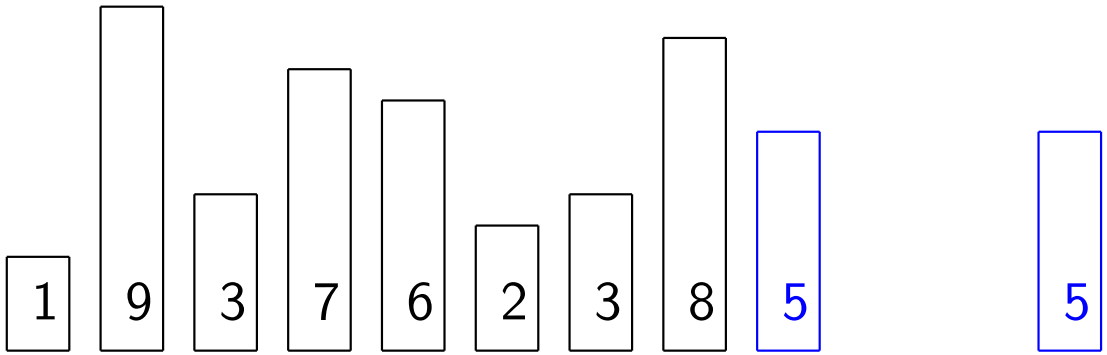
```
int particionar(int vetor[], int ini, int fim) {
    int pivo;

    pivo = vetor[fim];

    while (ini < fim) {
        while (ini < fim && vetor[ini] <= pivo)
            ini++;

        while (ini < fim && vetor[fim] > pivo)
            fim--;

        troca(&vetor[ini], &vetor[fim]);
    }

    return ini; // ini é a posição do primeiro elemento grande
}
```

Particionamento



Algoritmo

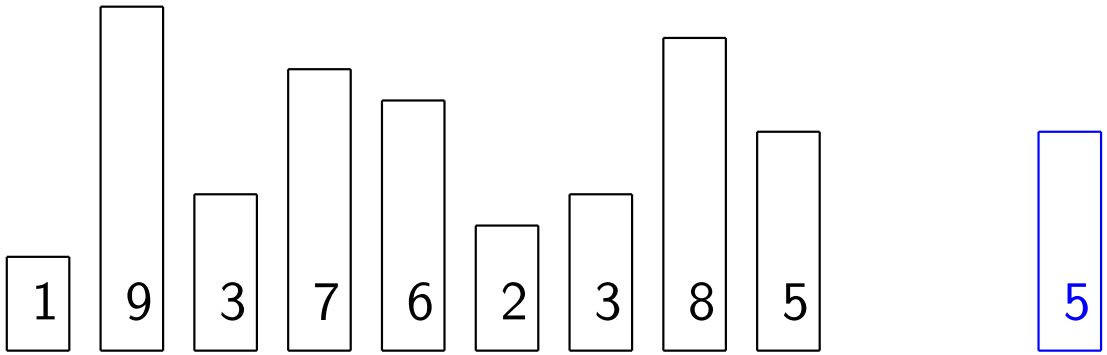
Particionamento



Algoritmo

1. Obtemos o valor do pivô:

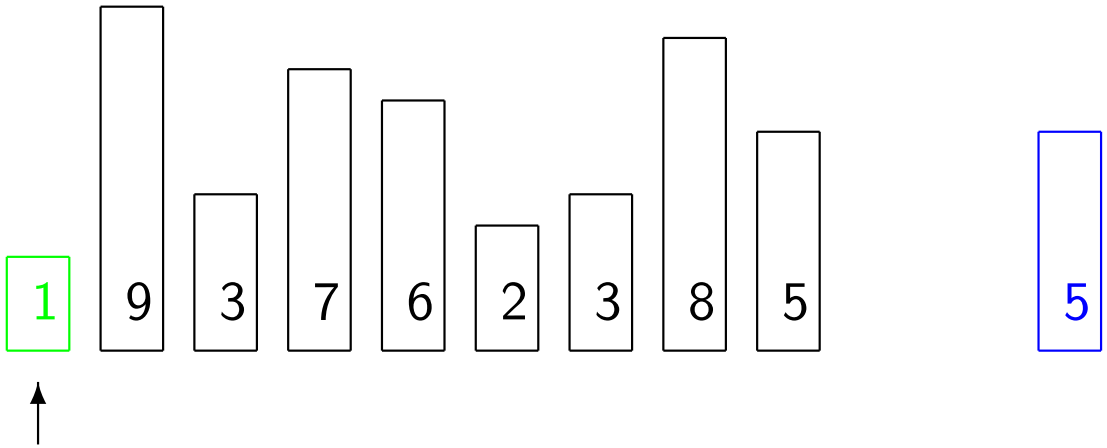
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:

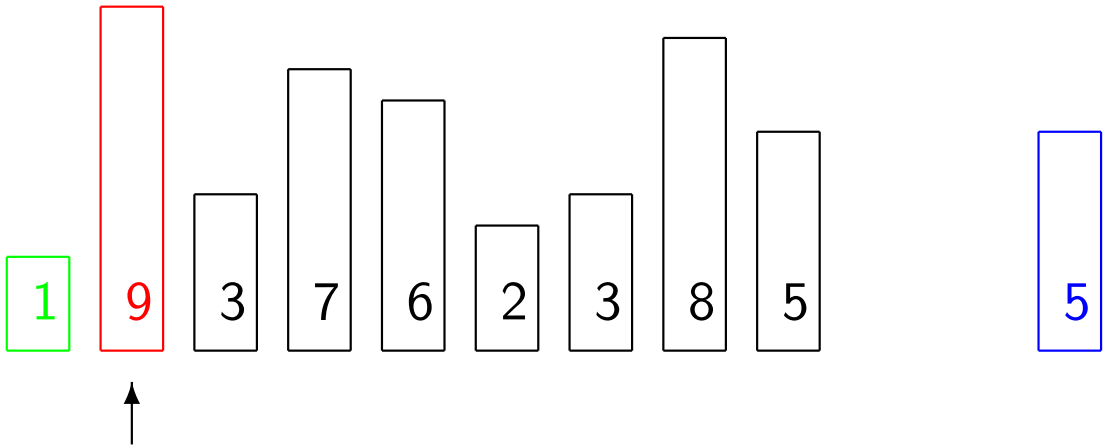
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô

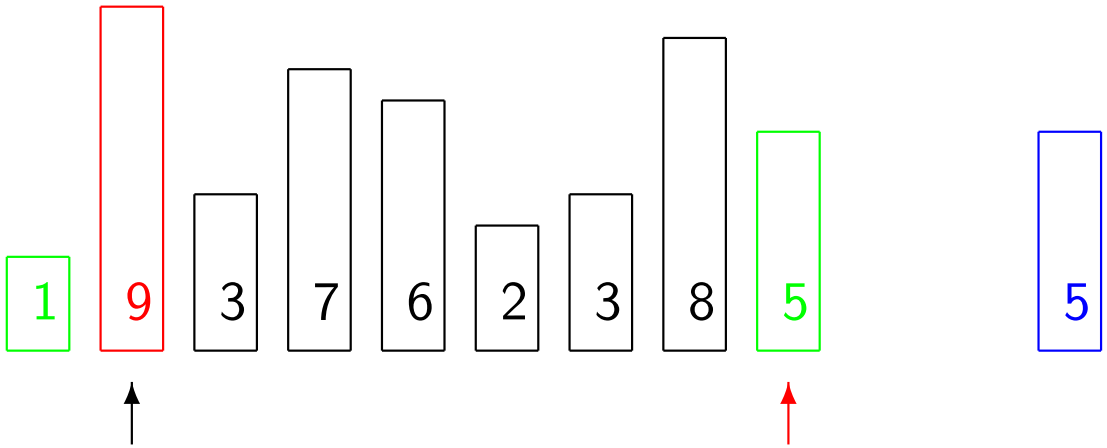
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô

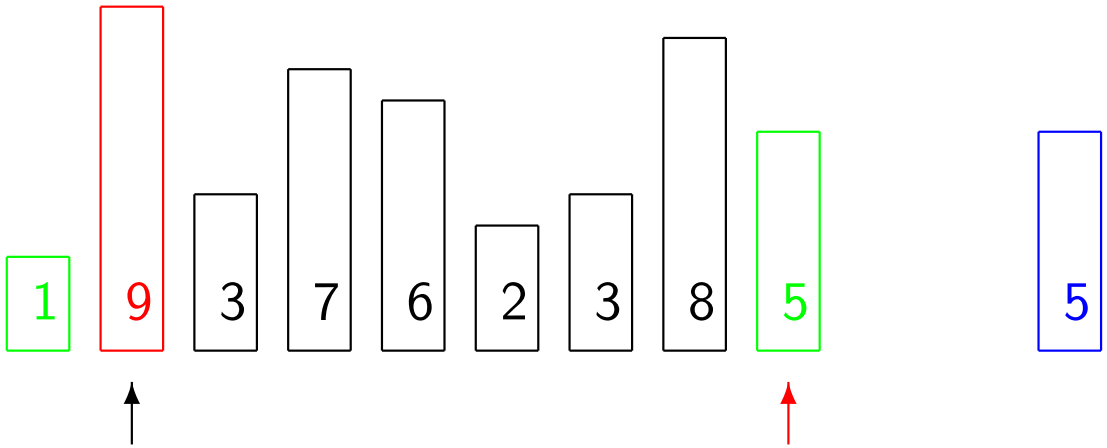
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô

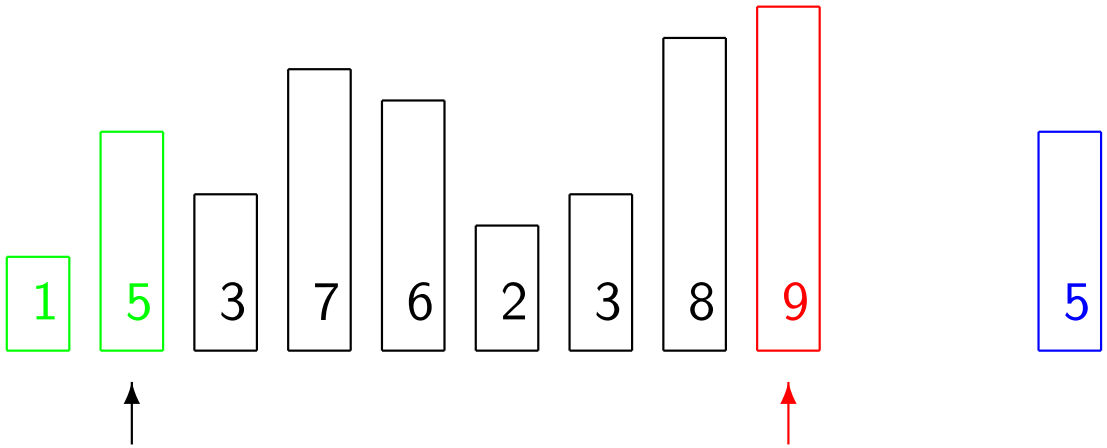
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas

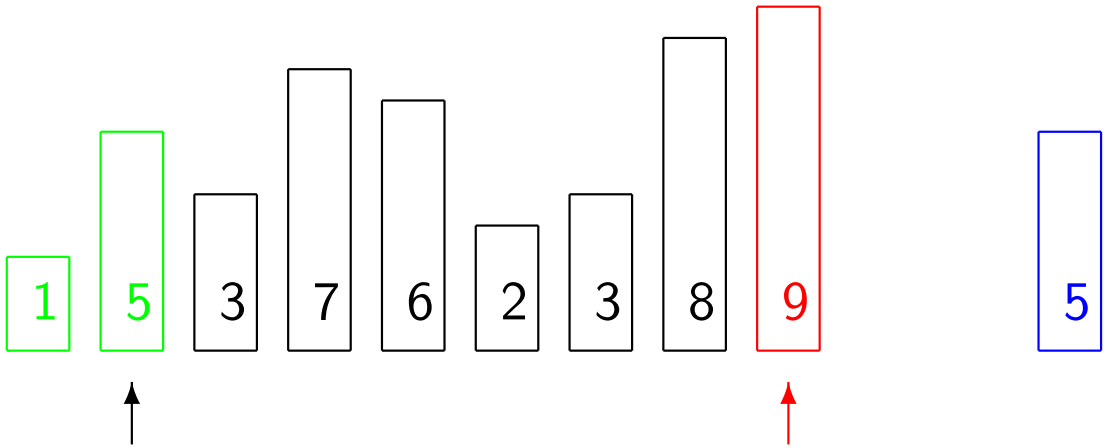
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas

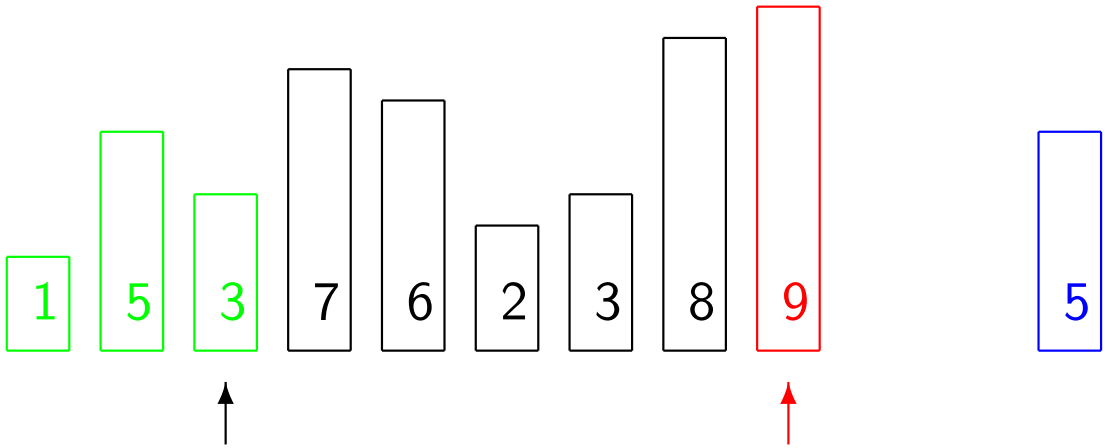
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

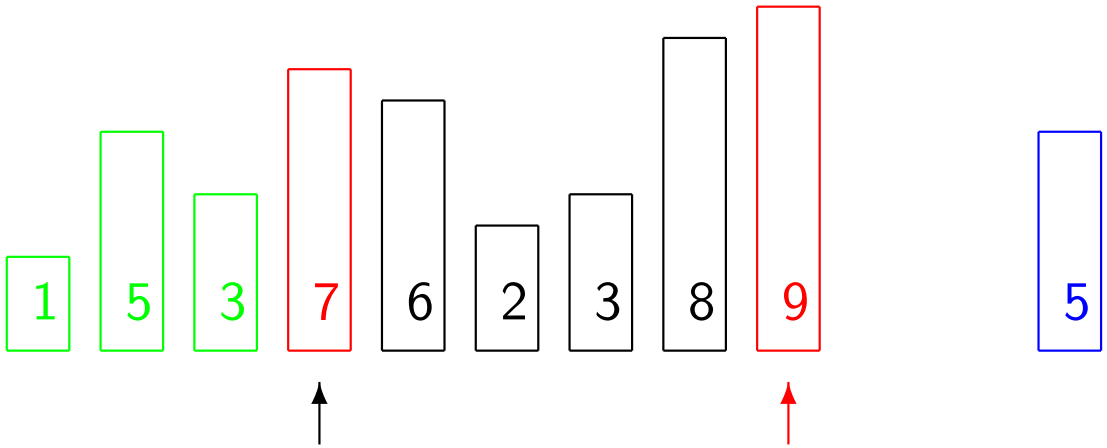
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

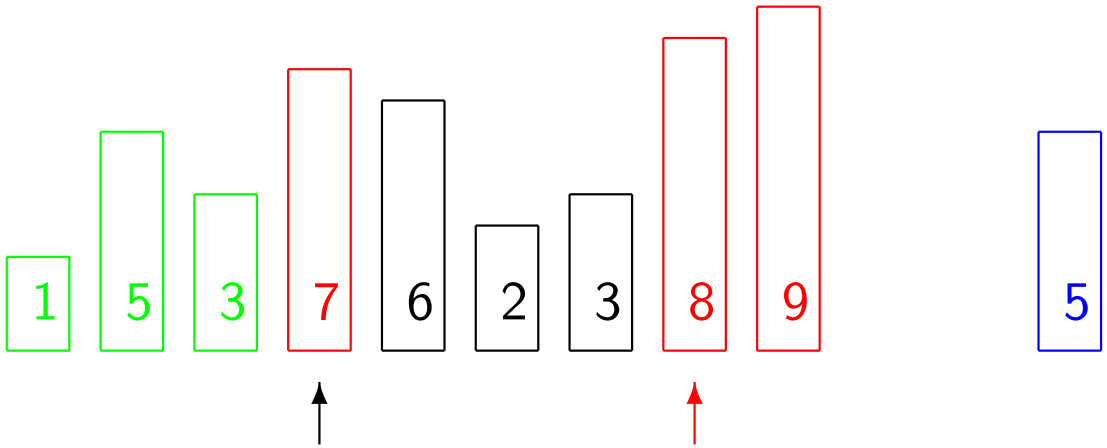
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

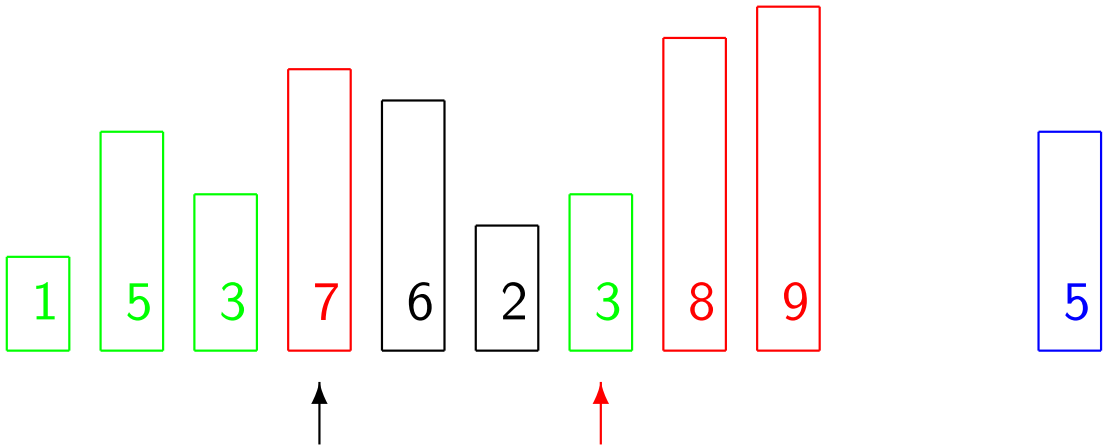
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

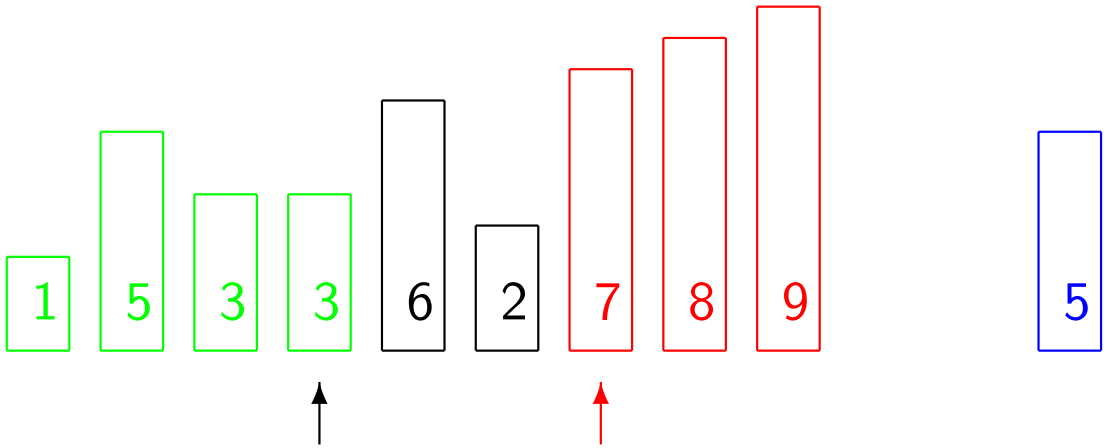
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

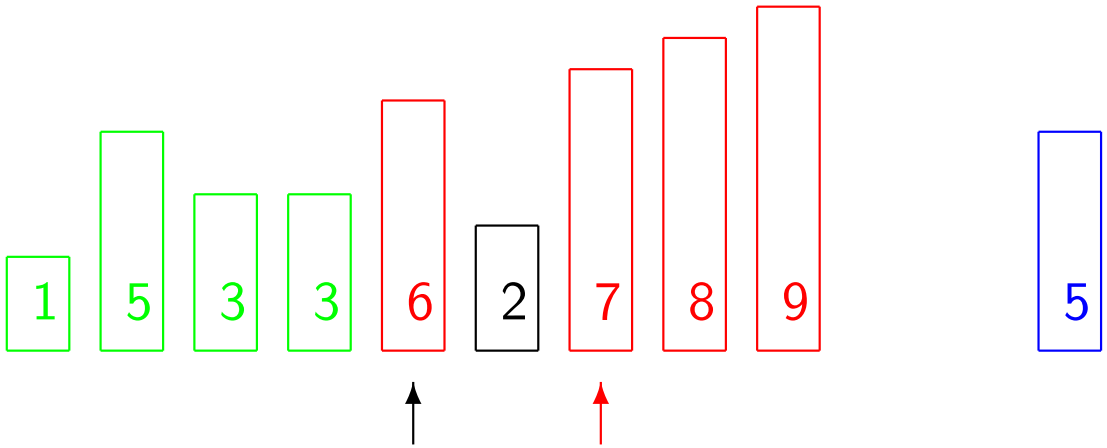
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

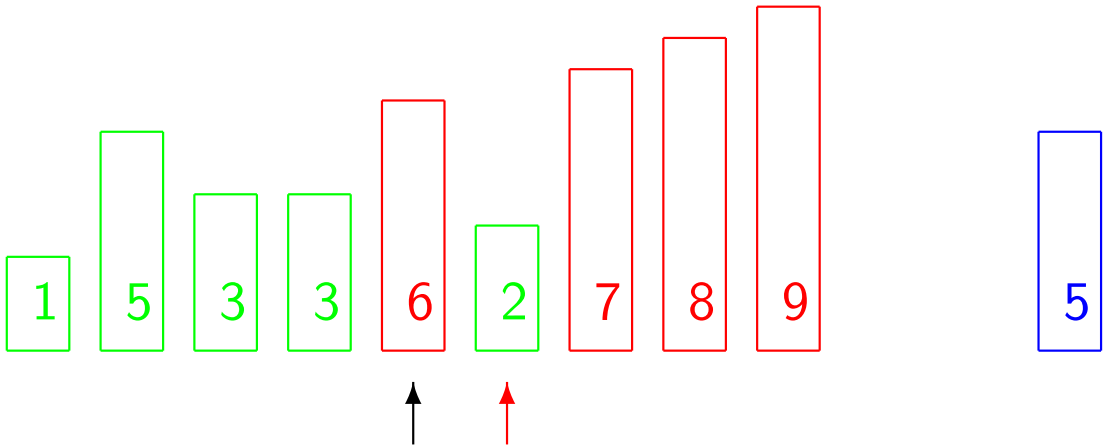
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

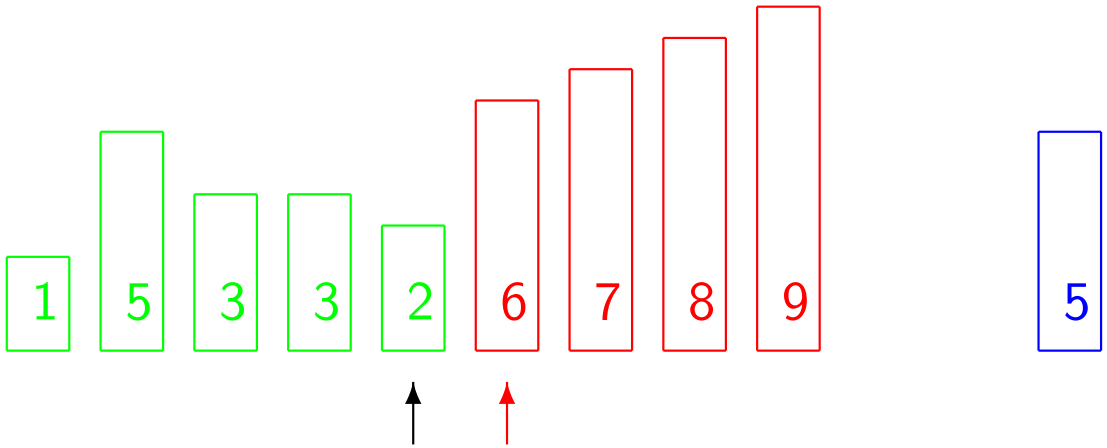
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

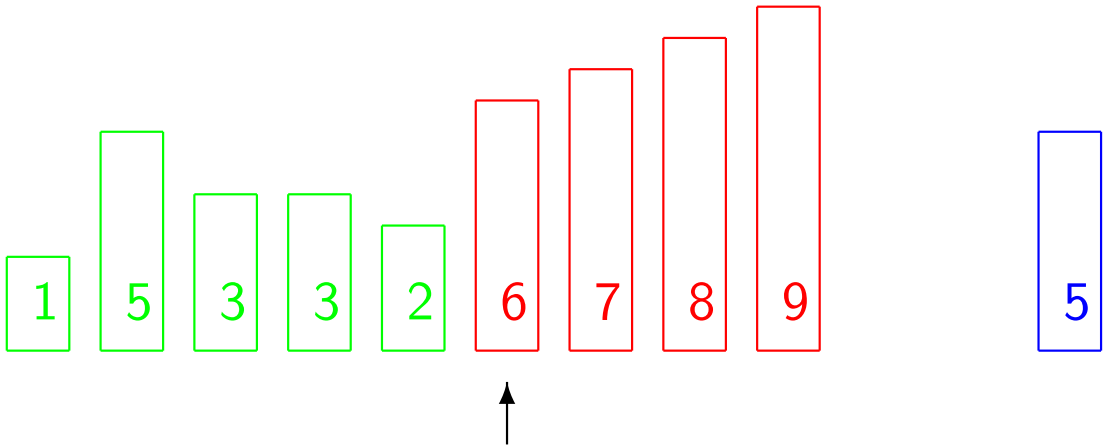
Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

Particionamento



Algoritmo

1. Obtemos o valor do pivô:
2. Procuramos elementos fora de ordem:
   - **do início ao fim**: em busca de elementos maiores que o pivô
   - **do fim ao início**: em busca de elementos menores ou iguais ao pivô
3. Trocamos os elementos em posições erradas
4. Continuamos passo 2 até índices se encontrarem

Exercício 1

1. Implemente uma busca binária de maneira **recursiva**.
2. No algoritmo de intercalação, alocamos um vetor auxiliar com tamanho máximo fixo `MAX`. Vimos que podemos flexibilizar essa restrição usando `malloc` para alocar memória no início da função e `free` para liberar memória no final da função. No entanto, essas funções são bem caras, isso é, gastam muito tempo, o que pode fazer com que o algoritmo fique mais lento já que a função `intercalar` pode ser chamada diversas vezes. Como você resolveria esse problema?

Exercício 2 - Eficiência do QuickSort

Ao contrário do *MergeSort* que tem complexidade de pior caso $O(n \log n)$, o *QuickSort* tem complexidade de pior caso $O(n^2)$ (na verdade, exatamente $\Theta(n^2)$). Surpreendentemente, na prática esse algoritmo é tão rápido quanto o *MergeSort* e (algumas vezes) até mais eficiente. Tente responder as questões a seguir:

1. Descreva o comportamento do particionamento para vetores: $\{1, 2, 3, 4, 5\}$, $\{5, 4, 3, 2, 1\}$, $\{1, 5, 4, 2, 3\}$. Pode ser útil tentar simular os passos iniciais.
2. A partição do vetor nem sempre será balanceada, isso é, nem sempre irá dividir o vetor em partes iguais. Você consegue dar um exemplo de vetor com n posições em que toda chamada de particionamento deixa uma parte com apenas um elemento?
3. Com que frequência você acha que o exemplo do item anterior acontece na prática? Tente pensar em uma maneira que diminua essa probabilidade (é necessário que o pivô sempre seja o último elemento?).