MC-202
Divisão e Conquista, MergeSort e Quicksort

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

Universidade Estadual de Campinas

Atualizado em: 2025-10-08 11:20

Na unidade anterior...

Vimos três algoritmos de ordenação $\mathrm{O}(n^2)$:

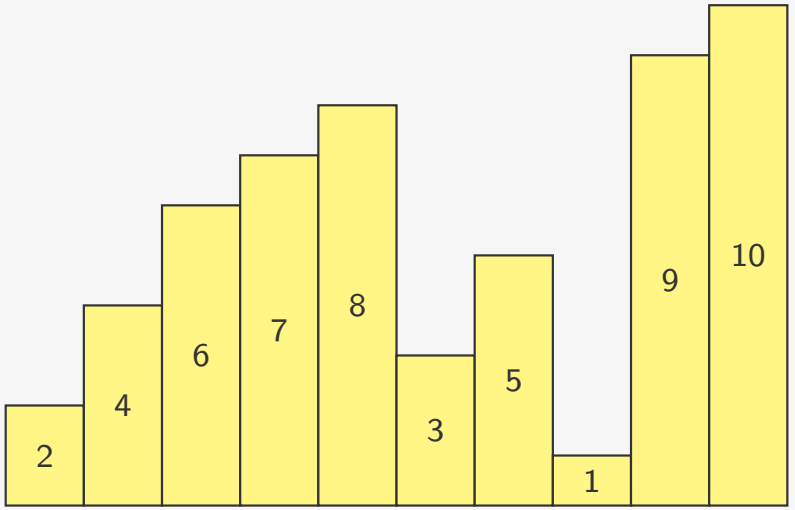
- `selectionsort`
- `bubblesort`
- `insertionsort`

E um algoritmo de ordenação $\mathrm{O}(n \lg n)$

- `heapsort`

Nessa unidade veremos mais dois algoritmos de ordenação

Estratégia: recursão

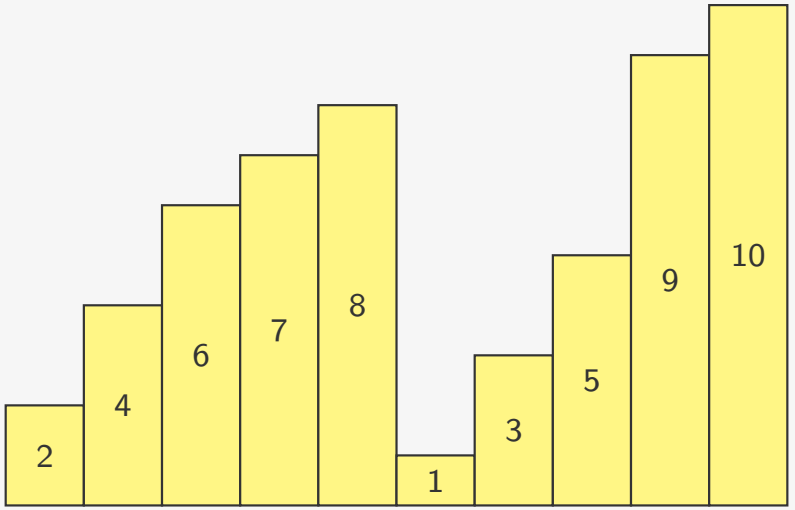


Como ordenar a primeira metade do vetor?

- usamos `ordenar(int *v, int esq, int dir)`
  - ordena o vetor das posições `esq` a `dir` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente
  - mas faremos algo mais simples e melhor
- executamos `ordenar(v, 0, 4);`

E se quiséssemos ordenar a segunda parte?

Ordenando a segunda parte



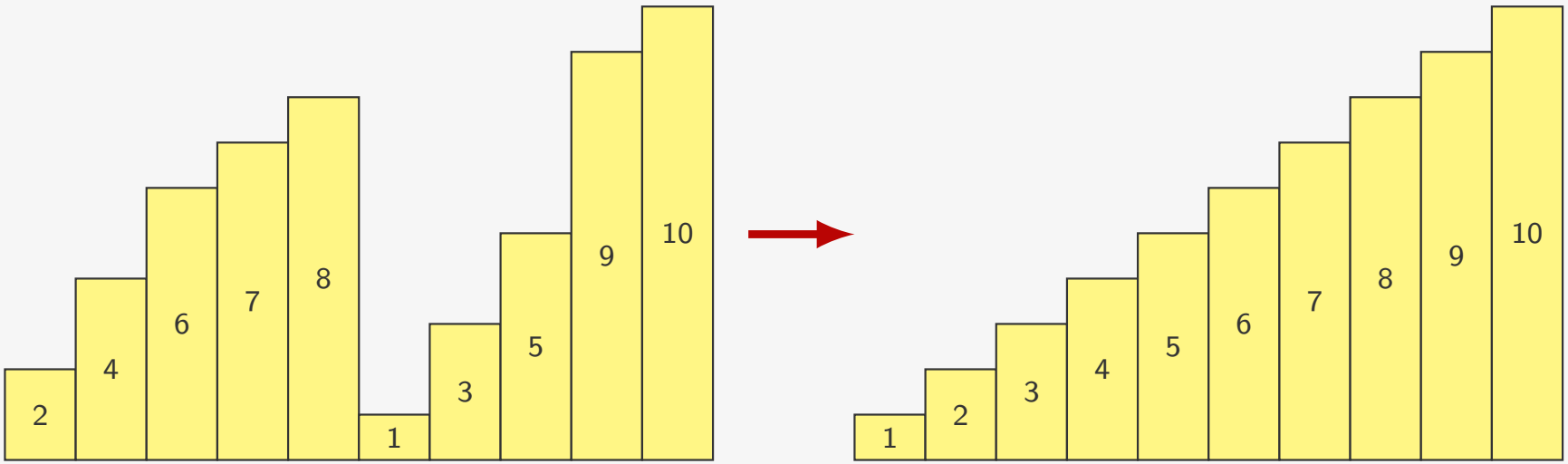
Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

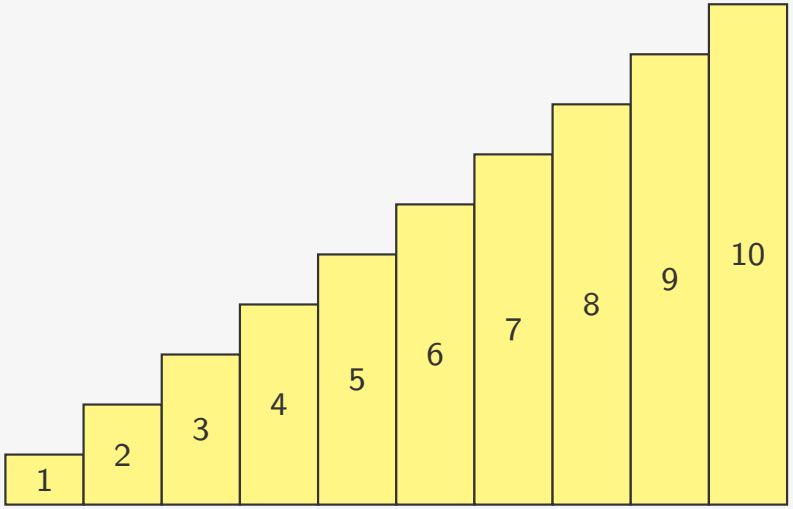
Ordenando todo o vetor

Se temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?



Intercalando



- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o original

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:

- Divisão: Quebramos o problema em vários subproblemas menores
  - ex: quebramos um vetor a ser ordenado em dois
- Conquista: Combinamos a solução dos problemas menores
  - ex: intercalamos os dois vetores ordenados

Ordenação por intercalação (*MergeSort*)

Intercalação:

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

Ordenação por intercalação (*MergeSort*)

```
1  void merge(int *v, int esq, int meio, int dir) {
2    int aux[MAX]; // podia ter feito alocação dinâmica
3    int i = esq, j = meio + 1, k = 0;
4    // intercala
5    while (i <= meio && j <= dir)
6      if (v[i] <= v[j])
7        aux[k++] = v[i++];
8      else
9        aux[k++] = v[j++];
10   // copia o resto do subvetor que não terminou
11   while (i <= meio)
12     aux[k++] = v[i++];
13   while (j <= dir)
14     aux[k++] = v[j++];
15   // copia de volta para v
16   for (i = esq, k = 0; i <= dir; i++, k++)
17       v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em `i` ou em `j`
- no máximo $n := r - l + 1$

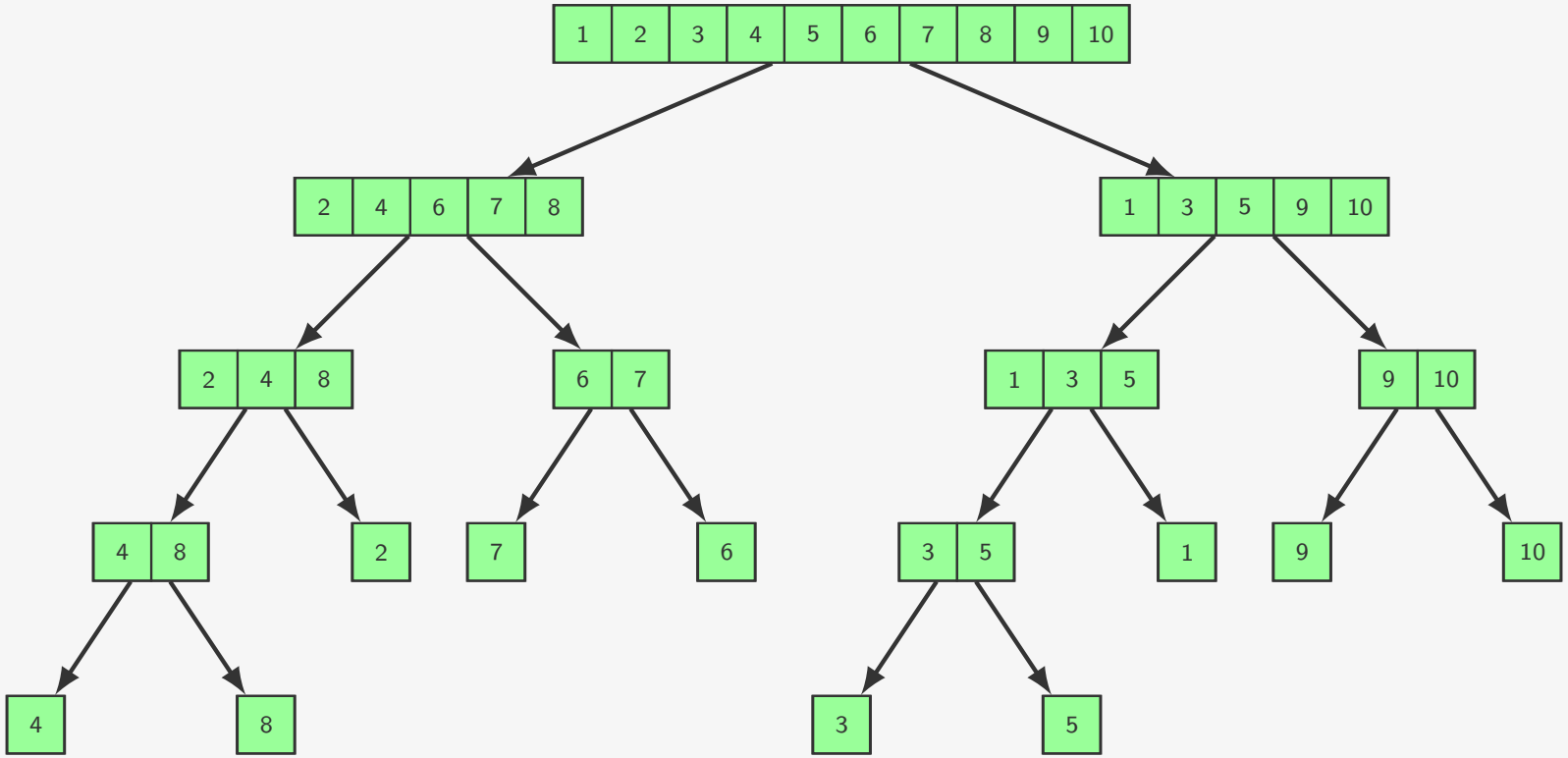
Ordenação por intercalação (*MergeSort*)

Ordenação:

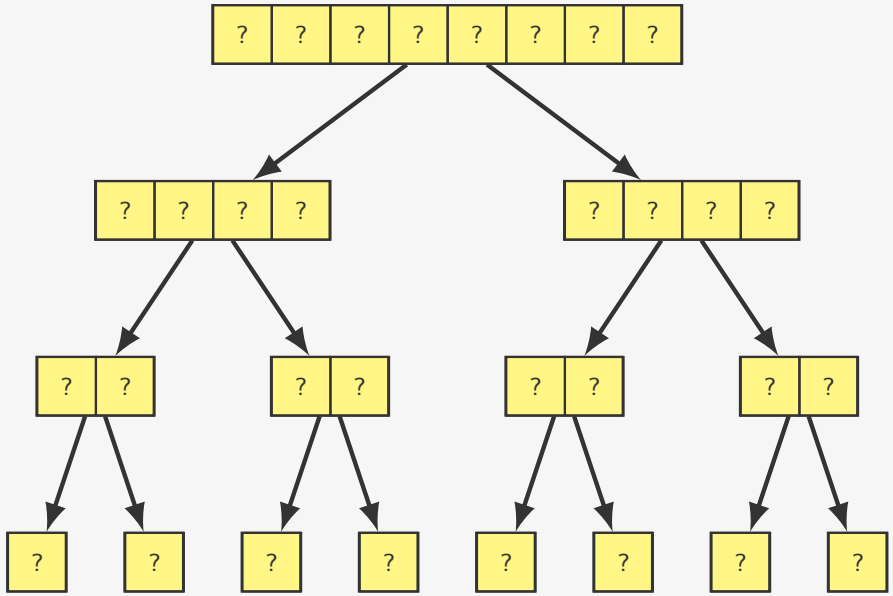
- Recebemos um vetor `v` de tamanho n com limites:
  - O vetor começa na posição `v[esq]`
  - O vetor termina na posição `v[dir]`
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho 0 ou 1

```
1  void mergesort(int *v, int esq, int dir) {
2    int meio = (esq + dir) / 2;
3    if (esq < dir) {
4      // divisão
5      mergesort(v, esq, meio);
6      mergesort(v, meio + 1, dir);
7      // conquista
8      merge(v, esq, meio, dir);
9    }
10 }
```

Simulação

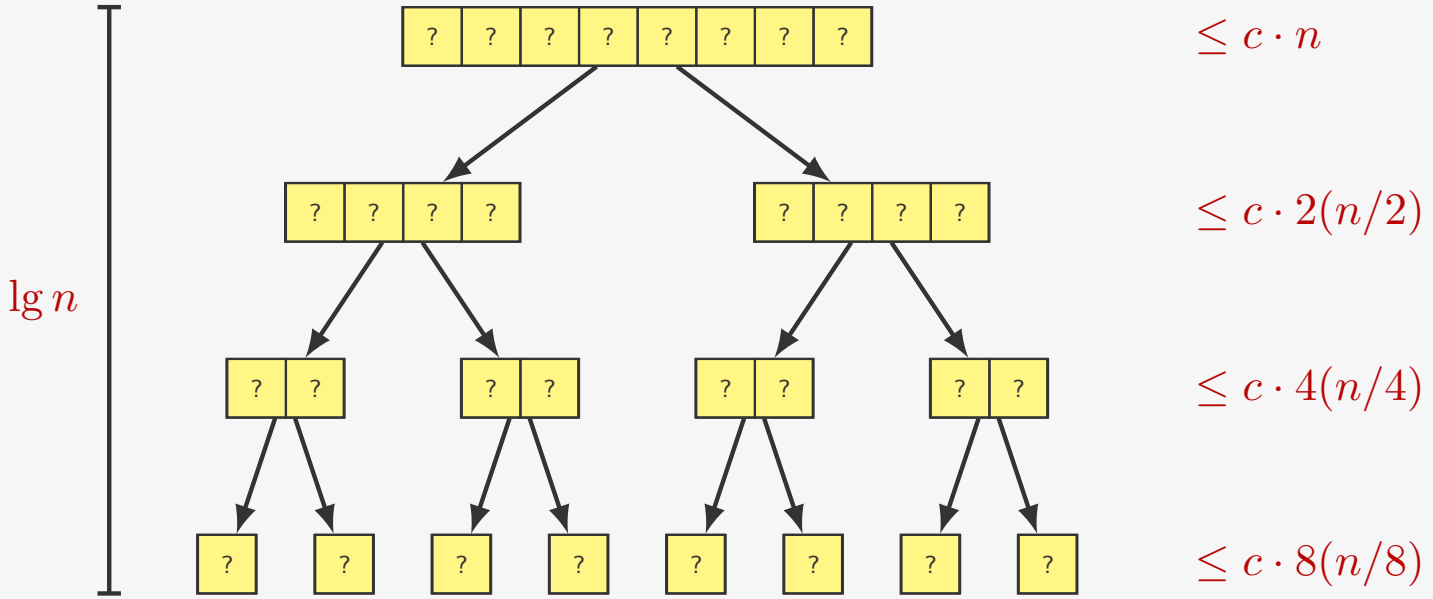


Tempo de execução para $n = 2^l$



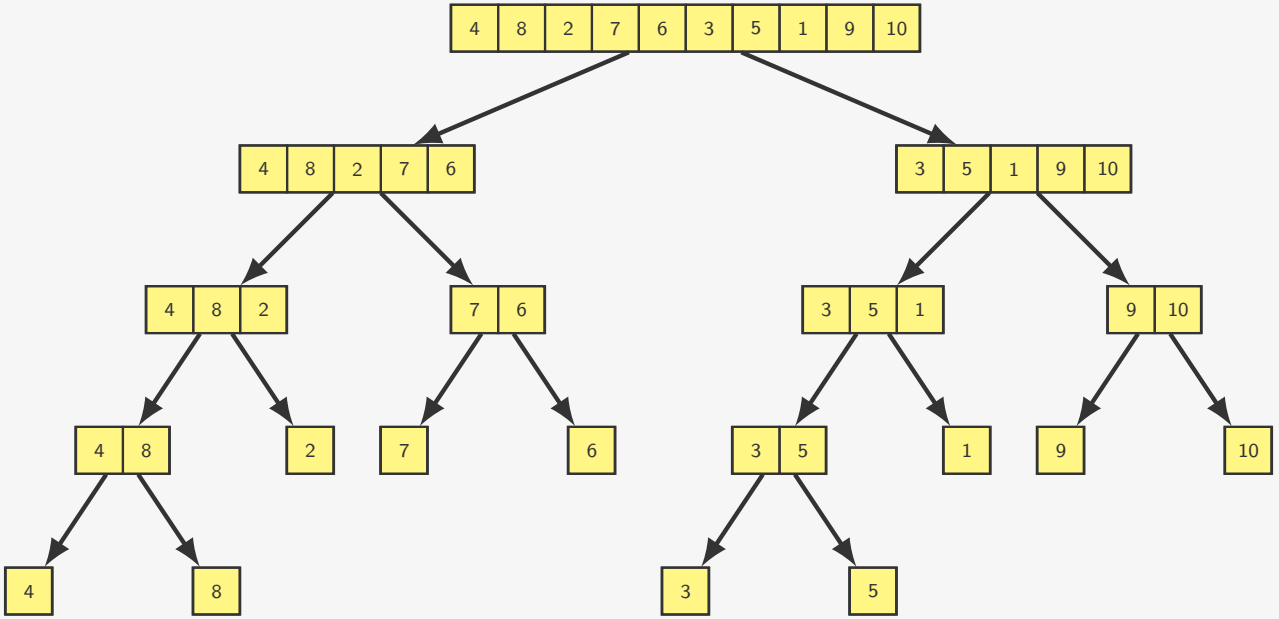
- No primeiro nível fazemos um `merge` com n elementos
- No segundo fazemos dois `merge` com $n/2$ elementos
- No $(k-1)$-ésimo fazemos 2^k `merge` com $n/2^k$ elementos
- No último gastamos tempo constante n vezes

Tempo de execução para $n = 2^l$



- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor ou igual a 1
  - Ou seja, $l = \lg n$
- Tempo total: $c\,n \lg n = \mathrm{O}(n \lg n)$

Tempo de execução para n qualquer

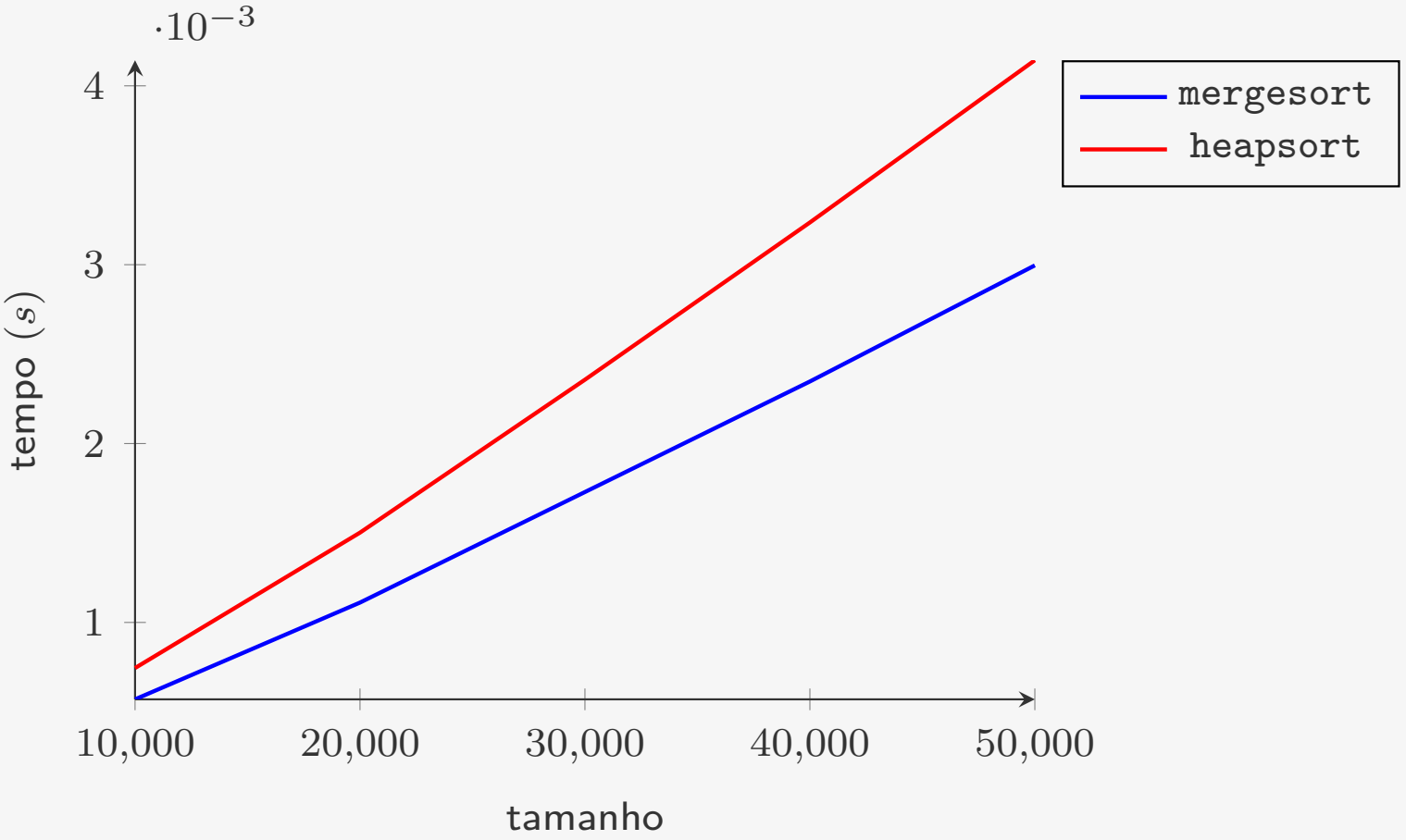


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
  - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n = \mathrm{O}(n \lg n)$$

Comparação entre `mergesort` e `heapsort`



`mergesort` é mais rápido do que o `heapsort`

- mas precisa de memória adicional
  - tanto para o vetor auxiliar - $\mathrm{O}(n)$
  - quanto para a pilha de recursão - $\mathrm{O}(\lg n)$

Quicksort - Ideia

- Escolhemos um pivô (ex: 4)
- Colocamos

Quicksort

```
1 int partition(int *v, int esq, int dir);
```

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

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

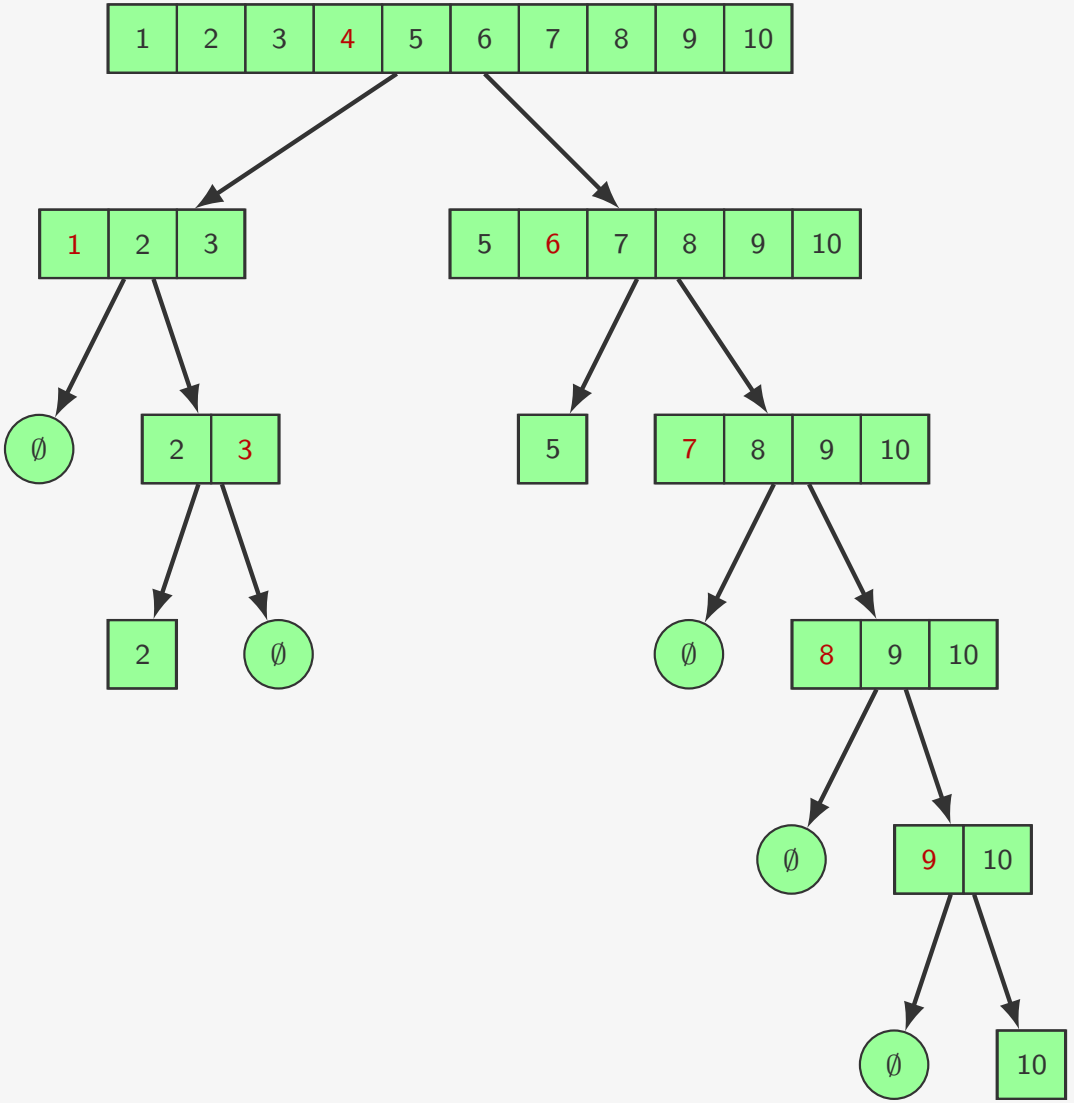
- Basta particionar o vetor em dois
- e ordenar o lado esquerdo e o direito

Como particionar um vetor?

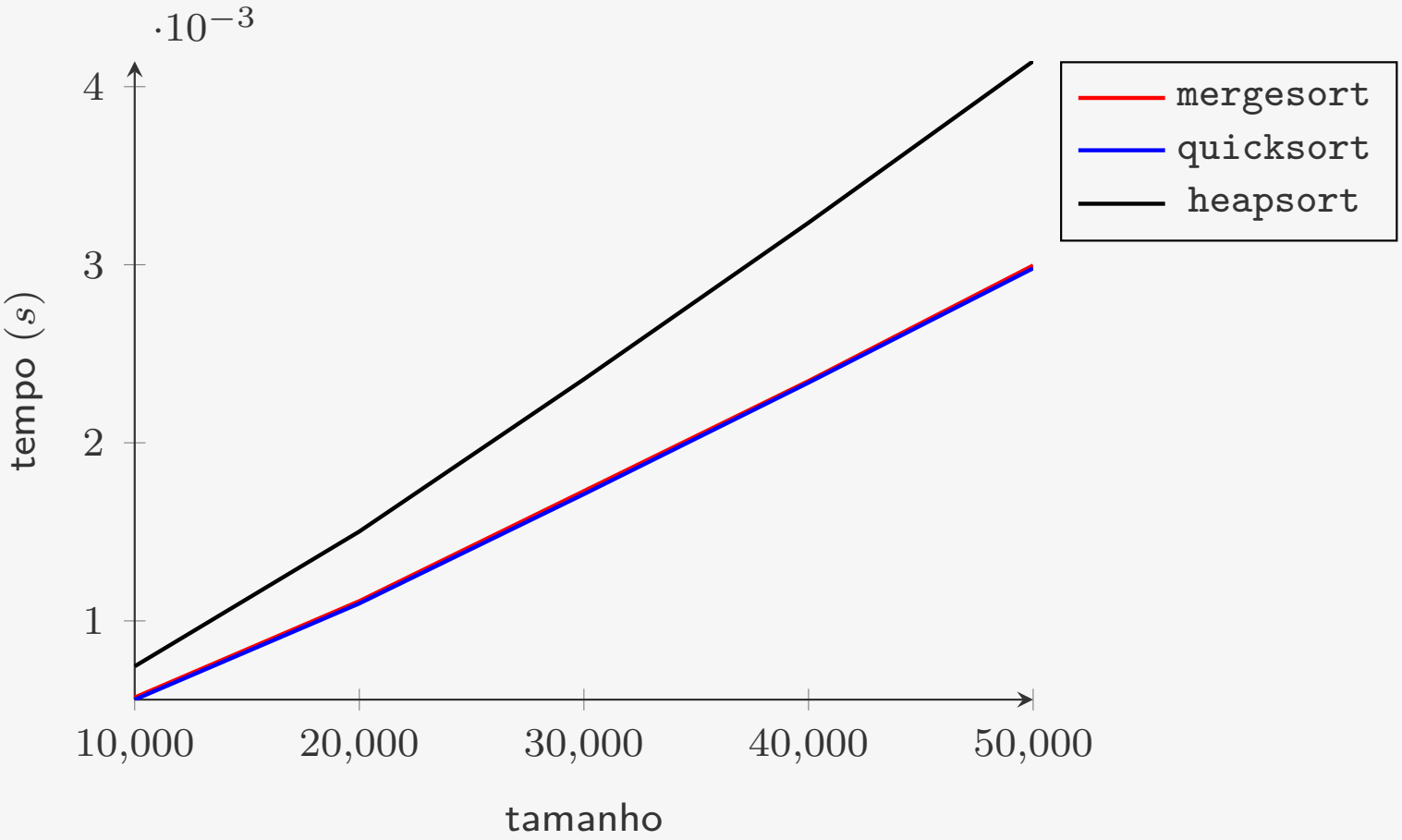
- Andamos da direita para a esquerda com um índice `i`
- De `i` até `pos - 1` ficam os menores do que o pivô
- De `pos` até `dir` ficam os maiores ou iguais ao pivô
- Se o elemento em `i` for maior ou igual ao pivô
  - diminuímos `pos` e realizamos uma troca de `i` com `pos`
- No final, o pivô está em `pos`

```
1  int partition(int *v, int esq, int dir) {
2    int pivo = v[esq], pos = dir + 1;
3    for (int i = dir; i >= esq; i--) {
4      if (v[i] >= pivo) {
5        pos--;
6        troca(&v[i], &v[pos]);
7      }
8    }
9    return pos;
10 }
```

Simulação do Quicksort



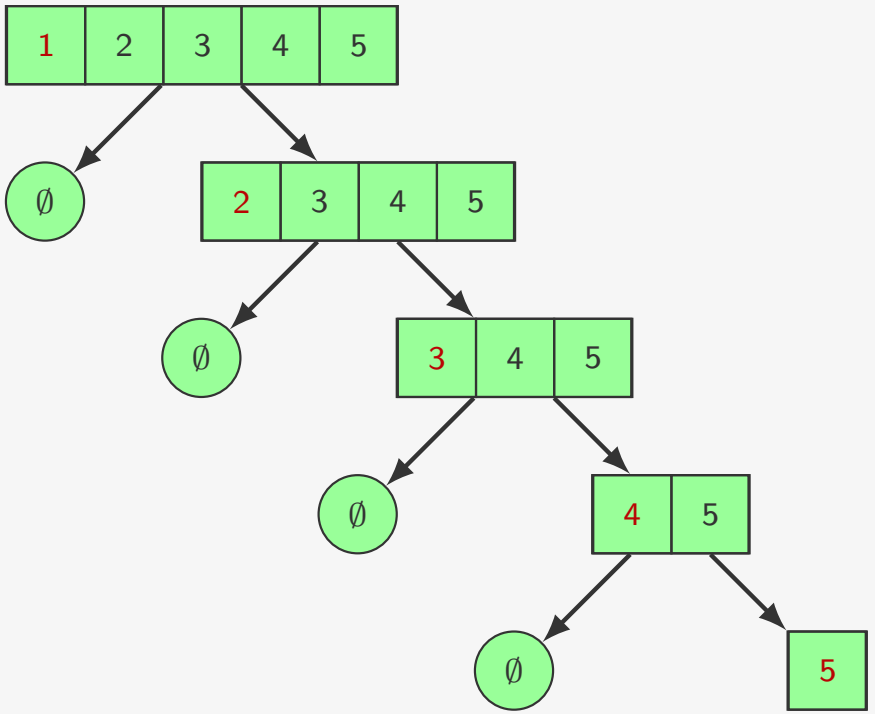
Comparação com o `mergesort` e `heapsort`



O `quicksort` foi levemente mais rápido do que o `mergesort`

- Mas ainda poderíamos otimizar o código dos três...
- Ou seja, um poderia ficar melhor do que o outro

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

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

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

Caso médio do QuickSort

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

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

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

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

Vamos ver duas formas de mitigar esse problema

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

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

Para isso:

- trocamos `v[(esq + dir) / 2]` com `v[esq + 1]`
- ordenamos `v[esq]`, `v[esq + 1]` e `v[dir]`
- particionamos `v[esq + 1]`, `...`, `v[dir - 1]`
  - `v[esq]` já é menor que o pivô
  - `v[dir]` já é maior que o pivô

Mediana de Três

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

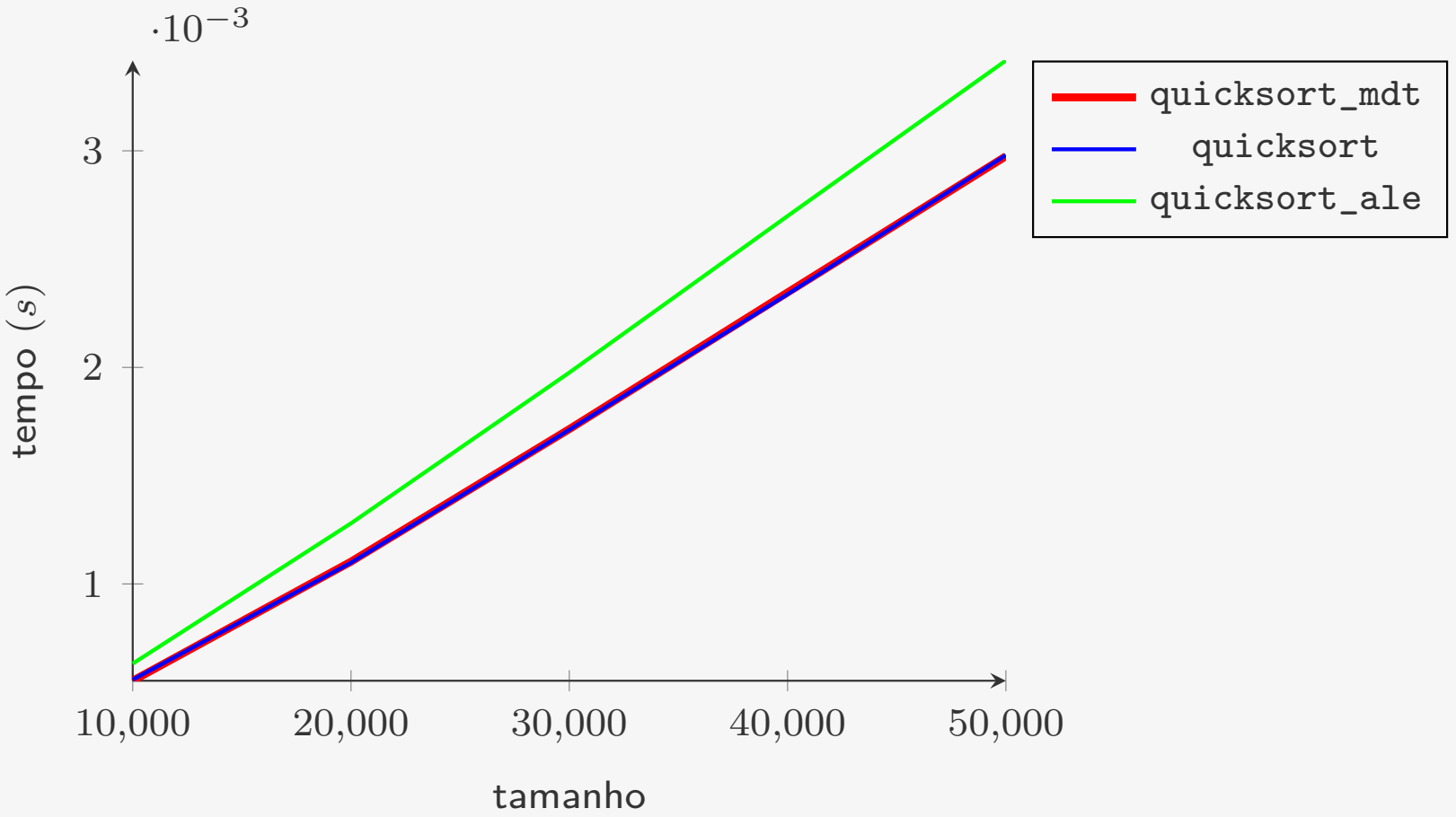
Quicksort Aleatorizado

```
1  int pivo_aleatorio(int esq, int dir) {
2    return esq + (int)(
3      (dir - esq + 1) *
4      (rand() / ((double)RAND_MAX + 1))
5    );
6  }
7
8  void quicksort_ale(int *v, int esq, int dir) {
9    int pos_pivo;
10   if(dir <= esq) return;
11   troca(&v[pivo_aleatorio(esq, dir)], &v[esq]);
12   pos_pivo = partition(v, esq, dir);
13   quicksort_ale(v, esq, pos_pivo - 1);
14   quicksort_ale(v, pos_pivo + 1, dir);
15 }
```

O tempo de execução depende dos pivôs sorteados

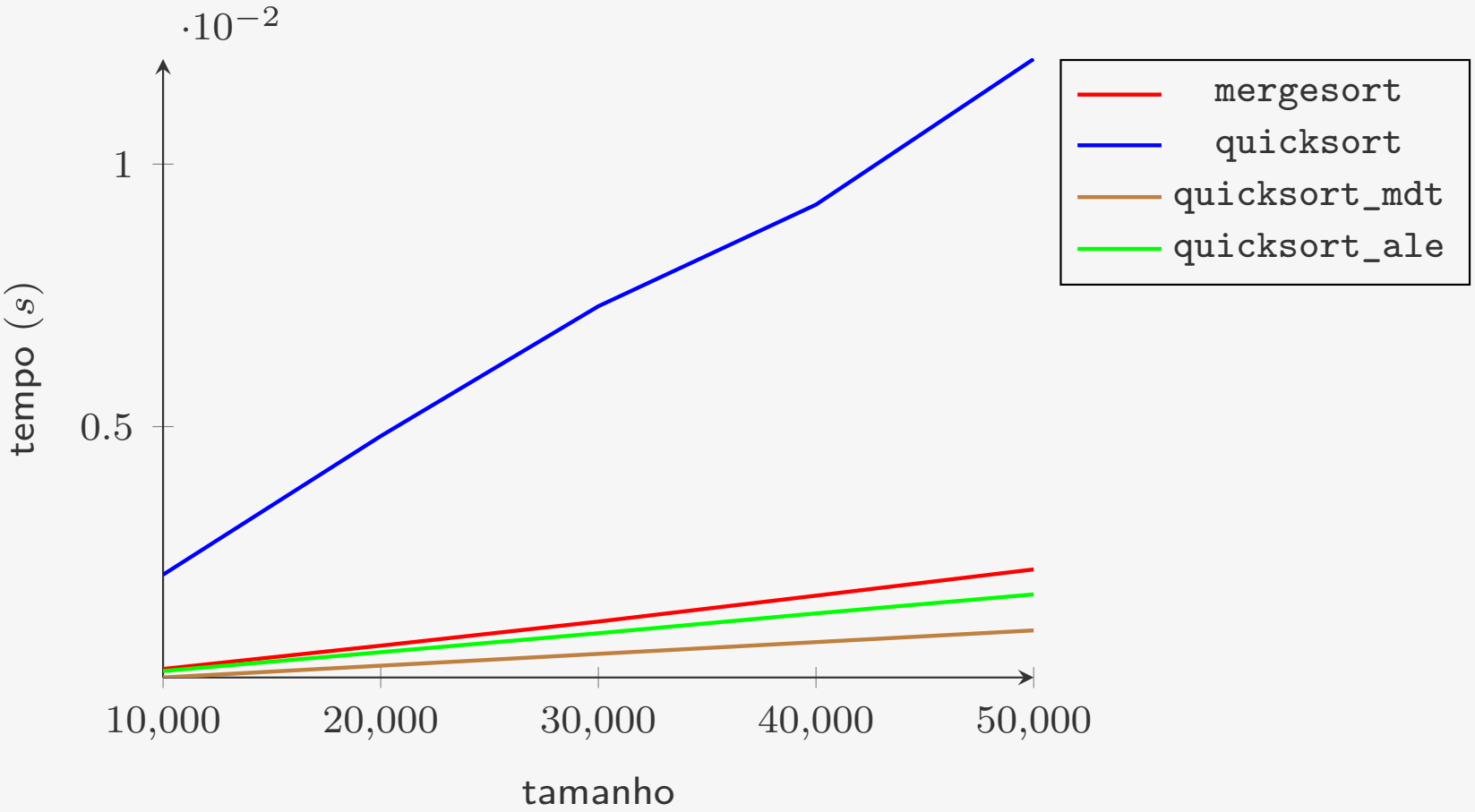
- O tempo médio é $\mathrm{O}(n \lg n)$
  - as vezes é lento, as vezes é rápido
  - mas não depende do vetor dado

Experimentos - Vetores aleatórios



`quicksort_ale` adiciona um overhead desnecessário

Experimentos - vetores quase ordenados



$0{,}5\%$ de trocas entre pares escolhidos aleatoriamentes

- `quicksort_mdt` é melhor
  - é esperado já que para vetores ordenados ele é $\mathrm{O}(n \lg n)$

Conclusão

O MergeSort é um algoritmo de ordenação $\mathrm{O}(n \lg n)$

- Em geral, melhor do que o HeapSort
- Mas precisa de espaço adicional $\mathrm{O}(n)$

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

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

Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$\mathrm{O}(n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(1)$
SelectionSort	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(1)$
InsertionSort	$\mathrm{O}(n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(1)$
HeapSort	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n \lg n)$	$\mathrm{O}(1)$
MergeSort	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n)$
QuickSort	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n \lg n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n)$

Exercício

Faça uma versão do MergeSort para listas ligadas.

Solução

```
1  p_no merge(p_no lista1, p_no lista2) {
2    p_no nova_lista = NULL;
3    if (lista1 == NULL)
4      return lista2;
5    if (lista2 == NULL)
6      return lista1;
7    if (lista1->dado <= lista2->dado) {
8      nova_lista = lista1;
9      nova_lista->prox = merge(lista1->prox, lista2);
10   } else {
11     nova_lista = lista2;
12     nova_lista->prox = merge(lista1, lista2->prox);
13   }
14   return nova_lista;
15 }
```

Solução

```
16 p_no merge_sort(p_no lista) {
17   if (lista == NULL || lista->prox == NULL)
18     return lista;
19   p_no lento = lista, rapido = lista->prox;
20   while (rapido != NULL && rapido->prox != NULL) {
21     // avança lento uma posição e rapido duas
22     lento = lento->prox;
23     rapido = rapido->prox->prox;
24   }
25   p_no lista1 = lista, lista2 = lento->prox;
26   lento->prox = NULL; // separa a lista em duas
27   return merge(merge_sort(lista1), merge_sort(lista2));
28 }
```

Exercício

Faça uma versão do QuickSort que seja boa para quando há muitos elementos repetidos no vetor.

- A ideia é particionar o vetor em três partes: menores, iguais e maiores que o pivô

Solução

```
1  void partition(int *v, int esq, int dir,
2                 int *meio_esq, int *meio_dir) {
3    int pivo = v[esq];
4    *meio_esq = *meio_dir = dir + 1;
5    for (int i = dir; i >= esq; i--) {
6      if (v[i] >= pivo) {
7        (*meio_esq)--;
8        troca(&v[i], &v[*meio_esq]);
9        if (v[*meio_esq] > pivo) {
10         (*meio_dir)--;
11         troca(&v[*meio_esq], &v[*meio_dir]);
12       }
13     }
14   }
15 }
16
17
18 void quicksort(int *v, int esq, int dir) {
19   int meio_esq, meio_dir;
20   if (dir <= esq) return;
21   partition(v, esq, dir, &meio_esq, &meio_dir);
22   quicksort(v, esq, meio_esq - 1);
23   quicksort(v, meio_dir + 1, dir);
24 }
```

Exercício

Implemente a função
`void mergeAB(int *v, int *a, int n, int *b, int m)`
que dados vetores `a` e `b` de tamanho `n` e `m` faz a intercalação de `a` e `b` e armazena no vetor `v`. Suponha que `v` já está alocado e que tem tamanho maior ou igual a `n+m`.

Solução

```
1  void mergeAB(int *v, int *a, int n, int *b, int m) {
2    int i = 0, j = 0, k = 0;
3    while (i < n && j < m)
4      if (a[i] < b[j])
5        v[k++] = a[i++];
6      else
7        v[k++] = b[j++];
8    while (i < n)
9      v[k++] = a[i++];
10   while (j < m)
11     v[k++] = b[j++];
12 }
```

Dúvidas?