Ordenação e Busca

João Paulo Dias de Almeida
jp.dias.almeida@gmail.com

Universidade Federal de Sergipe

O que vamos aprender hoje?

- A importância dos algoritmos de ordenação
- Diferenciar as maneiras como uma ordenação pode ser conduzida
- Diferentes métodos de ordenação e suas respectivas análises assintóticas

Motivação

- **Ordenação** é um componente útil para solucionar diferentes problemas
- **Ideias interessantes** para criar algoritmos são utilizadas durante a ordenação também (e.g. divisão e conquista, estrutura de dados, algoritmos randômicos)
- A ordenação está entre os algoritmos **mais estudados da computação**
  - Cada um possui uma vantagem em particular
- Aprender como estes algoritmos funcionam pode ajudar a resolver outros problemas

Aplicações

- Muitos problemas importantes podem ser resolvidos com a ajuda da ordenação
  - Sem a ordenação tais algoritmos costumam ser resolvidos em $O(n^2)$
  - Com os dados ordenados, o algoritmo passa a ser $O(n \log n)$
- Exemplos de aplicações:
  - **Busca:**

    Busca binária encontra um item em um dicionário em $O(\log n)$, **desde que os itens estejam ordenados**

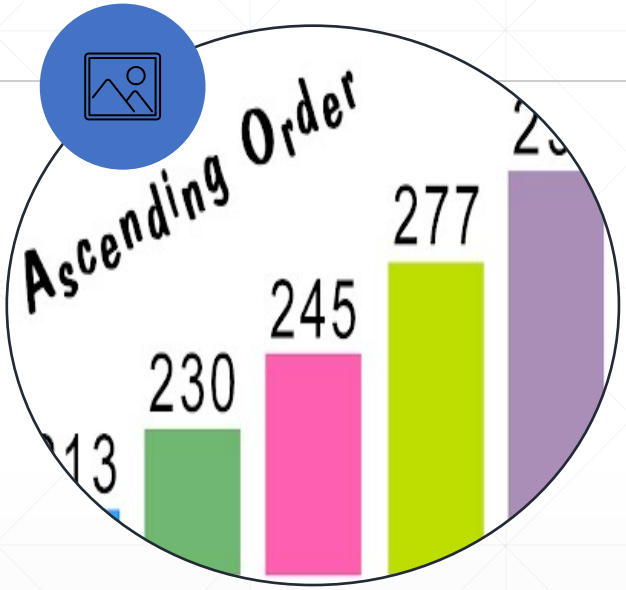
Aplicações

- **Pares próximos:**

  Como encontrar os pares de itens que possuem a menor diferença entre si? A ordenação dos itens resolve este problema, pois os pares estarão organizados. Além disso este algoritmo pode ser utilizado para identificar duplicatas

- **Frequência:**

  Identificar o item de um conjunto que ocorre com maior frequência é mais fácil com os itens ordenados. Inclusive para encontrar a frequência de um elemento específico



Como você faria para ordenar uma lista com 5 números de forma crescente?

Ordenação

- Dada uma sequência de itens, retornar a sequência ordenada
- Na prática:
  - Cada registro (item) é composto por uma chave (**key**) e pelos **dados satélites**
  - A ordenação é feita em relação a chave. Todos os demais dados são movidos em conjunto
  - Para dados satélites grandes, a ordenação deve ser feita apenas em relação à chave

Exemplo: chave e dados satélite

Chave

Dados satélites

key	c1	c2	c3	c4	c5
2	“nome2”	“cpf2”	“end2”	“tel2”	“sal2”
3	“nome3”	“cpf3”	“end3”	“tel2”	“sal2”
1	“nome1”	“cpf1”	“end1”	“tel1”	“sal1”
3	“nomex”	“cpfx”	“endx”	“telx”	“salx”
4	“nome4”	“cpf4”	“end4”	“tel4”	“sal4”
5	“nome5”	“cpf5”	“end5”	“tel5”	“sal5”

Exemplo: chave e dados satélite

key	c1	c2	c3	c4	c5
1	“nome1”	“cpf1”	“end1”	“tel1”	“sal1”
2	“nome2”	“cpf2”	“end2”	“tel2”	“sal2”
3	“nome3”	“cpf3”	“end3”	“tel2”	“sal2”
3	“nomex”	“cpfx”	“endx”	“telx”	“salx”
4	“nome4”	“cpf4”	“end4”	“tel4”	“sal4”
5	“nome5”	“cpf5”	“end5”	“tel5”	“sal5”

Métodos de ordenação

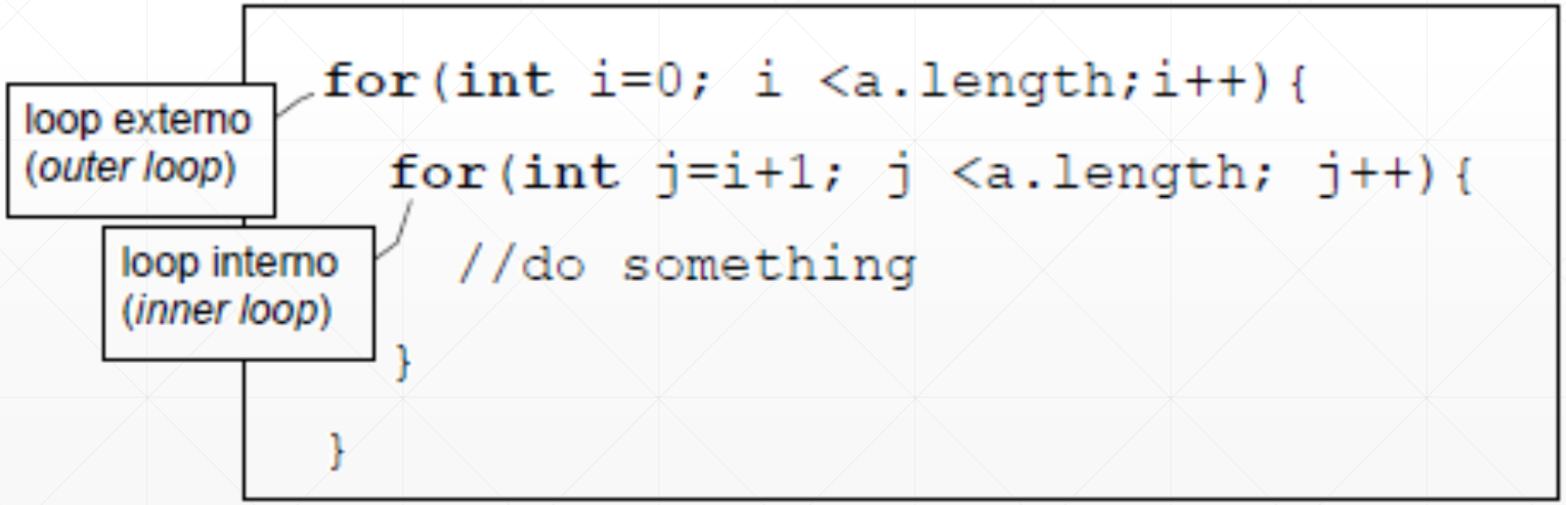
- Ordenação interna
  - Os dados ordenados cabem todos na memória principal
- Ordenação externa
  - É necessário usar o disco ou estrutura semelhante (e.g. fita) para armazenar os dados
  - Este caso é utilizado quando a memória não é suficiente para armazenar os dados
    - O disco funciona como um armazenamento auxiliar

Memoria: *in place* vs. *not in place*

- *In place* (no local)
  - Utiliza apenas a memória disponível para armazenar os dados (ex. próprio array)
  - Utiliza, além da memória disponível, uma quantidade fixa (constante) de memória
- *Not in place* (requer espaço extra)
  - Utiliza, além da memória disponível para armazenar os dados, uma quantidade que cresce proporcionalmente ao tamanho do array

Loop (laços): *inner* vs. *outer* loop

- *Inner/outer loop*
  - *Outer loop* → loop externo
  - *Inner lopp* → loop interno



Critério de parada

- Um **critério de parada** é uma condição que permite um algoritmo terminar
  - Pode ser um certo número de objetivos (gerações ou interações), permitindo retornar um resultado satisfatório
  - Pode ser uma condição que permite o algoritmo terminar mais cedo que o previsto, retornando o resultado correto

Ordenação Interna

Ordenação interna

- Os métodos de ordenação interna são classificados como:
  - **Métodos simples:**

    São métodos fáceis de entender

    Úteis para pequenas quantidades de dados

    Geralmente requerem $O(n^2)$ comparações
  - **Métodos eficientes**

    Adequados para quantidades maiores de dados

    Costumam requerer $O(n \log n)$ comparações

Bubble sort

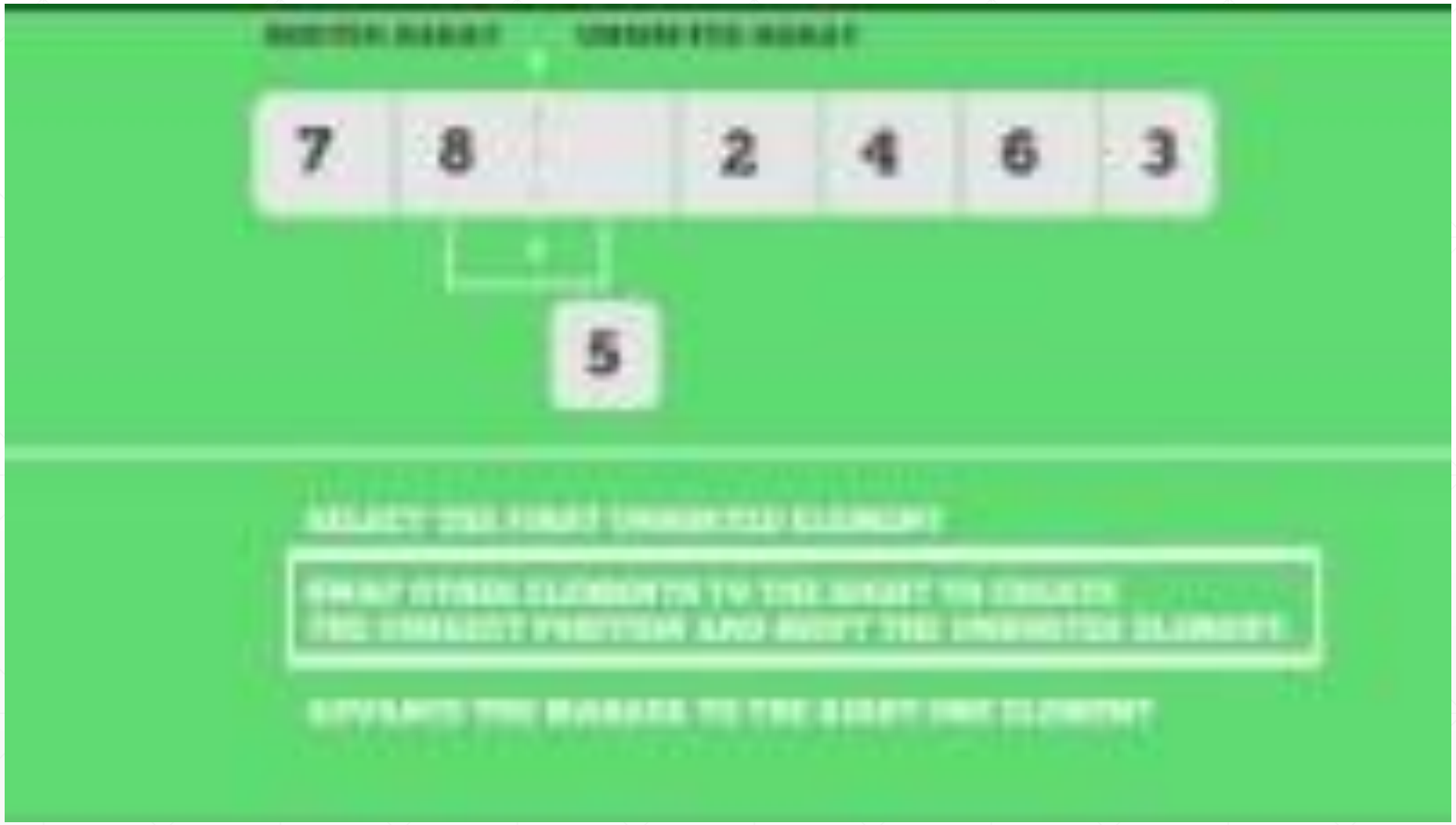
- É um algoritmo simples porém ineficiente
- Funciona ao trocar o elemento vizinho que está fora de ordem repetidas vezes
- O pior caso é $O(n^2)$
- Existem algumas variações deste algoritmo

Exemplo: Bubble sort

Insertion sort

- O array é virtualmente dividido em duas partes: ordenado e não-ordenado
  - Semelhante a alguém ordenando cartas com as mãos
- Valores da parte não ordenada são colocados na posição correta da parte ordenada
  - Geralmente a análise começa do segundo elemento da lista de itens
- No pior caso é $O(n^2)$

Exemplo: Insertion sort



Insertion sort - código

```
int n = arr.length;
for (int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while (j >= 0 && arr[j] > key) {
     arr[j + 1] = arr[j];
     j = j - 1;                }
  arr[j + 1] = key; }
```

Selection sort

- Este algoritmo encontra o menor valor do array e o coloca no início repetidas vezes
- Custo desse é algoritmo é $O(n^2)$
- Este algoritmo pode ser útil quando a movimentação dos valores a ser ordenado é custosa
  - Ele movimenta os valores do array em no máximo O(n)

Exemplo: Selection sort

arr[] = 64 25 12 22 11

// Encontre o menor elemento em arr[0...4] e o coloque no inicio

11 25 12 22 64

// Encontre o menor elemento em arr[1...4] e o coloque no inicio de arr[1...4]

11 12 25 22 64

// Encontre o menor elemento em arr[2...4] e o coloque no inicio de arr[2...4]

11 12 22 25 64

// Encontre o menor elemento em arr[3...4] e o coloque no inicio de arr[3...4]

11 12 22 25 64

Exemplo: Selection Sort

Métodos eficientes

Métodos eficientes

- São algoritmos com custo computacional de O (n log n)
- Dois algoritmos de ordenação bastante populares são o merge sort e o quick sort
  - São baseados na abordagem de divisão e conquista
- **Divisão e conquista:** resolve um problema recursivamente, aplicando três etapas em cada recursão

Divisão e conquista: etapas

1. **Divisão:** o problema é dividido em subproblemas. Desta forma, temos instâncias menores do mesmo problema
2. **Conquista:** resolva os subproblemas recursivamente. Se o subproblema é pequeno o bastante, apenas resolva os subproblemas diretamente.
3. **Combine:** junte as soluções dos subproblemas para criar a solução do problema original

Exemplo: Recursão

Merge sort

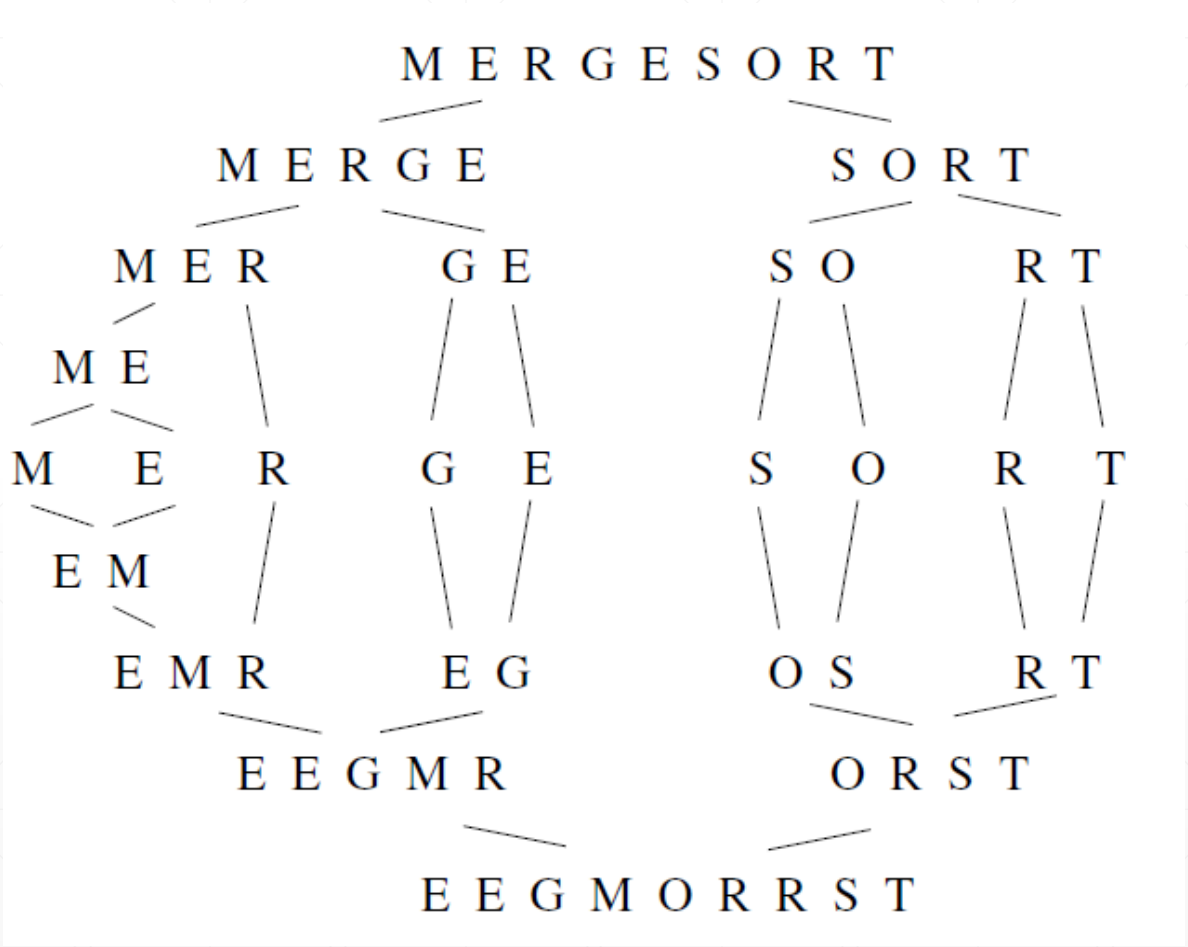
- É uma abordagem recursiva para ordenar itens
  1. Separa o conjunto de itens em duas partes
  2. Ordena cada uma dessas partes recursivamente
  3. Intercala os elementos dos dois subgrupos ordenados

$$\text{Mergesort}(A[1,n])$$
$$\text{Merge}(\ \text{MergeSort}(A[1,\lfloor n/2 \rfloor]),\ \text{MergeSort}(A[\lfloor n/2 \rfloor + 1, n])\ )$$

Merge sort

- O array é dividido em partes até chegar ao **caso base**
  - Caso base é quando o problema não precisa mais ser dividido e já pode ser resolvido
- No merge sort o caso base é quando houver apenas um elemento no array
- A eficiência desse algoritmo depende de quão eficiente conseguimos unir as partes

Exemplo: Merge Sort



- **Advantages of Divide and Conquer Algorithm:**
- The difficult problem can be solved easily.
- It divides the entire problem into subproblems thus it can be solved parallelly ensuring multiprocessing
- Efficiently uses cache memory without occupying much space
- Reduces time complexity of the problem

- **Disadvantages of Divide and Conquer Algorithm:**
- It involves recursion which is sometimes slow
- Efficiency depends on the implementation of logic
- It may crash the system if the recursion is performed rigorously

Vamos fazer juntos

Algoritmo: Merge sort

MergeSort(arr[], l, r)

if r > l

1. Encontre o meio do array para dividi-lo em dois:

   meio m = l + (r-l)/2

2. Invoque mergeSort para a primeira metade:

   Call mergeSort(arr, l, m)

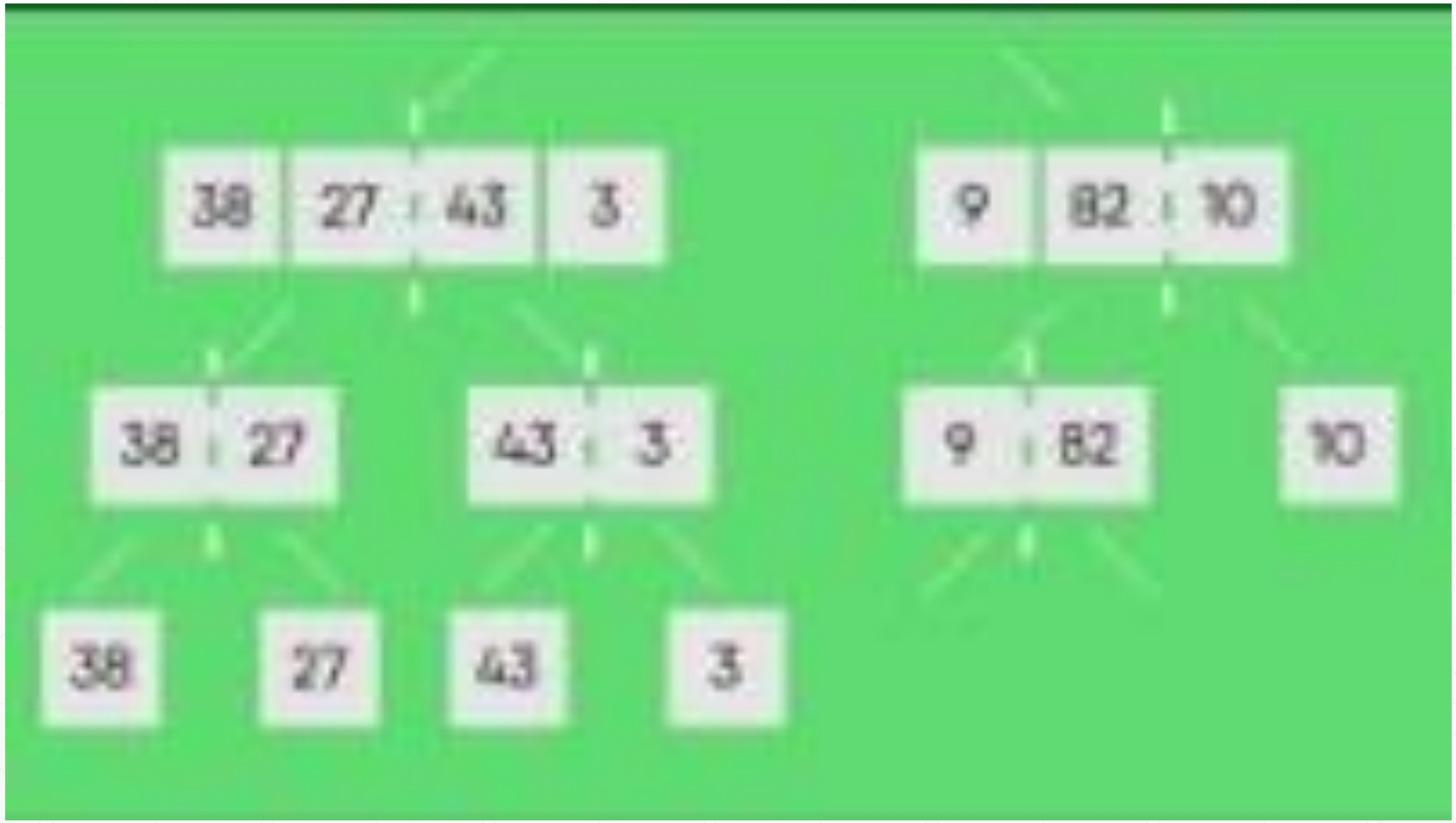
3. Invoque mergeSort para a segunda metade :

   Call mergeSort(arr, m+1, r)

4. Una as duas metades ordenadas no passo 2 e 3:

   Call merge(arr, l, m, r)

Fixando o exemplo



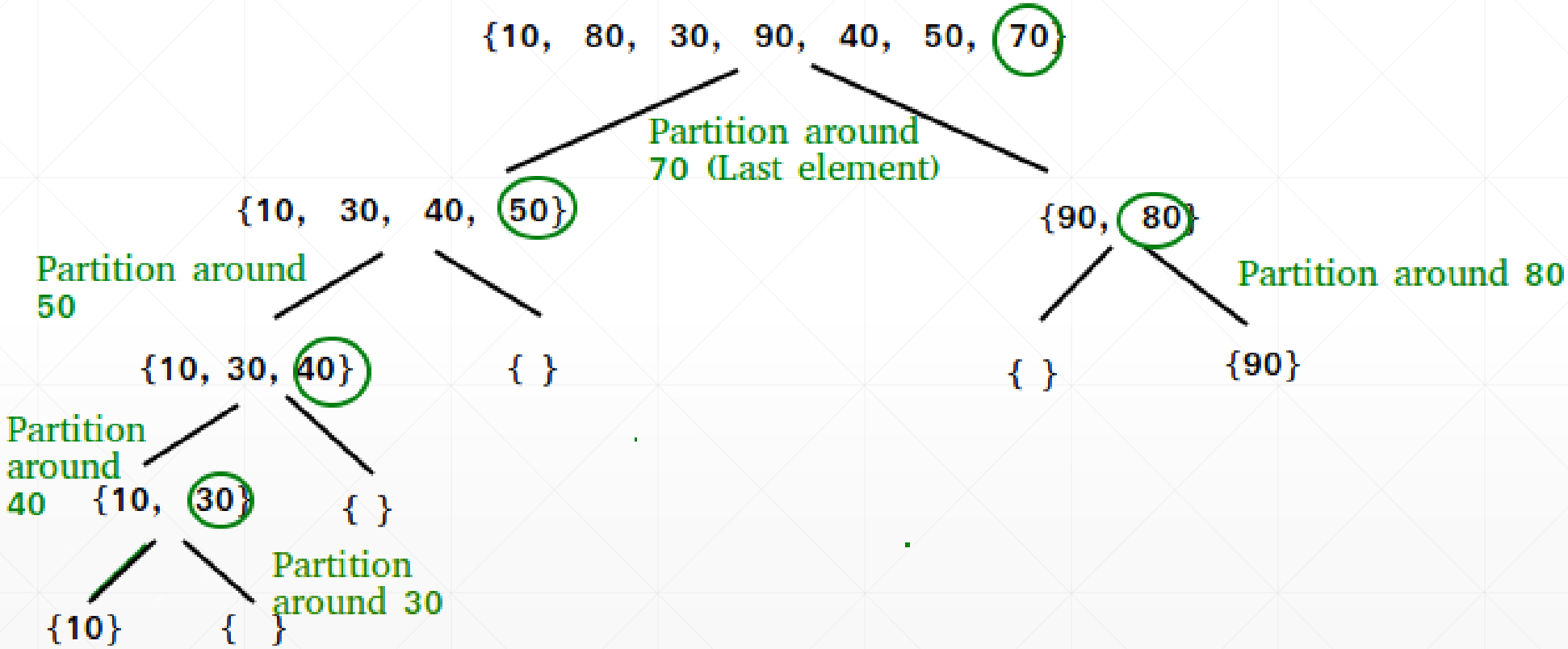
Merge sort: características

- Útil para ordenar listas encadeadas em O (n log n)
- Desvantagens:
  - Requer memória adicional para realizar o merge
  - O processo continua mesmo quando o vetor está ordenado
  - Pode ser mais lento que outros algoritmos de ordenação para tarefas pequenas

Quick sort

- Assim como o merge sort, também trabalha com a abordagem de divisão e conquista
- Esse algoritmo seleciona um pivô e divide o array a partir da posição deste pivô
  - Um grupo com os elementos menores do que o pivô à esquerda e os maiores à direita
  - A divisão do array é um processo chave neste algoritmo

Exemplo: Quick sort

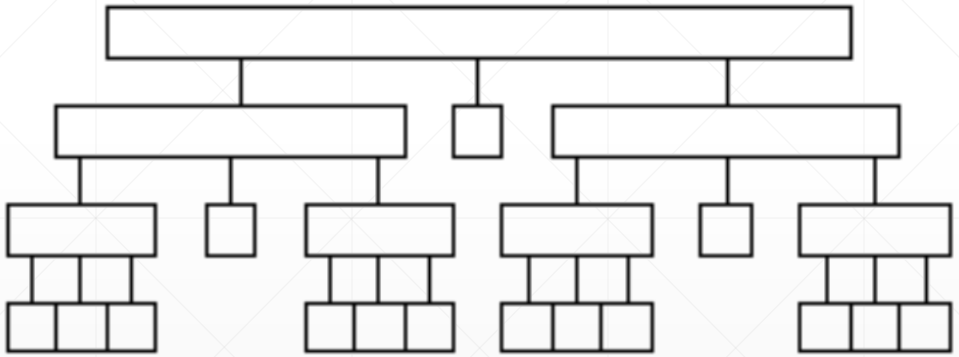


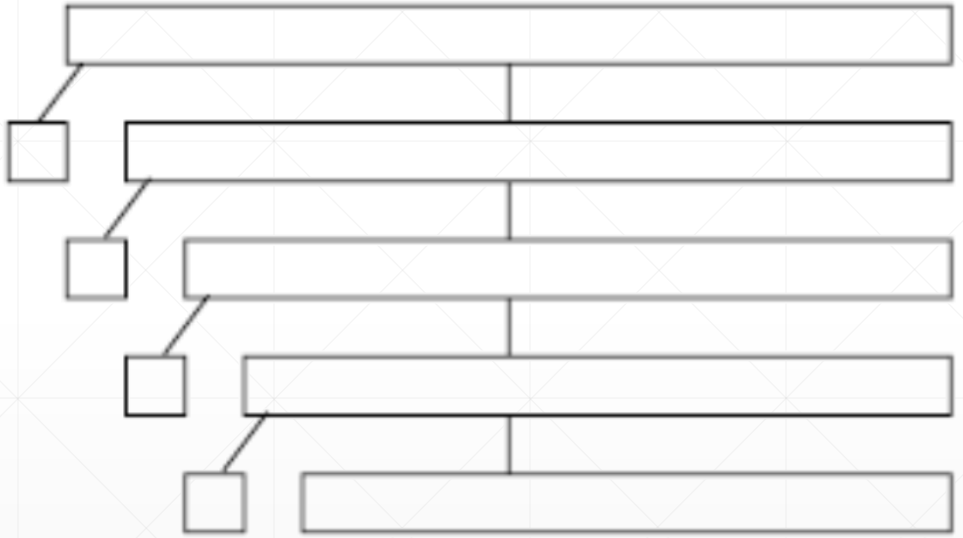
Fixando o exemplo: Quick sort

Selecionando o pivô

- O pivô pode ser selecionado de diferentes maneiras:
  - Sempre o primeiro elemento
  - Sempre o último elemento
  - Selecionar um elemento aleatório
  - Utilizar a mediana como pivô
- Skiena (2008), demonstra que selecionar o pivô aleatoriamente garante uma alta probabilidade do algoritmo executar em O (n log n)
  - Método mais utilizado

Melhor caso vs pior caso





Selecionando o pivô

- Utilizar sempre o último elemento como pivô gera o pior caso → $O(n^2)$
  - Sempre irá ocorrer se o vetor já estiver ordenado
    - Escolher o pivô aleatoriamente evita isto

Características: Quick sort

- O quick sort depende bastante da implementação e da sua aplicação
- Entre os algoritmos O(n log n) o quick sort é considerado o mais rápido
  - 2 a 3 vezes mais rápido do que heapsort e mergesort
- Algoritmo eficiente para ordenar arrays

Algoritmo Linear

Counting sort

- Realiza a contagem dos valores no conjunto a ser ordenado
- Em seguida realiza operações aritméticas para ordenar os valores
- Counting sort é O(n)

Exemplo: Counting sort

Características: Counting sort

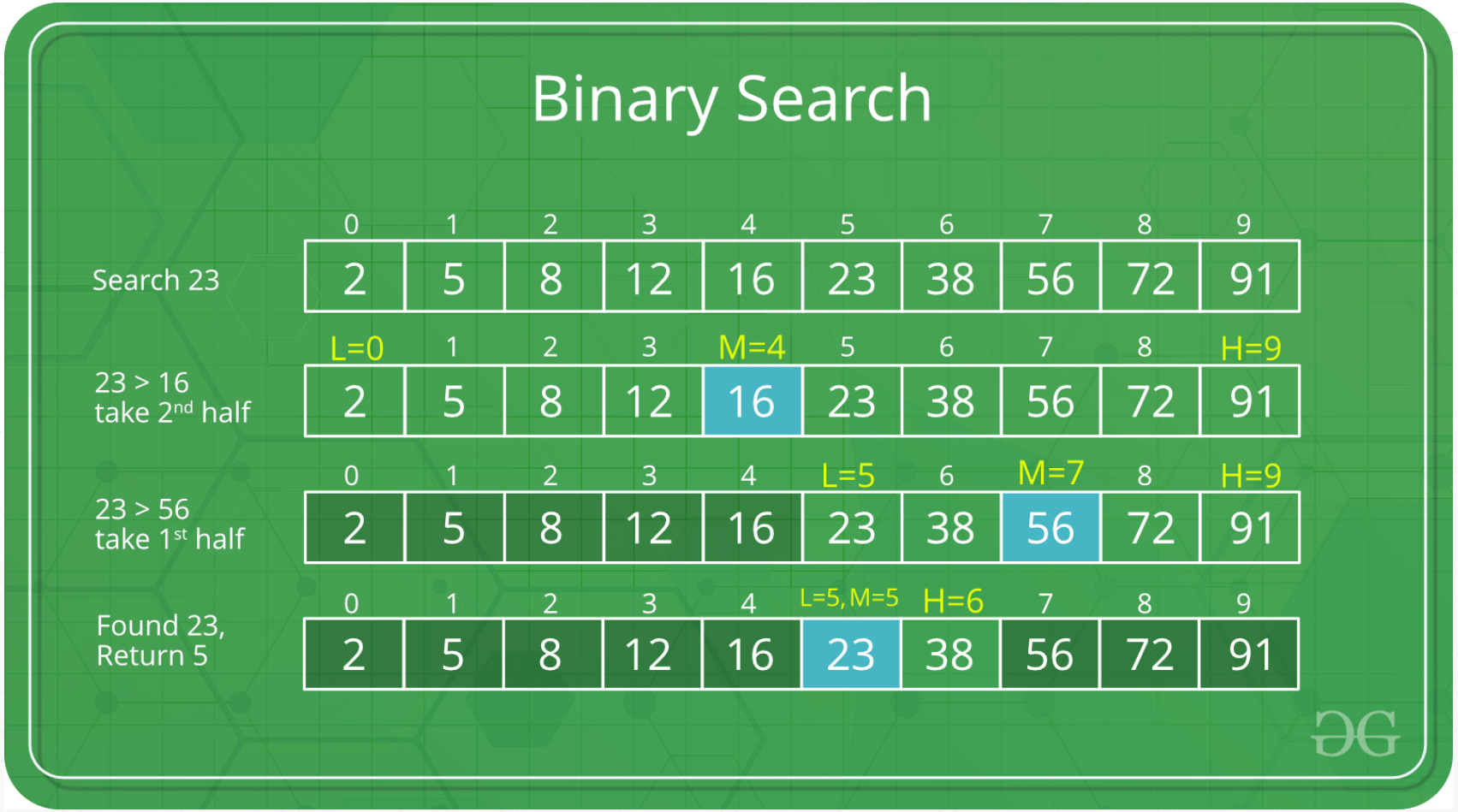
- É necessário fazer adaptações para ordenar números negativos
  - É preciso identificar o menor valor do conjunto e associar ele ao índice zero do array de contagem
- É eficiente quando o intervalo de valores a ser ordenado não é maior do que o número de itens
  - 10, 5, 10 000, 5 000
- **Não utiliza comparações para ordenar**

Busca em array ordenado

Busca binária – Binary search

- É um algoritmo de busca rápido para arrays ordenados
- Para uma palavra-chave de busca q, comparamos q com o valor do meio do array
  - Se o valor mediano for menor do que a chave, iremos olhar para o lado direito do array
  - Caso contrário olhamos para o lado esquerdo
- É executado em O(log n)
  - Bem mais rápido que o custo de uma busca linear O(n)

Exemplo: busca binária



Escolhendo o elemento do meio

- Poderíamos usar para encontrar o elemento do meio a equação:

$$int\ mid = (low + high)/2;$$

  - Mas isso pode causar estouro de memória

- No exemplo, estamos usando:

$$int\ mid = low + (high - low)/2;$$

Referências

- Rocha-junior, J. B. Força Bruta: ordenação. Notas de aula. Disponível em: https://bit.ly/2YvYWMJ. Acessado em: 09/11/2021
- CORMEN, Thomas. Desmistificando Algoritmos. Editora Campus, 2012.
- SKIENA, Steven. The Algorithm Design Manual. 6ª edição. Springer, 2020.

Dúvidas?