Algoritmos de ordenação – divisão e conquista

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

Licenciamento

Sumário

Divisão e conquista

Divida um problema grande, de forma recursiva, em problemas menores até que possa ser resolvido de forma trivial

1. **Divisão**
   - Problema maior é dividido em subproblemas menores e similares ao original
2. **Conquista**
   - Quando problema é pequeno o suficiente, a solução do problema é calculada
3. **Combinação**
   - Resultados dos problemas menores são combinados até chegar na solução do problema original

Algoritmo *merge sort*

Merge sort

Análise

- O algoritmo é $\Theta(n \log n)$, ou seja, melhor, pior e caso médio = $n \log n$
  - A ordenação será eficiente independente da disposição dos elementos
- Algoritmo recursivo baseado na abordagem de **divisão e conquista**
  - Consumo adicional da memória a cada chamada recursiva
  - O consumo de memória auxiliar é proporcional ao tamanho do problema (complexidade de espaço $= O(n)$)
- É um algoritmo **estável**

Merge sort

Algoritmo

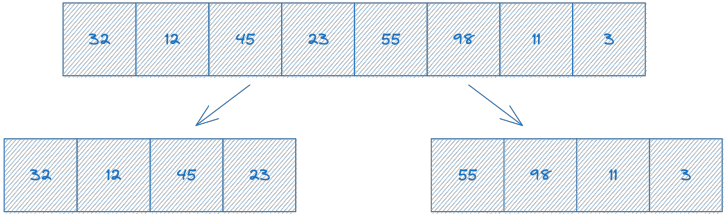
1. **Divisão** – $\Theta(1)$
   - Divida recursivamente, ao meio, em dois subvetores até ter apenas um elemento
2. **Conquista** – $2\frac{n}{2}$
   - Recursivamente, ordene cada um dos subvetores de tamanho $\frac{n}{2}$
3. **Combinação** – $\Theta(n)$
   - Combine (*merge*) os subvetores em um vetor ordenado

Merge sort

Divisão

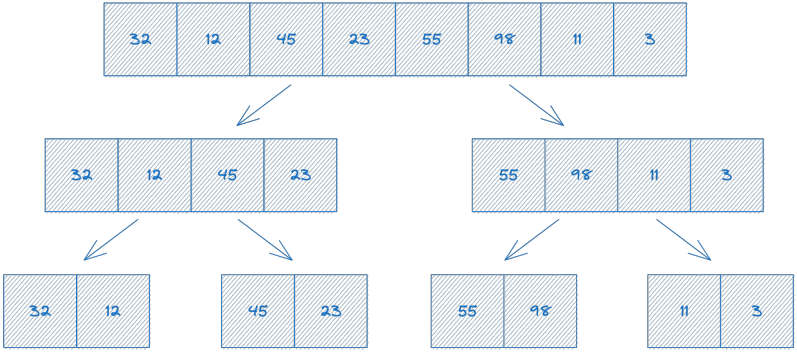
Merge sort

Divisão



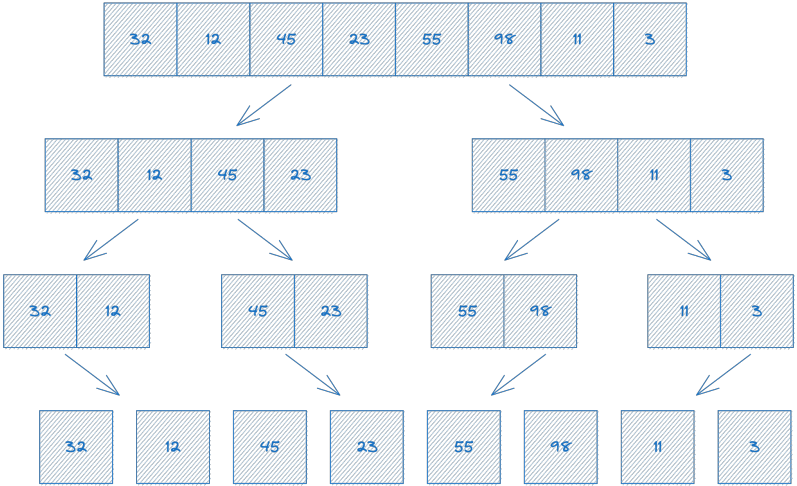
Merge sort

Divisão



Merge sort

Divisão



Merge sort

Conquista

Merge sort

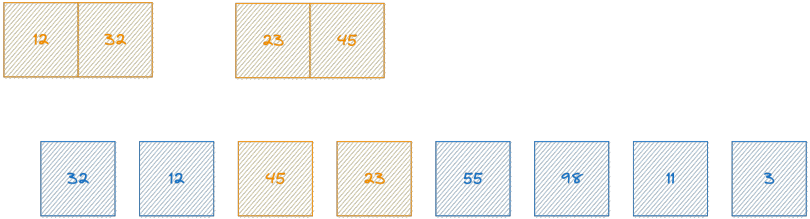
Conquista

Merge sort

Conquista

Merge sort

Conquista



Merge sort

Conquista

Merge sort

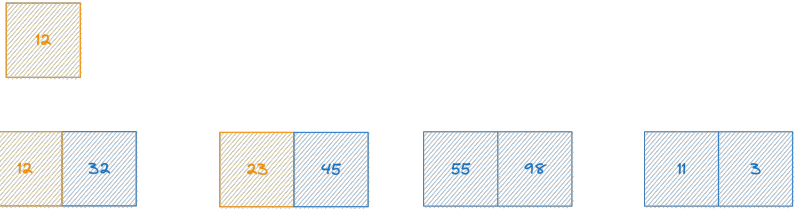
Conquista

Merge sort

Conquista

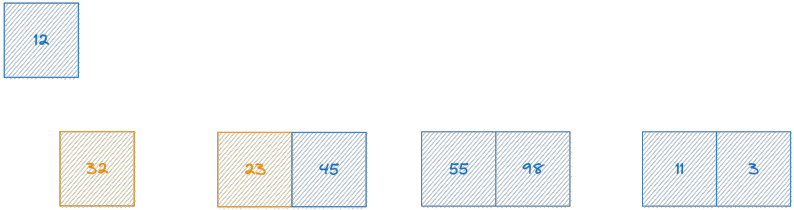
Merge sort

Conquista



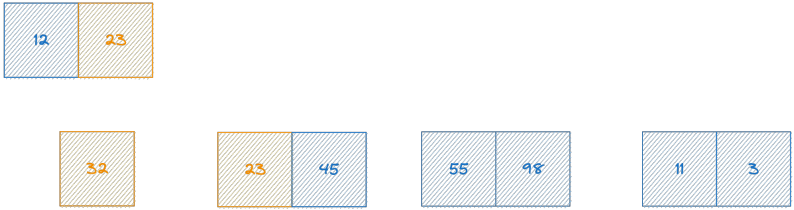
Merge sort

Conquista



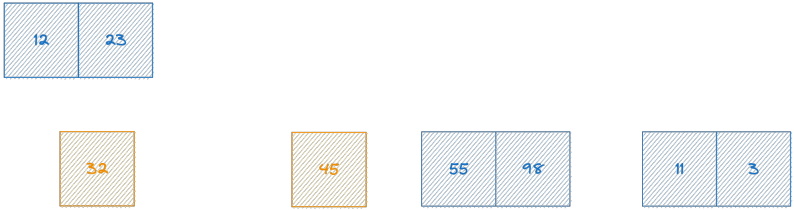
Merge sort

Conquista



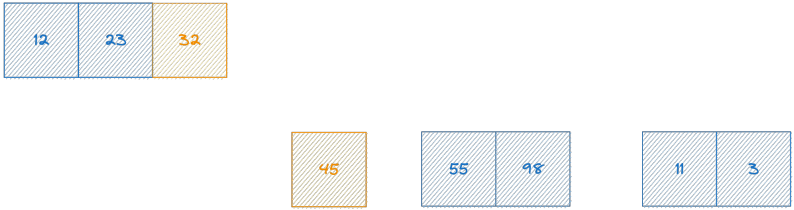
Merge sort

Conquista



Merge sort

Conquista



Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Conquista

Merge sort

Pseudocódigo

Algoritmo 1: Função *merge sort*

```
// Em C esquerda começa com 0 e direita com n − 1
1 função merge_sort(vetor[1..n], esquerda, direita):
2     se esquerda < direita então
3         meio ← esquerda + (direita − esquerda)/2;                    // divisão
          // ordene as duas metades (conquista)
4         merge_sort(vetor, esquerda, meio);
5         merge_sort(vetor, meio + 1, direita);
          // mescle as metades (combinação)
6         merge(vetor, esquerda, meio, direita);
7     fim
8     retorna (vetor);
9 fim
```

Merge sort – Pseudocódigo (*continuação*)

Algoritmo 2: Função *merge*

```
1  função merge(vetor, esquerda, meio, direita):
2      aux[1..(direita − esquerda + 1)];        // criando vetor auxiliar vazio
3      i ← esquerda; j ← (meio + 1); k ← 0;
4      enquanto i ≤ meio E j ≤ direita faça
5          se vetor[i] ≤ vetor[j] então
6              aux[k] ← vetor[i];
7              i ← i + 1;
8          senão
9              aux[k] ← vetor[j];
10             j ← j + 1;
11         fim
12         k ← k + 1;
13     fim
14     enquanto i ≤ meio faça
15         aux[k] ← vetor[i];
16         i ← i + 1; k ← k + 1;
17     fim
18     enquanto j ≤ (direita) faça
19         aux[k] ← vetor[j];
20         j ← j + 1; k ← k + 1;
21     fim
22     para i de esquerda até direita faça
23         vetor[i] = aux[i − esquerda];
24     fim
25 fim
```

Algoritmo *Quicksort*

Algoritmo *Quicksort*

Criado por Charles Hoare em 1960

- Baseia-se no paradigma de dividir para conquistar
- Tempo médio de $O(n \log n)$ o torna a melhor opção prática
  - constantes ocultas na notação são menores do que a dos demais algoritmos
- Tempo de execução no pior caso é $\Theta(n^2)$
  - Algumas estratégias minimizam a chance de ocorrer o pior caso
- Melhor que o *Merge sort* em termos de **complexidade em espaço**
  - $O(n \log n)$ Quicksort
  - $O(n)$ Mergesort - vetor temporário de tamanho n
- É um algoritmo **instável** (não preserva a ordem inicial)
- A biblioteca `stdlib.h` provê a função `qsort`[1]

[1] https://cplusplus.com/reference/cstdlib/qsort

Algoritmo *Quicksort*

Passos

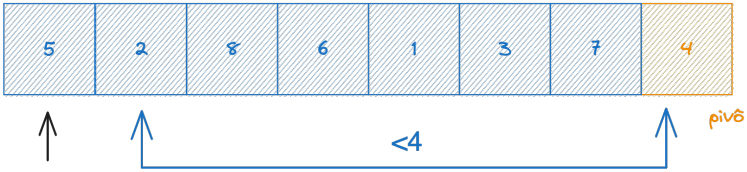
1. Escolha um elemento do arranjo, denominado **pivô**
   - Pode ser o primeiro, que está no metade, o último ou escolhido de forma aleatória
2. Particione
   - Reorganize a lista de tal forma que todos os elementos à esquerda do **pivô** seja menores ou iguais a ele, e todos elementos à direita sejam maiores
     - Compare o pivô com todos os elementos, a partir do primeiro
   - Ao término, o **pivô** estará em sua posição final e haverá dois subarranjos (esquerda e direita), ou seja, foi realizado o particionamento
   - Os dois subarranjos (esquerda e direta) não precisam estar ordenados
3. Recursivamente ordene os dois subarranjos usando Quicksort

Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare

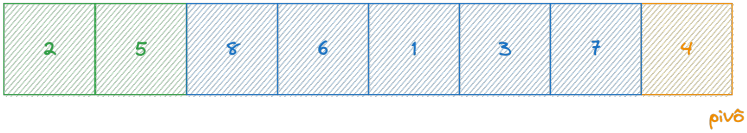
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



Quicksort

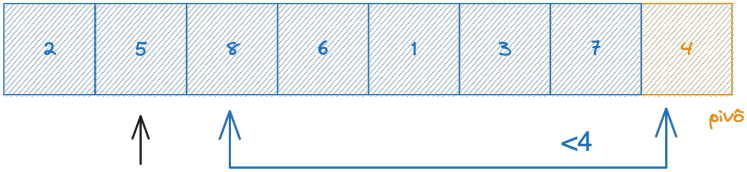
Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



troca

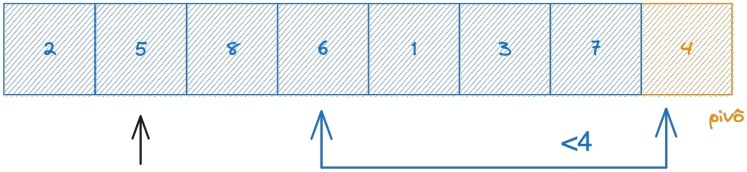
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



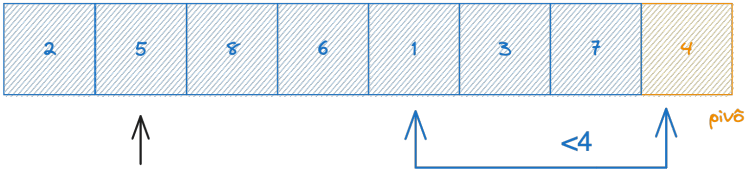
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



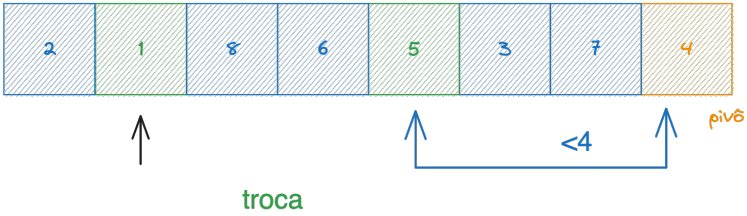
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



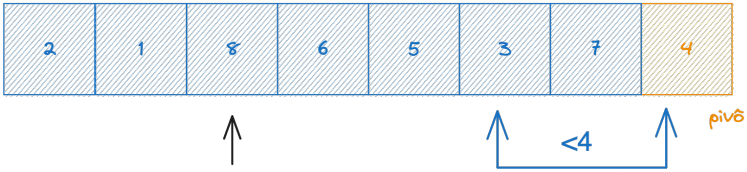
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



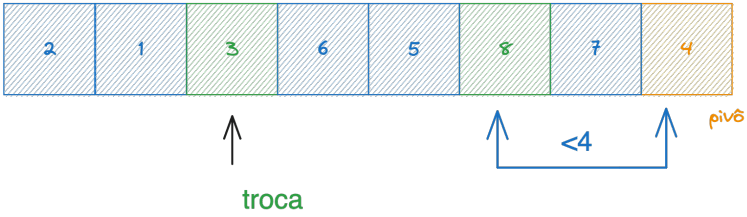
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



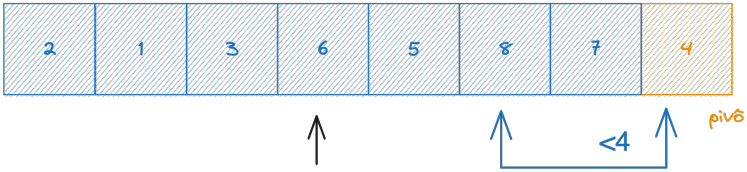
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



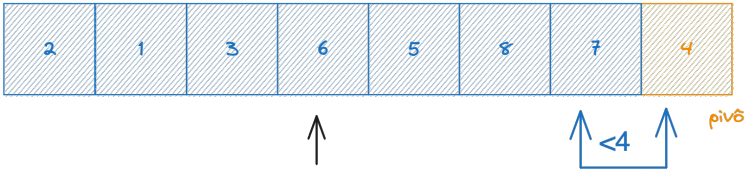
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



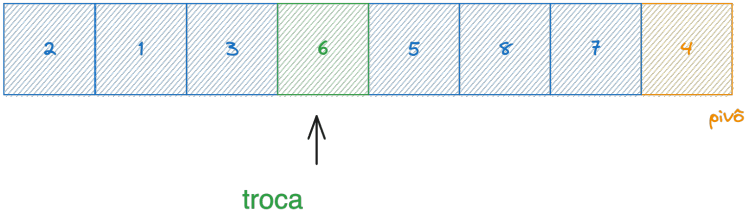
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



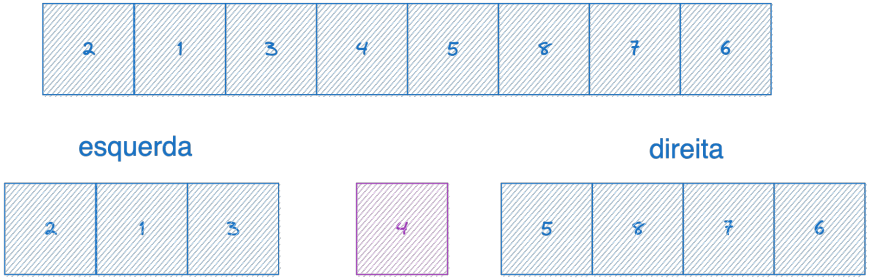
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare

troca

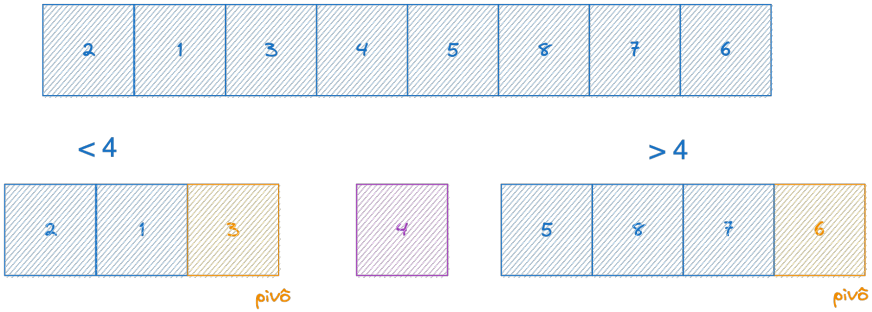
Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



Quicksort

Método de partição Lomuto, mais simples porém menos eficiente que o método Hoare



- E assim continua de forma recursiva...

Algoritmo *Quicksort* – ordenação rápida

Pseudocódigo

Algoritmo 3: Função *quicksort*

```
1 função quicksort(vetor[1..n], inicio, fim):
2     se inicio < fim então
3         p ← particiona(vetor, inicio, fim);
4         quicksort(vetor, inicio, p − 1);
5         quicksort(vetor, p + 1, fim);
6     fim
7     retorna (vetor);
8 fim
```

- A primeira execução considera todos elementos do arranjo
  - $inicio \leftarrow 1$ e $fim \leftarrow n$

Algoritmo *Quicksort* – ordenação rápida

Pseudocódigo (*continuação*)

Algoritmo 4: Função particiona

```
1  função particiona(vetor, inicio, fim):
2      pivo ← vetor[fim];
3      i ← inicio − 1;
4      para j de inicio até fim − 1 faça
5          se vetor[j] ≤ pivo então
6              i ← i + 1;
7              trocar_posicao(vetor[i], vetor[j]);
8          fim
9      fim
10     i ← i + 1;
11     trocar_posicao(vetor[i], vetor[fim]);
12     retorna (i);
13 fim
```

Resumo

- Nenhum algoritmo de ordenação por comparação tem desempenho melhor que $n \log n$ para complexidade em tempo, logo todos são $\Omega(n \log n)$.

Algoritmo	Tempo		
	Melhor	Médio	Pior
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Curiosidades

- **Comparação interativa de algoritmos de ordenação**
  - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
  - https://www.toptal.com/developers/sorting-algorithms
- **Quinze algoritmos de ordenação em 6 minutos**
  - https://www.youtube.com/watch?v=kPRA0W1kECg
- **Danças folclóricas**
  - *merge sort* – https://www.youtube.com/watch?v=dENca26N6V4
  - *quicksort* – https://www.youtube.com/watch?v=3San3uKKHgg

Exercício 1

- Evolua a biblioteca `libprg`, criada por você na aula de lista de sequencial, para ofertar diferentes funções para ordenar um vetor de inteiros por meio dos algoritmos
  a. Merge sort
  b. Quicksort
- As definições das funções devem ser feitas obrigatoriamente no arquivo de cabeçalho `libprg.h`
- As implementações das funções obrigatoriamente no arquivo com o nome `alg_ord_div.c`

Exercício 2

- Evolua o projeto CMake que você fez para o exercício 2 da aula de algoritmos de ordenação troca e seleção.
  - Inclua o uso dos algoritmos *merge sort* e *Quicksort*
- Deve-se imprimir na tela o tempo gasto (relógio de parede e de CPU) por todos algoritmos de ordenação que foram implementados na biblioteca `libprg`