BCC202 - Estruturas de Dados I

Aula 15: Ordenação: QuickSort

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

2021

Conteúdo

Conteúdo

Visão Geral

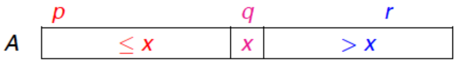
- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- O algoritmo *QUICKSORT* segue o paradigma de divisão-e-conquista.

Ordenar um subarranjo típico $A[l..r]$ (Cormen et al., 2011).

- Divisão: Particionar o arranjo $A[l..r]$
  - $A[l..q-1]$
  - $A[q+1..r]$
  - cada elemento de $A[l..q-1]$ é menor ou igual a $A[q]$
  - $A[q]$ é menor ou igual a cada elemento de $A[q+1..r]$
  - Calcular o índice q como parte desse procedimento de particionamento
- Conquista: Ordenar os dois subarranjos $A[l..q-1]$ e $A[q+1..r]$ por **chamadas recursivas a *QuickSort***.
- Combinação: Como os subarranjos já estão ordenados, não é necessário nenhum trabalho para combiná-los: o arranjo $A[p..r]$ inteiro agora está ordenado.

Divisão e Conquista

O que podemos afirmar sobre o elemento que está na posição q, o pivô?



$$A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$$

Partição em subproblemas

- A parte mais delicada do método é o processo de **partição**.
- O vetor `v[esq...dir]` é rearranjado por meio da escolha arbitrária de um **pivô** `x`.
- O vetor `v` é particionado em duas partes:
  - Parte **esquerda**: $\boldsymbol{chaves} <= \boldsymbol{x}$.
  - Parte **direita**: $\boldsymbol{chaves} >= \boldsymbol{x}$.

Partição em subproblemas

- **Algoritmo para o particionamento**:
  1. Escolha arbitrariamente um pivô `x`.
  2. Percorra o vetor a partir da esquerda até que `v[i] >= x`.
  3. Percorra o vetor a partir da direita até que `v[j] <= x`.
  4. Troque `v[i]` com `v[j]`.
  5. Continue este processo até os apontadores `i` e `j` se cruzarem.

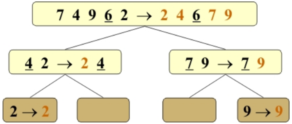
Partição em subproblemas

- Concluído o particionamento, o vetor `v[esq..dir]` está particionado de tal forma que:
  - Os itens em `v[esq]`, `v[esq + 1]`, `...`, `v[j]` são menores ou iguais a `x`.
  - Os itens em `v[i]`, `v[i + 1]`, `...`, `v[dir]` são maiores ou iguais a `x`.
- Pode-se concluir então que o pivô `x` encontra-se na posição correta de ordenação.

Conteúdo

Árvore Binária

- A execução do **QuickSort** pode ser facilmente descrita por uma árvore binária:
  - Cada nó representa uma chamada recursiva do **QuickSort**.
  - O nó raiz é a chamada inicial.
  - Os nós folha são vetores de tamanho 0 ou 1 (casos base).



A seguir, mais detalhes sobre o passo a passo do *quicksort*.

Animação Vídeo

Partição

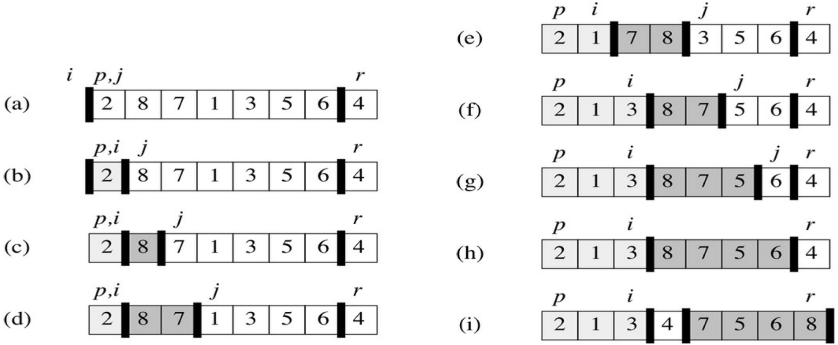


Figura: QuickSort: operação de *partição* (Cormen et al., 2011)

Conteúdo

Pseudocódigo: QuickSort

A seguir, o pseudocódigo do *QuickSort*.

```
Algorithm: QUICKSORT
Input: int* v, int l, int r // l = Left (esquerda); r = Righty (direita)
begin
    if l < r then
        q ← PARTITION(v, l, r)
        QUICKSORT(v, l, q − 1)
        QUICKSORT(v, q + 1, r)
    end
end
```

Pseudocódigo: Partition

A seguir, o pseudocódigo do procedimento de *partição*.

```
Algorithm: PARTITION
Input: int* v, int l, int r
Output: int // índice q
begin
    x ← v[r] // x é o pivô
    i ← l − 1
    for j ← l to  j < r do
        if A[j] ≤ x then
            i ← i + 1
            trocar A[i] com A[j]
        end
    end
    trocar A[i + 1] com A[r]
    // Pivô está na posição r  que vai ser trocado com o elemento da posição i + 1
    return i + 1 // Complexidade:  O(n) = n ← r − l + 1
end
```

Tempo, Comparação e Espaço

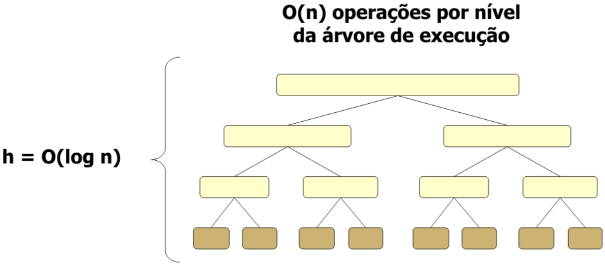
QuickSort é um algoritmo não estável.

- Complexidade de tempo no pior caso
  - $O(n^2)$ comparações
- Complexidade de tempo no melhor caso
  - $O(n \ logn)$ comparações
  - $O(n) + 0 = O(n)$
- Complexidade de espaço/consumo de espaço
  - Extra: $O(logn)$

Melhor Caso

$$C(n) = 2 * C(n/2) + n = n\ log\ n = O(n\ log\ n)$$

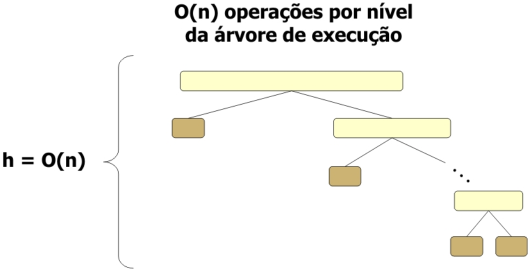
- Ocorre quando o problema é sempre dividido em subproblemas de igual tamanho após a partição.



Pior Caso

$$C(n) = O(n^2)$$

- Ocorre quando, sistematicamente, o **pivô** é escolhido como sendo um dos extremos de um arquivo já ordenado.



Pior Caso

- O pior caso pode ser evitado empregando pequenas modificações no algoritmo. Algumas opções:
  - Escolher o pivô aleatoriamente.
  - Escolher três itens quaisquer do vetor e usar a mediana dos três como pivô.
  - "**Embaralhar**" o vetor original antes de iniciar a ordenação. Um bom algoritmo é o de **Fischer-Yates** ($\boldsymbol{O(n)}$):

```
n = tamanhoDoVetor
para cada i entre n e 2
sorteie j como um número entre 1 e i
se i e j forem diferentes, troque os elementos i e j entre si

```

Caso Médio

$$C(n) \approx 1.386n\ log\ n - 0,846n = O(n\ log\ n)$$

- Análise por **Sedgewick e Flajolet (1996, p. 17)**.
- A proporção das divisões não será sempre constante.
- Ocorre quando há uma mistura de divisões boas e ruins.
- Perceba que o caso médio está muito mais próximo do melhor caso do que do pior caso.

QuickSort RECURSIVO: Características

Vantagens

- É extremamente eficiente para ordenar arquivos.
- Requer apenas uma pequena pilha como memória auxiliar.
- Requer $\boldsymbol{O(n\ log\ n)}$ comparações em média (caso médio) para ordenar $\boldsymbol{n}$ itens.

Desvantagens

- Tem um pior caso com $\boldsymbol{O(n^2)}$ comparações.
- Implementação delicada e difícil: um pequeno engano pode levar a efeitos inesperados.
- O método **não é estável**.

Algorithm: QUICKSORT-ITERATIVO

Input: int* v, int p, int r

begin

alocar $pilha\_l[0...(r-l+1)]$ e $pilha\_r[0...((r-l+1))]$

$pilha\_l[0] \leftarrow l$ $pilha\_r[0] \leftarrow r$ $t \leftarrow 0$

while $t \geq 0$ **do**

$l \leftarrow pilha\_l[t]$

$r \leftarrow pilha\_r[t]$

$--t$

if $l < r$ **then**

$j \leftarrow PARTITION(v, l, r)$

$++t$

$pilha\_l[t] \leftarrow l$

$pilha\_r[t] \leftarrow j-1$

$++t$

$pilha\_l[t] \leftarrow j+1$

$pilha\_r[t] \leftarrow r$

end

end

desalocar $pilha\_l[0...(r-l+1)]$ e $pilha\_r[0...((r-l+1))]$

end

Pilha de Recursão v.s. Pilha Explícita

- O que é colocado em cada uma das pilhas?
- Qual intervalo do vetor é empilhado em cada caso?

Conteúdo

- **QuickSort**: divisão e conquista.
- Duas implementações: **Recursiva** e **Iterativa**.
- Sobre a complexidade do Quicksort.

Quadro Comparativo dos métodos de ordenação

Algoritmo	Comparações			Movimentações			Espaço	Estável	In situ
	Melhor	Médio	Pior	Melhor	Médio	Pior			
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$	**Sim**	**Sim**
Selection	$O(n^2)$			$O(n)$			$O(1)$	**Não***	**Sim**
Insertion	$O(n)$	$O(n^2)$		$O(1)$	$O(n^2)$		$O(1)$	**Sim**	**Sim**
Merge	$O(n \log n)$			$O(n \log n)$			$O(n)$	**Sim**	**Não**
Quick	$O(n \log n)$		$O(n^2)$	–			$O(n)$	**Não***	**Sim**

* Existem versões estáveis.

Na próxima aula

ShellSort.

Exercício 01

- Dada a sequência de números: `3 4 9 2 5 1 8`.
- Ordene em ordem crescente utilizando o algoritmo aprendido em sala (**QuickSort**), apresentando a sequência dos números a cada passo (Teste de Mesa).

Conteúdo

Bibliografia

Os conteúdos deste material, incluindo figuras, textos e códigos, foram extraídos ou adaptados de:

- *Slides* **MO417 - Complexidade de Algoritmos I**, elaborados por Cid Carvalho de Souza, Cândida Nues da Silva e Orlando Lee e revisado por Zanoni Dias em agosto de 2011, https://www.ic.unicamp.br/~zanoni/teaching/mo417/2011-2s/aulas/handout/04-ordenacao.pdf. Acessado em 2021.

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford.
Introduction to Algorithms.
The MIT Press, 2011.

Exercício

- Dada a sequência de números: `3 4 9 2 5 1 8`.
- Ordene em ordem crescente utilizando o algoritmo aprendido em sala (**QuickSort**), apresentando a sequência dos números a cada passo (Teste de Mesa).