BCC202 - Estrutura de Dados I

Aula 14: Ordenação: QuickSort

Reinaldo Fortes

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM

Website: **www.decom.ufop.br/reifortes**
Email: **reifortes@iceb.ufop.br**

Material elaborado com base nos slides do Prof. Túlio Toffolo (curso de 2013/01).

2013/02

Conteúdo

Conteúdo

QuickSort

- 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.
- Provavelmente é o mais utilizado.

Ideia Básica (divisão e conquista)

- Dividir o problema de ordenar um conjunto com n itens em dois subproblemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

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**: ***chaves*** $<=$ ***x***.
  - Parte **direita**: ***chaves*** $>=$ ***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.

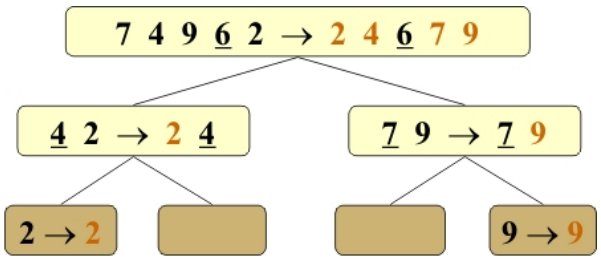
Partição em subproblemas

- 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

Representação por Árvore

- 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 0 ou 1 número (casos bases).



Animação

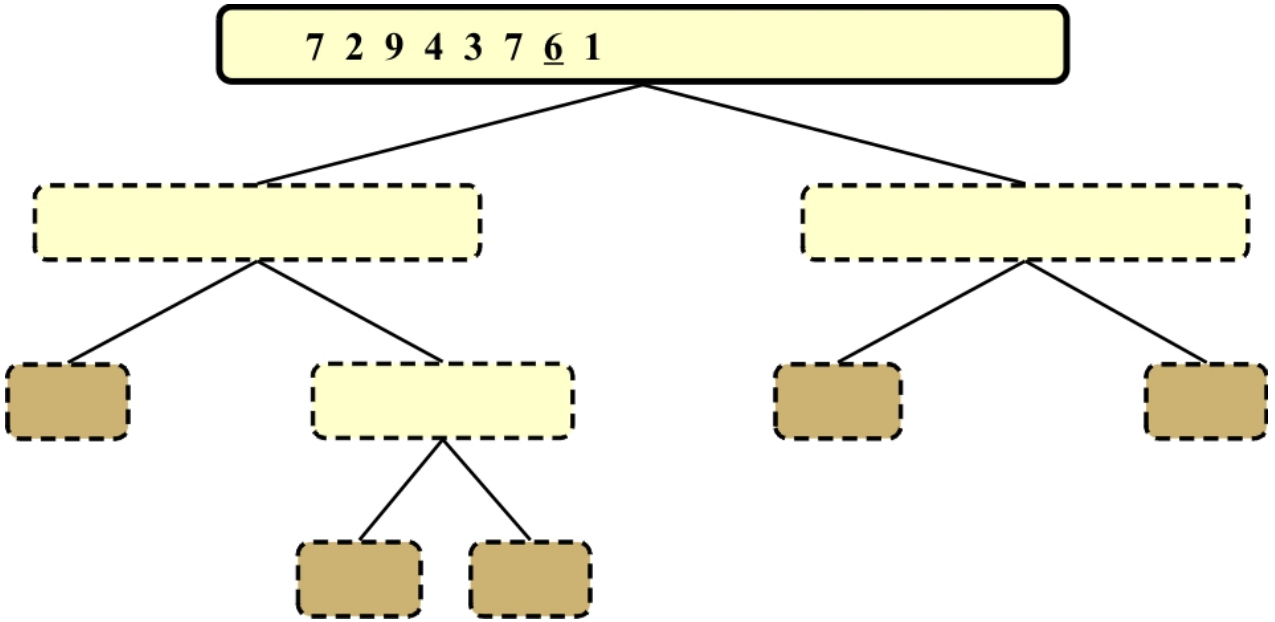
Algoritmo

- Escolha um pivô.
- Realize o particionamento.
- Repita o processo de forma recursiva para cada uma das partes.


Animação

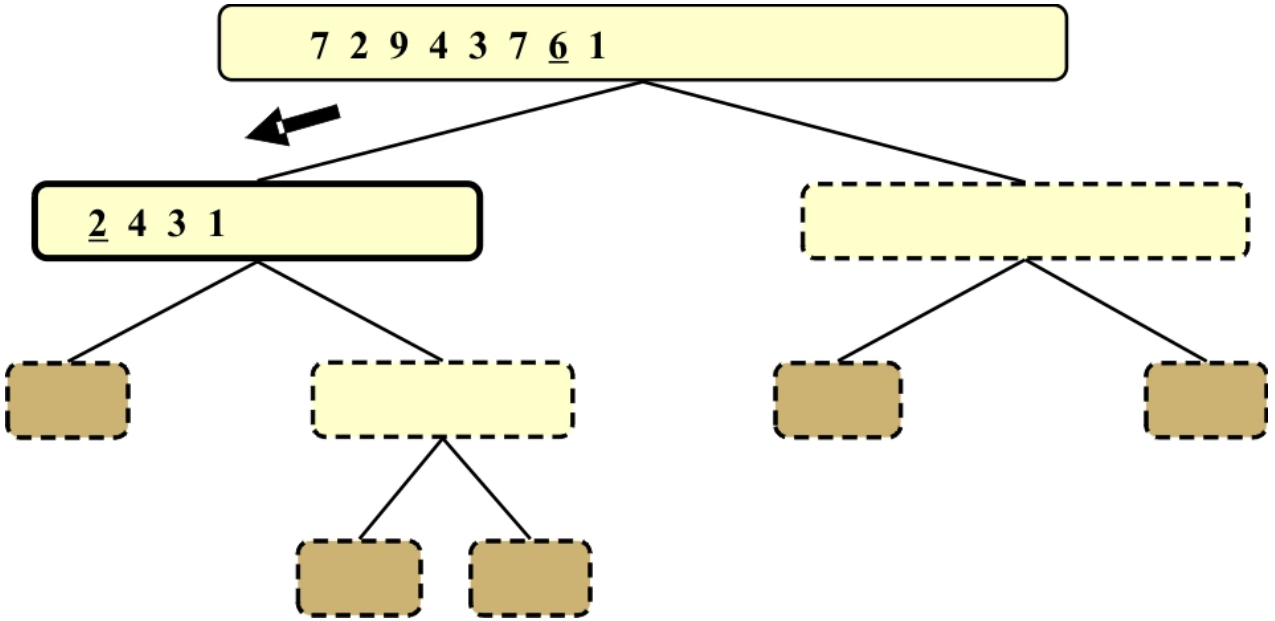

Vídeo

Execução



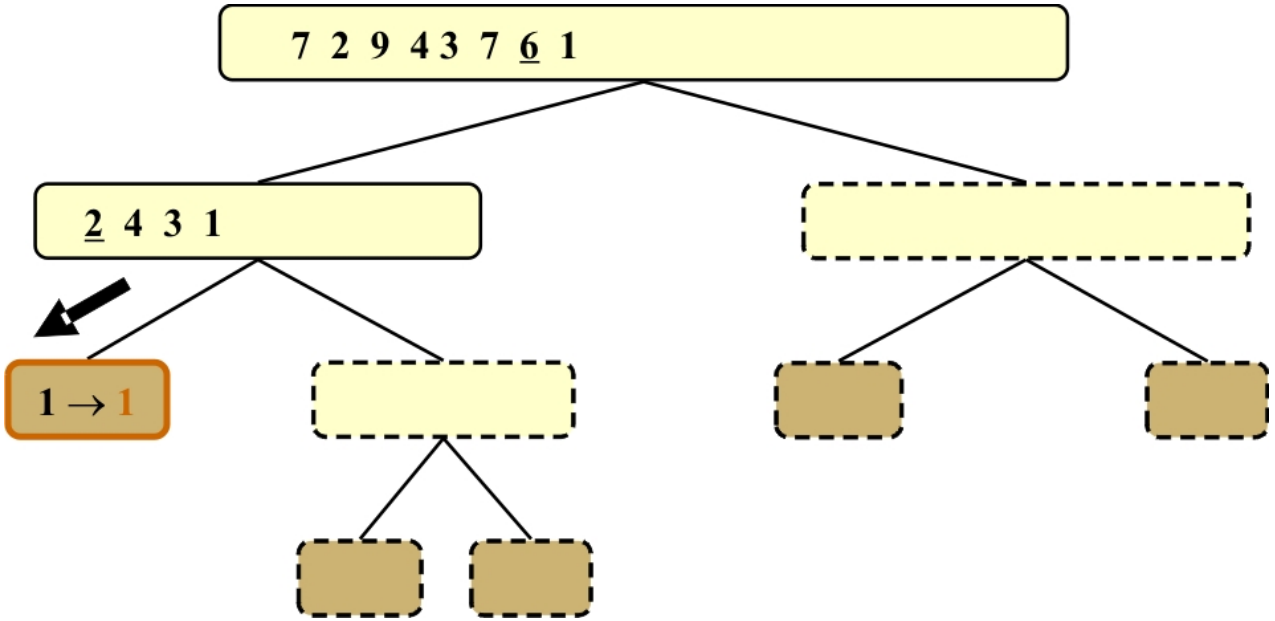
Seleção do pivô.

Execução



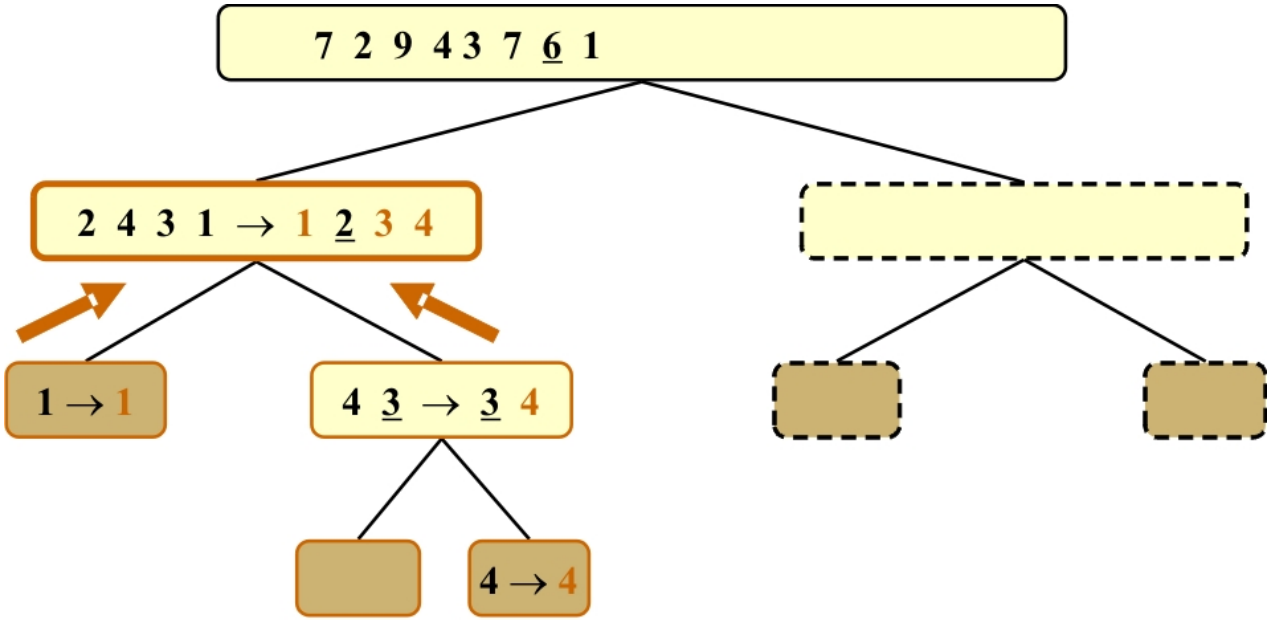
Partição, chamada recursiva e seleção do pivô.

Execução



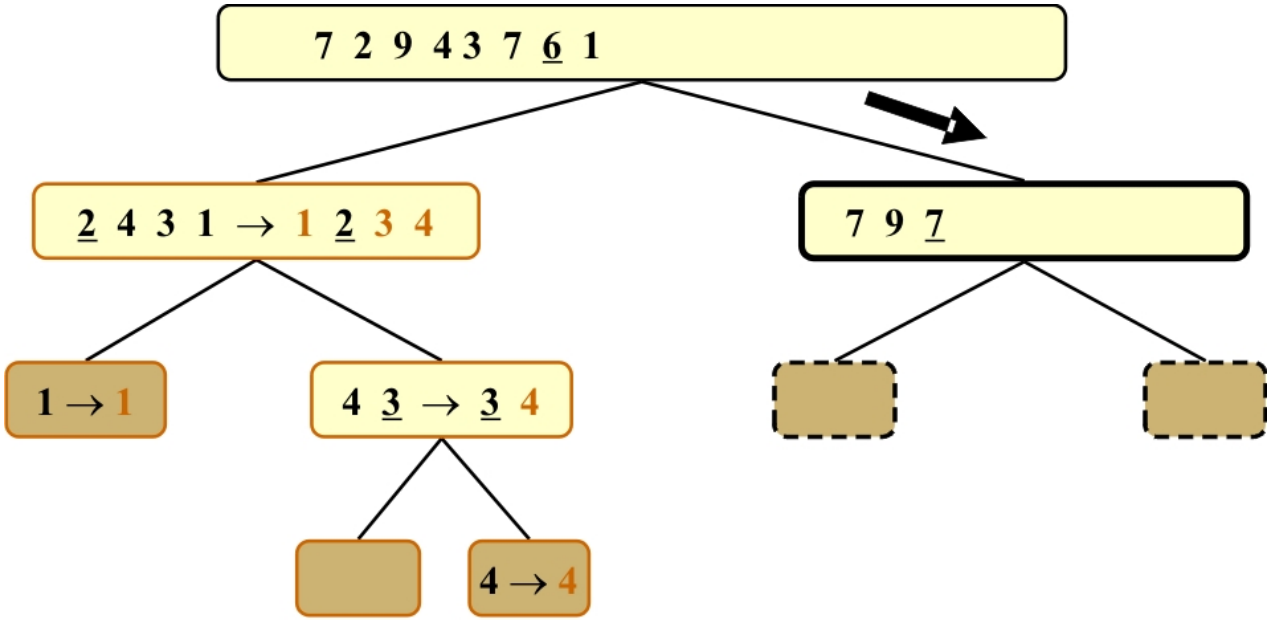
Partição, chamada recursiva e caso base.

Execução



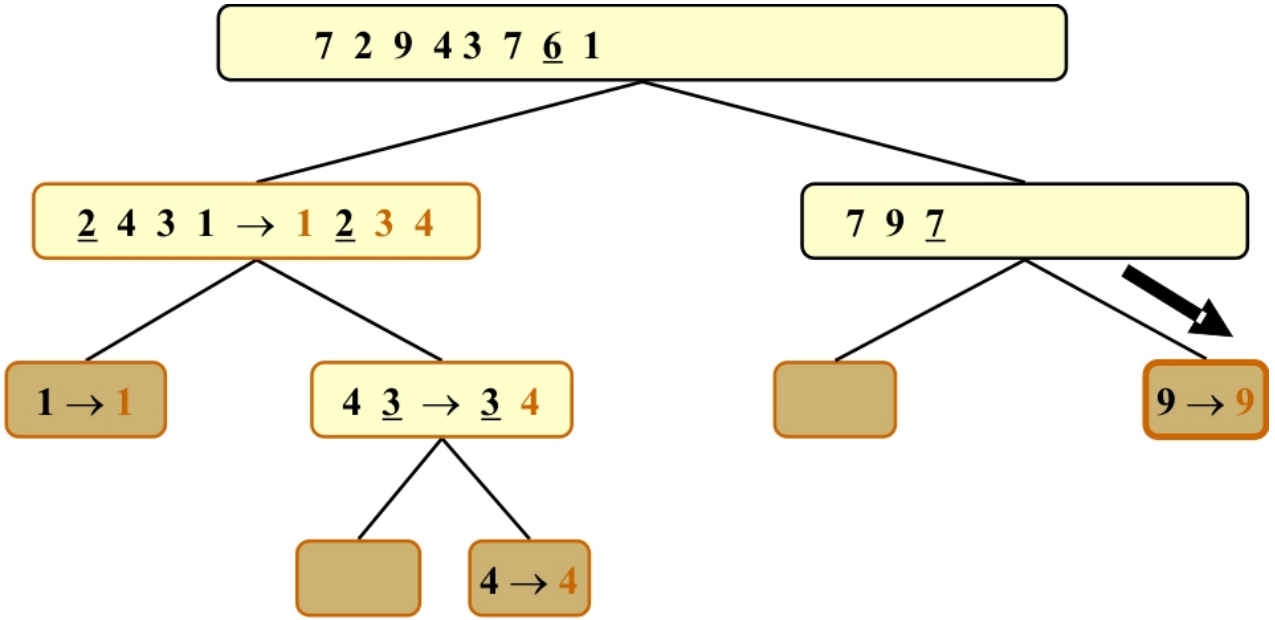
Chamada recursiva, ..., caso base e união.

Execução



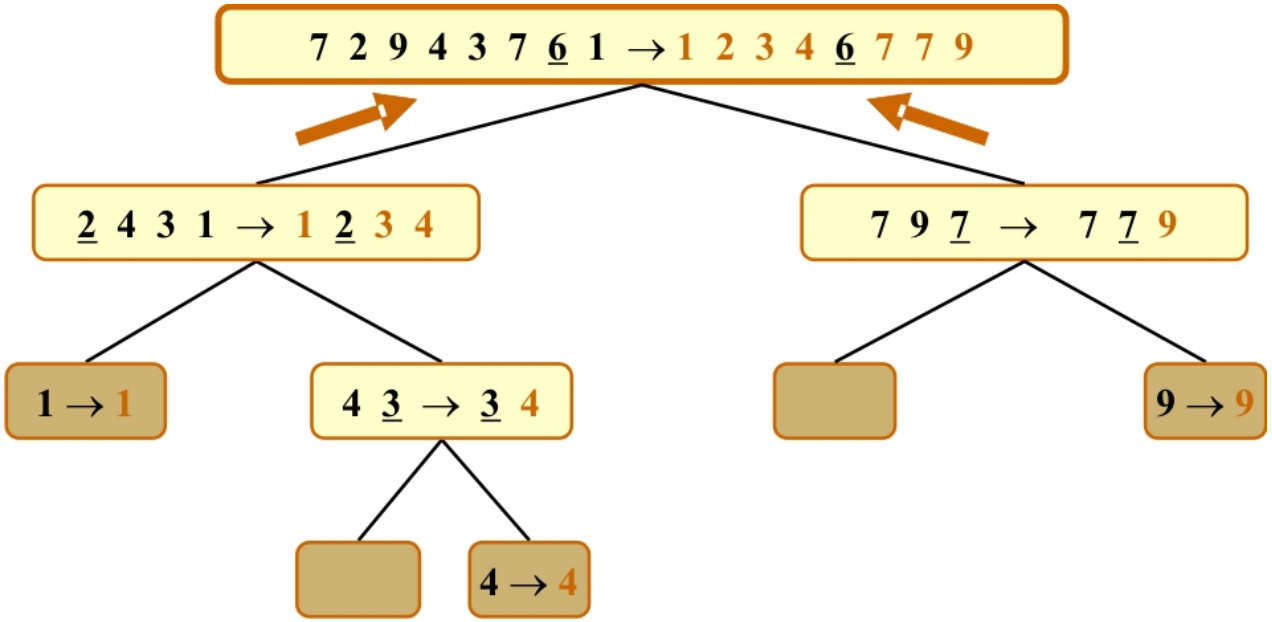
Chamada recursiva e seleção do pivô.

Execução



Partição, ..., chamada recursiva, caso base.

Execução



União.

Conteúdo

QuickSort RECURSIVO

```
1  // Ordena o vetor v[0..n-1]
2  void QuickSort(TItem *v, int n) {
3    QuickSort_ordena(v, 0, n-1);
4  }
5
6  // Ordena o vetor v[esq..dir]
7  void QuickSort_ordena(TItem *v, int esq, int dir) {
8    int i, j;
9    QuickSort_particao(v, esq, dir, &i, &j);
10   if (esq < j) QuickSort_ordena(v, esq, j);
11   if (i < dir) QuickSort_ordena(v, i, dir);
12 }
```

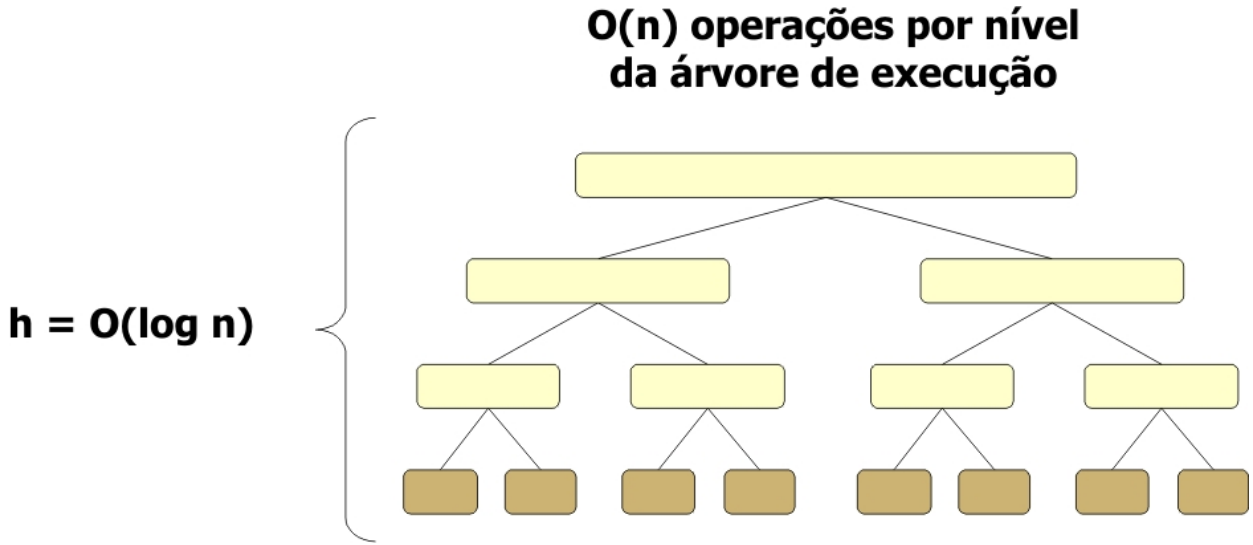
QuickSort RECURSIVO

```
14 void QuickSort_particao(TItem *v, int esq, int dir, int *i,
       int *j) {
15   TItem pivo, aux;
16   *i = esq; *j = dir;
17   pivo = v[(*i + *j)/2]; /* obtem o pivo x */
18   do {
19     while (!(pivo.chave <= v[*i].chave)) (*i)++;
20     while (pivo.chave < v[*j].chave) (*j)--;
21     if (*i <= *j) {
22       aux = v[*i];
23       v[*i] = v[*j];
24       v[*j] = aux;
25       (*i)++; (*j)--;
26     }
27   } while (*i <= *j);
28 }
```

Melhor Caso

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

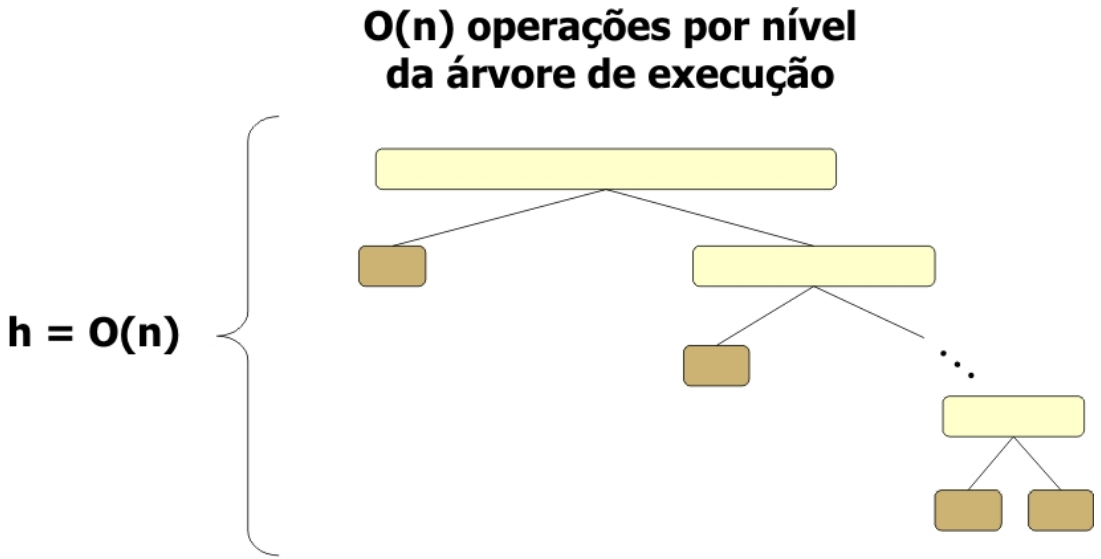
- Ocorre quando o problema é sempre dividido em subproblemas de igual tamanho após a partição.



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** ($O(n)$):

```
1 n = tamanho do vetor
2 para cada i entre n e 2
3   sorteie j como um número entre 1 e i
4   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 $O(n \log n)$ comparações em média (caso médio) para ordenar n itens.

Desvantagens

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

Conteúdo

QuickSort ITERATIVO

```
1  void QuickSort_iter(TItem *v, int n) {
2    TPilha pilha_dir, pilha_esq;
3    int esq, dir, i, j;
4
5    TPilha_Inicia(&pilha_dir);
6    TPilha_Inicia(&pilha_esq);
7    esq = 0;
8    dir = n-1;
9
10   TPilha_Push(&pilha_dir, dir);
11   TPilha_Push(&pilha_esq, esq);
```

QuickSort ITERATIVO

```
13    do {
14      if (dir > esq)  {
15        QuickSort_particao(v, esq, dir, &i, &j);
16        TPilha_Push(&pilha_dir, j);
17        TPilha_Push(&pilha_esq, esq);
18        esq = i;
19      }
20      else {
21        TPilha_Pop(&pilha_dir, &dir);
22        TPilha_Pop(&pilha_esq, &esq);
23      }
24    } while (!TPilha_EhVazia(&pilha_dir));
25  }
```

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?

QuickSort ITERATIVO: Características

Vantagens

- É extremamente eficiente para ordenar arquivos de dados.
- Requer apenas uma pequena pilha como memória auxiliar.

Desvantagens

- Implementação delicada e difícil: um pequeno engano pode levar a efeitos inesperados.

Conteúdo

Redução do tempo de execução

- Usar **algoritmo de inserção** (**InsertionSort**) para vetores pequenos (menos de 10 ou 20 elementos).
- Mediana de três: não empilhar quando tem apenas um item.
- Escolha correta do lado a ser empilhado.

Resultado

- **Melhoria no tempo de execução de 25% a 30%.**

Conteúdo

Conclusão

- Nesta aula tivemos contato com o *algoritmo de ordenação* chamado **QuickSort**.
- Foram vistas duas versões: **Recursiva** e **Iterativa**.
- Provavelmente este algoritmo é o mais utilizado.

Conclusão

- 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	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$	**Sim**	**Sim**
Merge	$O(n \log n)$			–			$O(n)$	**Sim**	**Não**
Quick	$O(n \log n)$		$O(n^2)$	–			$O(n)$	**Não***	**Sim**

* Existem versões estáveis.

Conclusão

- A tarefa de ordenação é muito importante, ela é uma necessidade básica para a solução de muitos problemas.
- *Próxima aula*: *ShellSort*.
- Dúvidas?

Conteúdo

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).