Ordenação Eficiente

Luciano Nascimento Moreira

Ordenação Eficiente

- Métodos simples:
  - Adequados para pequenos arquivos.
  - Requerem $O(n^2)$ comparações.
  - Produzem programas pequenos.
- Métodos eficientes:
  - Adequados para arquivos maiores.
  - Requerem $O(n \log n)$ comparações.
  - Usam menos comparações.
  - As comparações são mais complexas nos detalhes.
  - Métodos simples são mais eficientes para pequenos arquivos.

ShellSort

- Proposto por Shell em 1959.
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas n - 1 comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

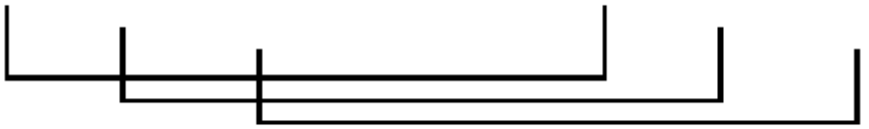
ShellSort

- Os itens separados de h posições são rearranjados.
- Todo h-ésimo item leva a uma sequência ordenada.
- Tal sequência é dita estar h-ordenada.

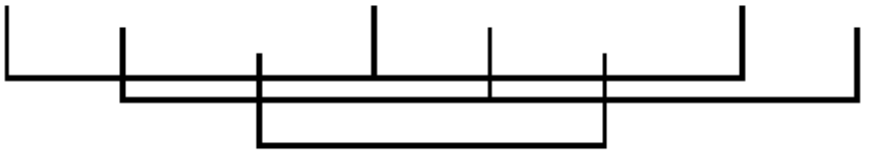
ShellSort

Original 25 57 48 37 12 92 86 33

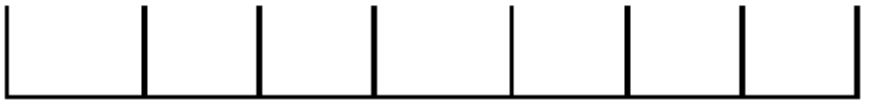
Incremento 5 25 57 48 37 12 92 86 33



Incremento 3 25 57 33 37 12 92 86 48



Incremento 1 25 12 33 37 48 92 86 57



12 25 33 37 48 57 86 92

ShellSort

- Como escolher o valor de h:
  - Sequência para h:

$$h(s) = 1, \qquad \text{para } s = 1$$

ShellSort

- Como escolher o valor de h:
  - Sequência para h:

$$h(s) = 1, \quad \text{para } s = 1$$

$$h(s) = 3h(s - 1) + 1, \quad \text{para } s > 1$$

ShellSort

- Como escolher o valor de h:
  - Sequência para h:

$$h(s) = 1, \quad \text{para } s = 1$$

$$h(s) = 3h(s - 1) + 1, \quad \text{para } s > 1$$

  - A sequência para h corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, ...
  - Knuth (1973, p. 95) mostrou experimentalmente que esta sequência é difícil de ser batida por mais de 20% em eficiência.

ShellSort

```
void Shellsort (Item[] A, int n){
    int i, j;
    int h = 1;
    Item aux;
    do {
        h = h * 3 + 1;
    } while (h < n);
    do {
        h /= 3;
        for( i = h ; i < n ; i++ ) {
            aux = A[i];
            j = i;
            while ((j >= h) && aux.compara(A[j – h]) < 0){
                A[j] = A[j – h];
                j -= h;
            }
            A[j] = aux;
        }
    } while (h != 1);
}
```

Exemplo

O R D E N A

h = 4

h = 2

h = 1

Shellsort

- Exemplo de utilização:

	0	1	2	3	4	5
	O	R	D	E	N	A
h=4	**N**	**A**	D	E	**O**	**R**
h=2	**D**	A	**N**	E	O	R
h=1	**A**	**D**	**E**	**N**	O	R

- Quando *h* = *1*, Shellsort corresponde ao algoritmo de inserção.

ShellSort

- A implementação do Shellsort não utiliza registros sentinelas.
- Seriam necessários h registros sentinelas, uma para cada h-ordenação.

ShellSort

- Análise
  - A razão da eficiência do algoritmo ainda não é conhecida, pois ninguém ainda foi capaz de analisar o algoritmo.
  - A sua análise contém alguns problemas matemáticos muito difíceis.
  - A começar pela própria sequência de incrementos, o que se sabe é que cada incremento não deve ser múltiplo do anterior.

ShellSort

- Análise
  - Conjecturas referente ao número de comparações para a sequência de Knuth:
    - Conjectura 1 : $C(n) = O(n^{1,25})$
    - Conjectura 2 : $C(n) = O(n\ (\log n)^2\ )$

ShellSort

- Vantagens:
  - Shellsort é uma ótima opção para arquivos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.
- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - O método não é estável,

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.
- A ideia básica é dividir o problema de ordenar um conjunto com *n* itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor A [Esq..Dir] é rearranjado por meio da escolha arbitrária de um **pivô** x.
- O vetor A é particionado em duas partes:
  - Parte esquerda: chaves ≤ x.
  - Parte direita: chaves ≥ x.

Quicksort - Partição

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

Quicksort – Após a Partição

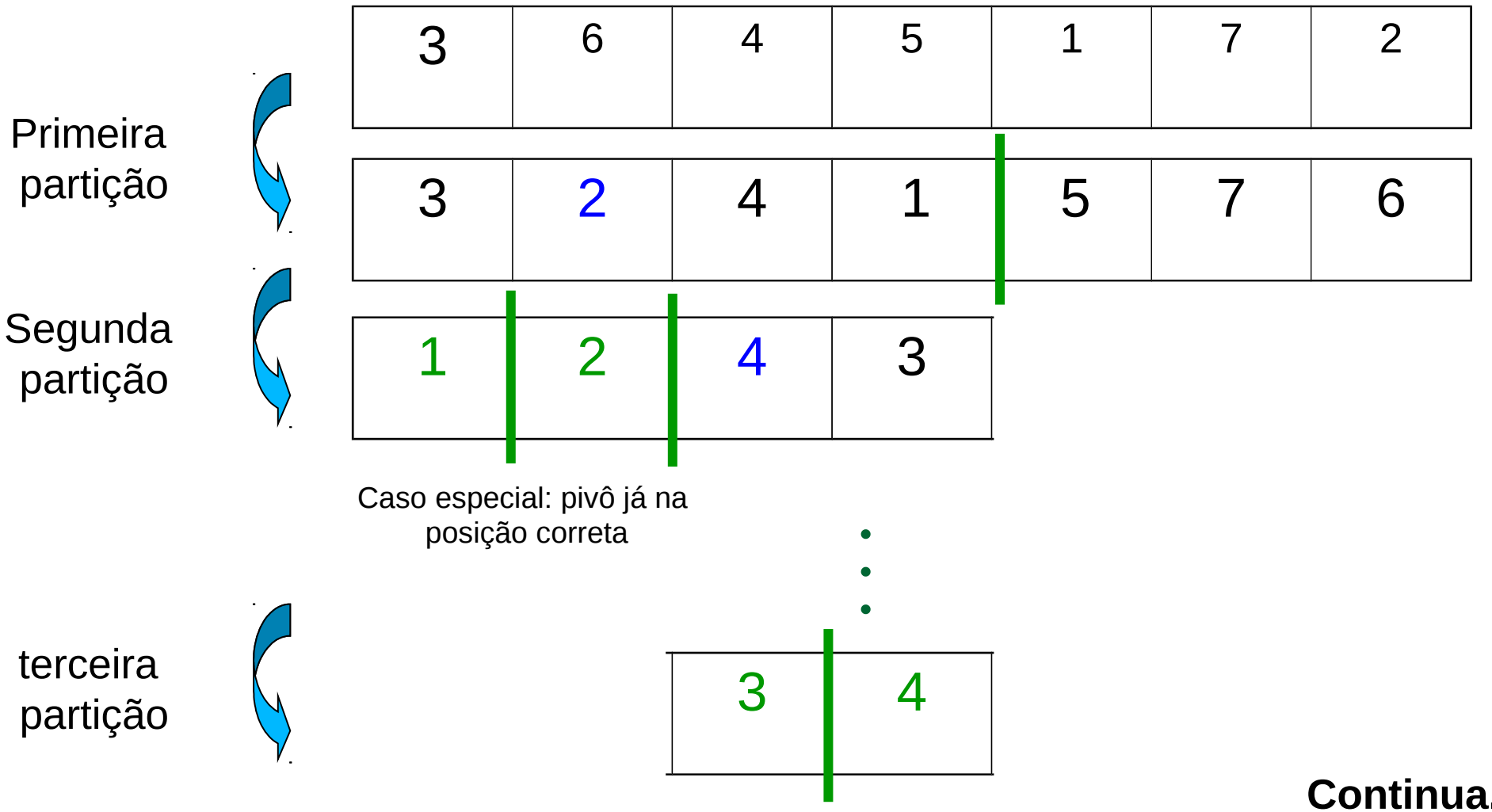
- Ao final, do algoritmo de partição:
  - o vetor A[Esq..Dir] está particionado de tal forma que:
    - Os itens em A[Esq], A[Esq + 1], ..., A[j] são menores ou iguais a x;
    - Os itens em A[i], A[i + 1], ..., A[Dir] são maiores ou iguais a x.

Quicksort - Exemplo

- O pivô x é escolhido como sendo:
  - O elemento central: A[(i + j) / 2].
- Exemplo:

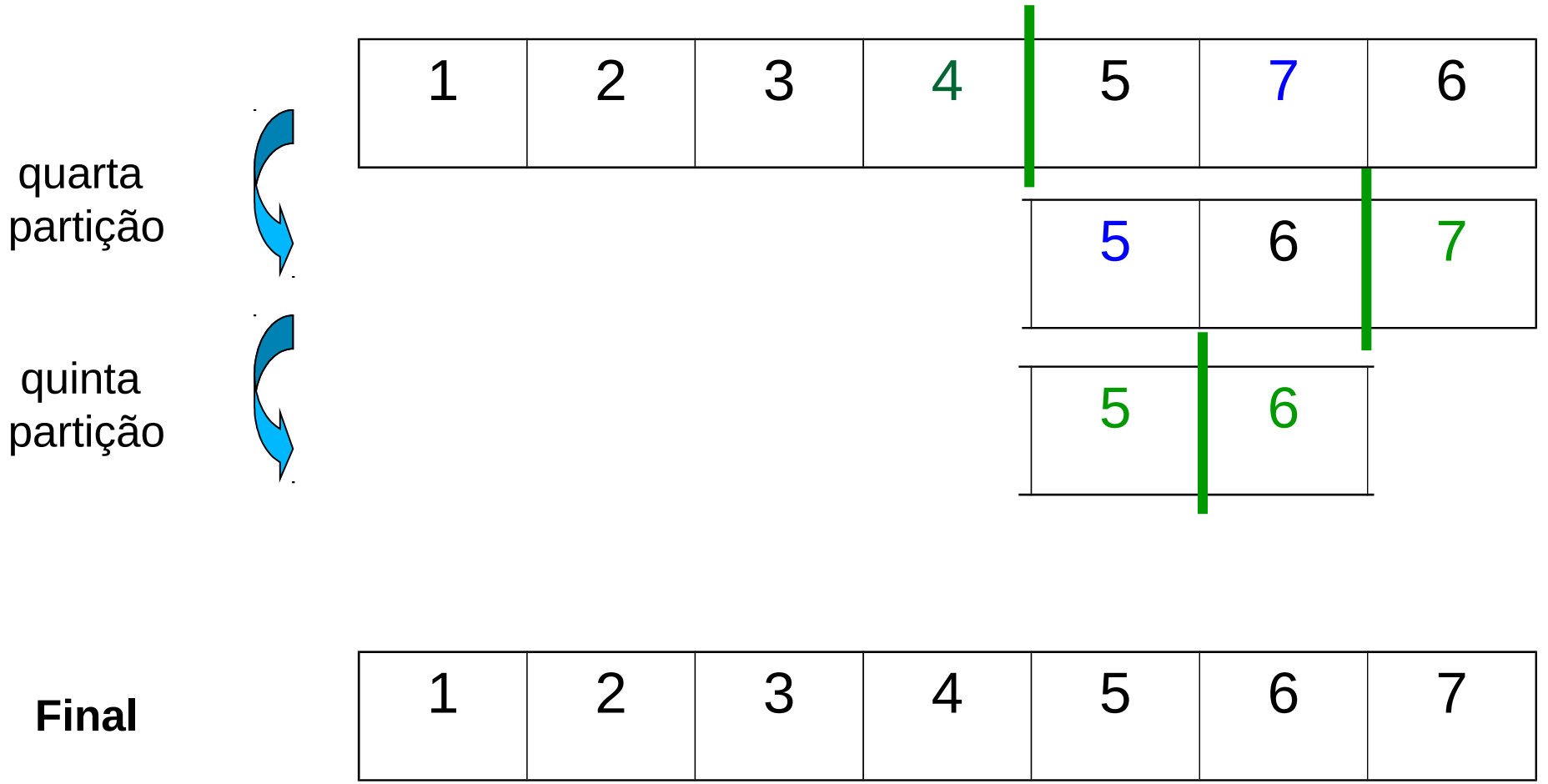
3	6	4	5	1	7	2

Quicksort - Exemplo



Continua...

Quicksort - Exemplo



Quicksort

```
void quicksort(Item[] v, int n) {
    ordena (vetor, 1, n);
}


void ordena(Item[] vetor, int esq, int dir) {
    if (esq < dir){
        int p = particiona(vetor, esq, dir);
        ordena(vetor, esq, p);
        ordena(vetor, p+1, dir);
    }
}
```

Quicksort - Partição

```
int Particao(Item[] v, int esq, int dir) {
    Item x, aux;
    int i = esq, j = dir;
    Item x = v[(i + j)/2]; // obtém o pivô x
    do {
        while (x.compara(v[i]) > 0) i++;
        while (x.compara(v[j]) < 0) j--;
        if (i <= j) {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
            j--;
        }
    } while (i <= j); // i<j ?
    return j;
}
```

O anel interno da função partição é extremamente simples, razão pela qual o algoritmo Quicksort é tão rápido

Quicksort

- Características
  - Qual o pior caso para o Quicksort?
    - Por que?
    - Qual sua ordem de complexidade?
  - Qual o melhor caso?
  - O algoritmo é estável?

Quicksort: Análise

- Seja C(n) a função que conta o número de comparações.
- Pior caso: $C(n) = O(n^2)$
  - O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
  - Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.
  - O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
  - Para isso basta escolher três itens quaisquer do vetor e usar a mediana dos três como pivô.

Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

  - Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

$$C(n) \approx 1{,}386n \log n - 0{,}846n,$$

  - Isso significa que em média o tempo de execução do Quicksort é O(n log n).

Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de n log n comparações em média para ordenar n itens.
- Desvantagens:
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada e difícil:
    - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é estável.

Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

  - Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

$$C(n) \approx 1{,}386n \log n - 0{,}846n,$$

  - Isso significa que em média o tempo de execução do Quicksort é O(n log n).