Estrutura de Dados

Prof. Tatiane Fernandes

UNIVERSIDADE FEDERAL DO CEARÁ
Campus Russas

Análise de Algoritmos

- A análise de algoritmos estuda a correção e o desempenho de algoritmos.
- Em outras palavras, a análise de algoritmos procura respostas para perguntas do seguinte tipo:

  "Este algoritmo resolve o meu problema?

  “Quanto tempo o algoritmo consome para processar uma 'entrada' de tamanho n?"

- A resposta à segunda pergunta é necessariamente um tanto grosseira, algo como "o consumo de tempo é proporcional a $n^2 \log n$ no pior caso".)

Análise assintótica

- Ao ver uma expressão como n+10 ou n^2+1, a maioria das pessoas pensa automaticamente em valores pequenos de n.
- A análise de algoritmos faz exatamente o contrário: <u>ignora os valores pequenos</u> e <u>concentra-se nos valores enormes de n</u>.
- Para valores enormes de n, as funções:

  $n2$, $(3/2)n^2$, $9999n^2$, $n^2/1000$, n^2+100n ,   etc.

  crescem todas com a mesma velocidade e portanto são todas "equivalentes".

Análise assintótica

- Esse tipo de matemática, interessado somente em valores enormes de n, é chamado ***assintótico***.
- Nessa matemática, as funções são classificadas em ***"ordens"*** (como as ordens religiosas da Idade Média);
- Todas as funções de uma mesma ordem são "***equivalentes***".
- As cinco funções apresentadas, por exemplo, pertencem à mesma ordem.

Ordem O

- Convém restringir a atenção a funções assintoticamente ***<u>não negativas</u>***, ou seja, funções f tais que $f(n) \geq 0$ para todo n suficientemente grande.

- Mais explicitamente: f é assintoticamente não negativa se existe n_0 tal que $f(n) \geq 0$ para todo n maior que n_0.

Ordem O

- **Definição:** Dadas funções assintoticamente não negativas f e g, dizemos que f está na ordem O de g e escrevemos $f = O(g)$ se $f(n) \leq c \cdot g(n)$ para algum c positivo e para todo n suficientemente grande.

  → Em outras palavras, existe um número positivo c e um número n0 tais que $f(n) \leq c \cdot g(n)$ para todo n maior que n0.

Exemplos

- Exemplo : Suponha que $f(n) = 2n^{\overset{3}{\cancel{2}}} + 3n + 4$ e $g(n) = n^2$. Observe que:

f(n) <= c g(n)
2n2 + 3n + 4 <= c n2

$$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 = 9n^2$$

desde que $n \geq 1$. Resumindo, $f(n) \leq 9\, g(n)$ para todo $n \geq 1$. Portanto, $f(n) = O(g(n))$.

Exercícios

- Exercício 1: 100n está em $O(n^2)$

Exercícios

- Exercício 1: 100n está em $O(n^2)$

→ Para todo $n \geq 100$

f(n) <= c .g(n)

$$100n \leq n \cdot n$$

$$= n^2$$

$$= 1 \cdot n2$$

Exercícios

- Exercício 1: $2n^3 + 100n$ está em $O(n^3)$

Exercícios

- Exercício 1: $2n^3 + 100n$ está em $O(n^3)$

$\rightarrow$ Para todo $n \geq 1$

f(n) <= c n3

$$2n^3 + 100n \leq 2n^3 + 100n^3$$

$$\leq 102n^3$$

Analisando Algoritmos

PROBLEMA DA ORDENAÇÃO: Rearranjar um vetor $A[1..n]$ de modo que ele fique em ordem crescente.

ORDENAÇÃO-POR-INSERÇÃO (A, n)

```
para j crescendo de 2 até n faça
    x ← A[j]
    i ← j–1
    enquanto i > 0 e A[i] > x faça
        A[i+1] ← A[i]
        i ← i–1
    A[i+1] ← x
```

Quanto tempo consome?

A pergunta fundamental da análise de algoritmos é ***quanto tempo o algoritmo consome?*** À primeira vista, a pergunta não faz sentido porque o tempo depende:

1- da instância do problema, ou seja, do particular vetor A[1..n] sendo ordenado e

2 -da máquina (computador) sendo usada.

Quanto tempo consome?

- Para responder à primeira objeção, digo que estou interessado no pior caso: para cada valor de n, considero a instância A[1..n] para a qual o algoritmo consome mais tempo. Vamos denotar esse tempo por T(n).

- Para responder à segunda objeção, observo que o tempo não depende tanto assim do computador. Ao mudar de um computador para outro, o consumo de tempo do algoritmo é apenas multiplicado por uma constante.

→ *Por exemplo, se $T(n) = 250n^2$ em um computador, então $T(n) = 500n^2$ em um computador duas vezes mais lento e $T(n) = 25n^2$ em um computador dez vezes mais rápido.*

Quanto tempo consome?

Podemos tratar então da determinação de T(n) para o algoritmo de inserção. A coluna direita da figura abaixo registra o número de execuções de cada linha no pior caso.

ORDENAÇÃO-POR-INSERÇÃO (A, n)

para j crescendo de 2 até n faça	n
$x \leftarrow A[j]$	$n-1$
$i \leftarrow j-1$	$n-1$
enquanto $i > 0$ e $A[i] > x$ faça	$2+3+\ldots+n$
$A[i+1] \leftarrow A[i]$	$1+2+3+\ldots+n-1$
$i \leftarrow i-1$	$1+2+3+\ldots+n-1$
$A[i+1] \leftarrow x$	$n-1$

- Suponha agora que a execução de qualquer das linha do código consome 1 unidade de tempo. Então o tempo no pior caso será a soma da coluna direita da figura:

$$T(n) = (3/2)n^2 + (7/2)n - 4.$$

- ***Se tivéssemos levado em conta o tempo exato de execução de cada linha, obteríamos coeficientes diferentes de 3/2, 7/2 e −4, mas a expressão de T(n) ainda seria da forma $an^2 + bn + c$.***

Quanto tempo consome?

- ***O coeficiente 3/2 de n^2 não é importante: ele não depende do algoritmo mas de nossa hipótese "1 unidade de tempo por linha". Já o "n^2" é fundamental: ele é característico do algoritmo em si e não depende nem do computador nem dos detalhes da implementação do algoritmo. Em resumo, a única parte importante em T(n) é o "n2". Dizemos que a quantidade de tempo que o algoritmo Ordenação-Por-Inserção consome no pior caso é $\Theta(n^2)$.***

  Dizemos que o algoritmo é quadrático.

Ordem Omega

A expressão "f = O(g)" tem o mesmo sabor que "f ≤ g". Agora precisamos de um conceito que tenha o sabor de ***"f ≥ g"***.

Definição: Dadas funções assintoticamente não negativas f e g, dizemos que f está na ordem Omega de g e escrevemos $f = \Omega(g)$ se $f(n) \geq c \cdot g(n)$ para algum c positivo e para todo n suficientemente grande. Em outras palavras, existe um número positivo c e um número n0 tais que $f(n) \geq c \cdot g(n)$ para todo n maior que n0.

Ordem Teta

Além dos conceitos que têm o sabor de "$f \leq g$" e de "$f \geq g$", precisamos de um que tenha o sabor de ***"f = g"***.

Definição: Dizemos que f e g estão na mesma e escrevemos $f = \Theta(g)$ se $f = O(g)$ e $f = \Omega(g)$. Trocando em miúdos, $f = \Theta(g)$ significa que existe números positivos c e d tais que $c\, g(n) \leq f(n) \leq d\, g(n)$ para todo n suficientemente grande.

Exemplo: As funções abaixo pertencem todas à ordem Θ(n2):

n^2 , $(3/2)n^2$, $9999n^2$, $n^2/1000$, n^2+100n

Algoritmos de ordenação

Mergesort

→ Mergesort é um algoritmo de ordenação recursivo;

→ Ele recursivamente ordena as duas metades do vetor;

→ Usa a estratégia de divisão e conquista;

→ Mergesort é um algoritmo eficiente;

→ Tem tempo de execução O (n log n);

Método de Divisão e Conquista

- **Divisão:** Divida o problema em duas ou mais partes, criando subproblemas menores;
- **Conquista**: Os subproblemas são resolvidos recursivamente usando divisão e conquista. Caso os subproblemas sejam suficientemente pequenos resolva-os de forma direta;
- **Combina:** Tome cada uma das partes e junte-as todas de forma a resolver o problema original;

Mergesort

Caso o tamanho do vetor seja maior que 1:

1. divida o vetor no meio;
2. ordene a primeira metade recursivamente;
3. ordene a segunda metade recursivamente;
4. intercale as duas metades se possível: senão devolva o elemento.

Intercalação

- A intercalação de dois vetores ordenados pode ser feito em tempo linear;
- Uma variável em cada vetor indica o próximo elemento a ser inserido a lista intercalada;
- Enquanto ambas os vetores tiverem elementos;
- Coloque o menor entre os dois elemento indicados no vetor intercalado e incremente índice respectivo;
- Quando um dois vetores não tiver mais elementos, concatene o outro no final do vetor intercalado.

Mergesort

Problema: Dados $A[p \mathinner{.\,.\,.} q]$ e $A[q+1 \mathinner{.\,.\,.} r]$ crescentes, rearranjar $A[p \mathinner{.\,.\,.} r]$ de modo que ele fique em ordem crescente.

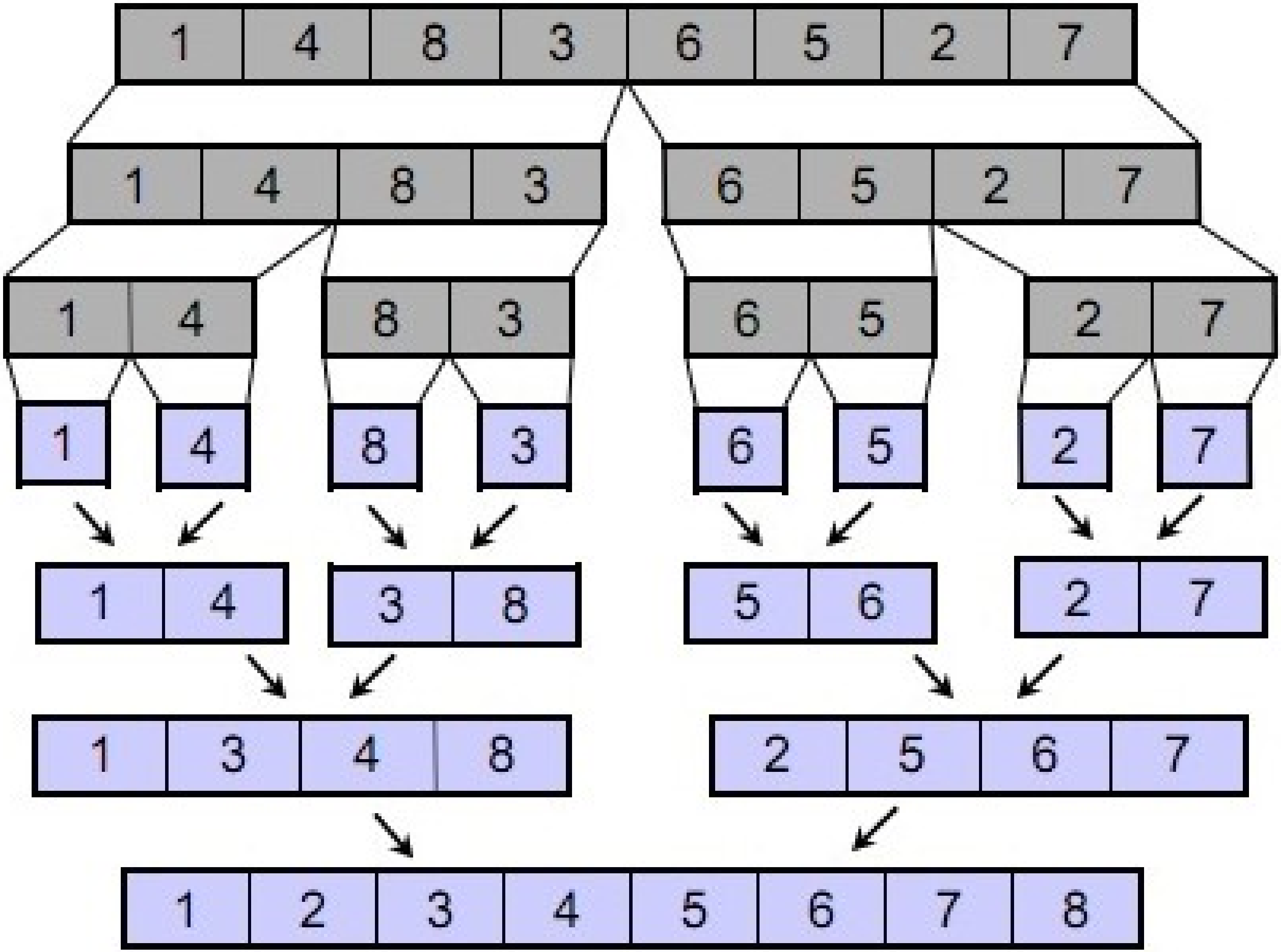
Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

MergeSort - Exemplo



Mergesort

```
MERGESORT (A, p, r)
se p < r então
      q ← ⌊(p+r)/2⌋
      MERGESORT (A, p, q)
      MERGESORT (A, q+1, r)
      INTERCALA (A, p, q, r)
```

```
INTERCALA (A, p, q, r)
para i crescendo de p até q faça
      B[i] ← A[i]
para j crescendo de q+1 até r faça
      B[r+q+1–j] ← A[j]
i ← p
j ← r
para k crescendo de p até r faça
      se B[i] ≤ B[j]
            então A[k] ← B[i]
                        i ← i+1
            senão A[k] ← B[j]
                        j ← j–1
```