**Instituto de Ciências Matemáticas e de Computação**

**Graduação - Engenharia de Computação**

`http://tidia-ae.usp.br/`

| | |
|---|---|
| **Curso:** | **SCC0602 Algoritmos e Estrutura de Dados I** |
| **Responsável:** | **Profa. Maria Carolina Monard** `mcmonard@icmc.usp.br` |
| **Estagiário PAE:** | **Ígor Assis Braga** `igorab@icmc.usp.br` |
| **Semestre:** | $2^o$ **de 2008** Sala 01 Bloco 2 Campus II |
| | $4^{as}$ feiras 10:10–11:50hs |
| | $5^{as}$ feiras 16:20–18:00hs |

## Nota–Complexidade

Ainda que os algoritmos que serão discutidos neste curso representam somente uma muito pequena fração do código que é gerado em um projeto de software de grande porte, essa fração é extremamente importante para o sucesso de todo o projeto. Entretanto, uma abordagem freqüentemente usada consiste em projetar um algoritmo e uma estrutura de dados ineficiente para resolver o problema e, após, tentar melhorar esse projeto para melhorar a performance. Essa abordagem é incorreta pois, se o projeto inicial não for bom, não será possível melhorar realmente a sua performance. Assim, é de fundamental importância tanto o entendimento de técnicas para projetar bons algoritmos quanto a analise da eficiência desses algoritmos.

Achar o tempo de execução de um algoritmo pode exigir a derivação de fórmulas, ou funções, bastante complexas. Assim, é interessante encontrar uma maneira simples de representar essas funções complexas tal que é possível capturar as propriedades de crescimento dessas funções. Esse é o objetivo da *análise assintótica* de algoritmos. A análise assintótica está baseada em duas suposições, as quais verificam-se na maioria, mas não em todos, os casos. Assim, é importante entender essa suposições bem como as limitações da análise assintótica. A primeira suposição tem a ver com o tamanho da entrada de um algoritmo, ao qual denotaremos por $n$. A análise assintótica considera valores grandes de $n$ e tem como foco verificar como incrementa o tempo de execução de um algoritmo para valores grandes de $n$. A segunda suposição consiste em ignorar fatores constantes, tais como velocidade do hardware, otimização de compiladores e outros.

A justificativa de considerar valores de $n$ grandes deve-se ao fato de caso $n$ ser pequeno, então, praticamente qualquer algoritmo executa o suficientemente rápido. Assim, em geral, o interesse está no tempo de execução de algoritmos para $n$ grande. Para a maioria dos casos, as duas suposição da análise assintótica são razoáveis quando são realizadas comparações entre funções que têm comportamentos significativamente diferentes. Por exemplo, considerando dois algoritmos, um com tempo de execução $T_1(n) = n^3$ e outro com tempo de execução $T_2(n) = 100n$, o primeiro algoritmo executa em menos tempo que o segundo para $n$ pequeno ($n < 10$). Entretanto, para valores crescentes de $n$, a diferença relativa no tempo de execução cresce. Na Tabela 1 são mostrados alguns desses valores considerando que são realizadas um milhão de operações por segundo. Como pode ser observado, a performance do algoritmo que é assintoticamente pior degrada muito mais rapidamente.

Para representar o tempo de execução de um algoritmo na sua forma mais simples, é utilizada a *notação assintótica* a qual representa uma função pelo termo que cresce mais rapidamente, ignorando fatores constantes. Para o propósito de análise de algoritmos, as seguintes definições baseadas em limites são simples de usar e valem para praticamente todas as funções que resultam na análise de tempo de execução de algoritmos [1]

[1] Funções para as quais não existem esses limites devem ser tratadas usando a definição formal. Definições

| $n$ | $T_1(n)$ | $T_2(n)$ | $\frac{T_1(n)}{T_2(n)}$ |
|---|---|---|---|
| 10 | 0.001 seg | 0.001 seg | 1 |
| 100 | 1 seg | 0.01 seg | 100 |
| 1000 | 17 min | 0.1 seg | 10.000 |
| 10.000 | 11.6 dias | 1 seg | 1.000.000 |

**Tabela 1:** Tempo de execução de $T_1(n) = n^3$ e $T_2(n) = 100n$

Sejam $f(n)$ e $g(n)$ duas funções que assumimos serem positivas. Considerando que quer-se afirmar que $f(n)$ e $g(n)$ crescem aproximadamente com a mesma velocidade, ou razão, para $n$ grande, ignorando fatores constantes, isso seria equivalente a afirmar que

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

com $c$ uma constante diferente de zero. Na notação assintótica escreve-se $f(n) \in \Theta(g(n))$. Intuitivamente, significa que $f(n)$ e $g(n)$ são assintoticamente equivalentes. No caso de querer afirmar que $f(n)$ não cresce significativamente mais rápido que $g(n)$, então, no limite, a razão $\frac{f(n)}{g(n)}$ deve-se aproximar a uma constante diferente de zero, *i.e.* $f(n)$ e $g(n)$ são assintoticamente equivalentes, or 0 (zero). Nesse último caso diz-se que $f(n) \in O(g(n))$. Deve ser observado que freqüentemente é utilizado o símbolo $=$ em lugar do símbolo $\in$, mas o correto é considerar $O(g(n))$ como um conjunto de funções. A Tabela 2 mostra todas as definições[2]

| Forma Assintótica | Relação | Definição |
|---|---|---|
| $f(n) \in \Theta(g(n))$ | $f(n) \equiv g(n)$ | $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ |
| $f(n) \in O(g(n))$ | $f(n) \preceq g(n)$ | $0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ |
| $f(n) \in \Omega(g(n))$ | $f(n) \succeq g(n)$ | $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$ |
| $f(n) \in o(g(n))$ | $f(n) \prec g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ |
| $f(n) \in \omega(g(n))$ | $f(n) \succ g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ |

**Tabela 2:** Notações assintóticas

Por exemplo, $T(n) = (n^3 + 3n^2 + 2n)/6 \in \Theta(g(n))$ pois

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{n^3 + 3n^2 + 2n)/6}{n^3} = \lim_{n\to\infty} \left(\frac{1}{6} + \frac{1}{2n} + \frac{1}{3n^2}\right) = \frac{1}{6}$$

e $0 < \frac{1}{6} < \infty$. Também pode ser notado que $T(n) \in O(n^3)$ e $T(n) \in \Omega(n^3)$, o que é consistente com a definição informal da notação assintótica relacionada a ignorar fatores constantes considerando somente valores de $n$ grandes, visto que $n$ elevado ao maior expoente será o fator dominante quando $n$ cresce.

Seguem algumas regras úteis para trabalhar com limites.

**Regra de L'Hôpital** Se ambas funções, $f(n)$ e $g(n)$ tendem a 0, ou ambas tendem a $\infty$ no

formais dessas funções serão vistas na disciplina SCC0601 Introdução à Ciência de Computação II, na qual o tema análise de algoritmos faz parte do programa.

[2]Neste curso utilizaremos principalmente o conjuntos de funções $O(g(n))$ para analisar a complexidade de algoritmos.

limite, então

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

onde $f'(n)$ e $g'(n)$ são as derivadas de $f$ e $g$ em relação a $n$.

**Constantes** Constantes multiplicativas e aditivas podem ser ignoradas. Entretanto, quando constantes aparecem como expoentes ou como a base de um exponencial, elas são significativas. Por exemplo, $2n \equiv 3n$, mas $n^2 \prec n^3$ e $2^n \prec 3^n$.

**Base de logaritmos** Logaritmos em bases diferentes somente diferem em um fator constante, ou seja, $\log_a n \equiv \log_b n$. Esse é o motivo pelo qual não é especificada a base de logaritmos na notação assintótica, como no caso de $O(n \log n)$, pois não tem importância.

**Logaritmos e expoentes** Lembrar que é possível colocar como um multiplicador o expoente de um logaritmo. Por exemplo, $n \log_2(n^2) = 2n \log_2 n \equiv nlog_2n$

**Expoentes e logaritmos** Lembrar que exponenciais e logaritmos cancelam um com outro. Por exemplo, $2^{\log n} = n$.

**Logaritmos e polinomiais** Para qualquer $a, b \geq 0, (\log n)^a \prec n^b$. Em outras palavras, logs crescem mais devagar que qualquer polinomial.

**Polinomiais e exponenciais** Para qualquer $a \geq 0$, $b > 1$, $n^a \prec b^n$. Em outras palavras, polinomiais crescem mais devagar que qualquer exponencial.

**Serie aritmética** Para $n \geq 0$

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

que é $\Theta(n^2)$

**Serie geométrica** Seja $x \neq 1$ uma constante qualquer independente de $n$, então, para $n \geq 0$

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^n = \frac{x^{(n+1)} - 1}{x - 1}$$

Se $0 < x < 1$ então ela é $\Theta(1)$. Se $x > 1$ então ela é $\Theta(x^n)$; em outras palavras, a soma total é proporcional ao último elemento da serie.

**Serie quadrática** Para $n \geq 0$

$$\sum_{i=0}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{2n^3 + 3n^2 + n}{6}$$

**Serie linear-geométrica** Esta serie aparece freqüentemente na análise de algoritmos recursivos para árvores. Seja $x \neq 1$ uma constante qualquer independente de n, então, para $n \geq 0$

$$\sum_{i=0}^{n-1} ix^i = x + 2x^2 + 3x^3 + \ldots + (n-1)x^{(n-1)} = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}$$

Observar que para $x = 1$ tem-se a serie aritmética. Quando $n$ cresce, o termo dominante é $(n-1)x^{(n+1)}/(x-1)^2$. Também, para $n$ grande o termo multiplicativo $(n-1)$ é aproximadamente igual a $n$. Assim, como $x$ é uma constante, é possível multiplicar o termo dominante pela constante $(x-1)^2/x$ sem afetar o resultado assintótico, o que mostra que essa soma é $\Theta(nx^n)$

**Serie harmonica** Esta serie aparece freqüentemente na análise probabilística (caso médio) de algoritmos. Ela não possui uma solução exata mas uma solução aproximada. Para $n \geq 0$, e $n$ crescente

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n$$

**Regras gerais para calcular o tempo de execução** $T(n)$

- Regra 1 – `for` *loops*: dado pelo tempo de execução dos comandos dentro do `for` *loop* multiplicado pelo número de iterações
- Regra 2 – *nested* `for` *loops*: dado pelo tempo de execução do comando multiplicado pelo produto da repetição dentro de cada *loops*. Por exemplo, o seguinte fragmento de programa é $O(n^2)$

```
for i:= 1 to n do
    for j:= 1 to n do k:= k + 1:
```

- Regra 3 – comandos consecutivos: dado pela adição dos tempos de execução dos comandos. Por exemplo, dado o seguinte fragmento de programa

```
for i:= 1 to n do
    a[i] := 0;
for i:= 1 to n do
    for j := 1 to n do a[i] := a[i] + a[j] + i + j;
```

  o primeiro fragmento é $O(n)$ enquanto que o segundo é $O(n^2)$, portanto o fragmento total é $O(n^2)$

- Regra 4 – `if/else`: para o fragmento de programa

```
if <condicao> then S1 else S2;
```

  o tempo de execução nunca é maior que o tempo de execução da `<condicao>` mais o maior tempo de execução do comando `S1` ou `S2`