Capítulo 4 - Crescimento de Funções

Prof. Manoel Ribeiro

4.1 Introdução: Notações O (omicron), Ω(Ômega) e θ(Teta)

A análise de algoritmos trata principalmente do desempenho de pior caso. Felizmente, não é preciso determinar uma fórmula exata para T(n): basta entender o comportamento assintótico da função, ou seja, o comportamento quando n tende a infinito.

Para valores grandes de n, o termo dominante das fórmulas é muito maior que os demais e portanto os termos de ordem inferior podem ser desprezados .

Na fórmula $10n^2 + 100n$, por exemplo, o segundo termo pode ser desprezado porque é dominado pelo primeiro. Assim, a fórmula se reduz a $10n^2$. Além disso, as constantes multiplicativas — como o “10” em “$10n^2$” — podem ser ignoradas pois seu efeito será anulado se o computador for substituído por outro mais rápido. Com tudo isso, em vez de dizer que $T(n) = 10n^2 + 100n$, é mais apropriado dizer que T(n) está em $O(n^2)$; onde o “O” serve para esconder os termos de ordem inferior e as constantes. Na verdade, a notação O( ) esconde mais que as constantes e os termos de ordem inferior, pois ela tem o sabor de “<=”. Ao dizer que T(n) está em $O(n^2)$, estamos afirmando apenas que $T(n) <= cn^2$ para alguma constante c (e todo n suficientemente grande).

Apesar do sabor “<=” de O( ), tornou-se hábito escrever “$T(n) = O(n^2)$” no lugar de “$T(n)$ está em $O(n^2)$”, o que pode ser um tanto confuso . . .

Há uma notação análoga com sabor de “>=”: dizemos que $T(n)$ está em $\Omega\ (n^2)$, ou que $T(n) = \Omega\ (n^2)$, se $T(n) >= d\ n^2$ para alguma constante d (e todo n suficientemente grande).

Finalmente, dizemos que $T(n)$ está em $\theta(n^2)$, ou que $T(n) = \theta\ (n^2)$, se $T(n)$ está em $O(n^2)$ e também em $\Omega(n^2)$. Assim, a notação $\theta(\ )$ tem sabor de “=”.

Lembre-se de que caracterizamos o tempo de execução do pior caso da ordenação por inserção como $an^2 + bn + c$, para algumas constantes a, b e c. Quando afirmamos que o tempo de execução da ordenação por inserção é $\Theta(n^2)$, abstraímos alguns detalhes dessa função. Como a notação assintótica aplica-se a funções, o que quisemos dizer é que $\Theta(n^2)$ era a função $an^2 + bn + c$ que, aqui, por acaso caracteriza o tempo de execução do pior caso da ordenação por inserção.

Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.

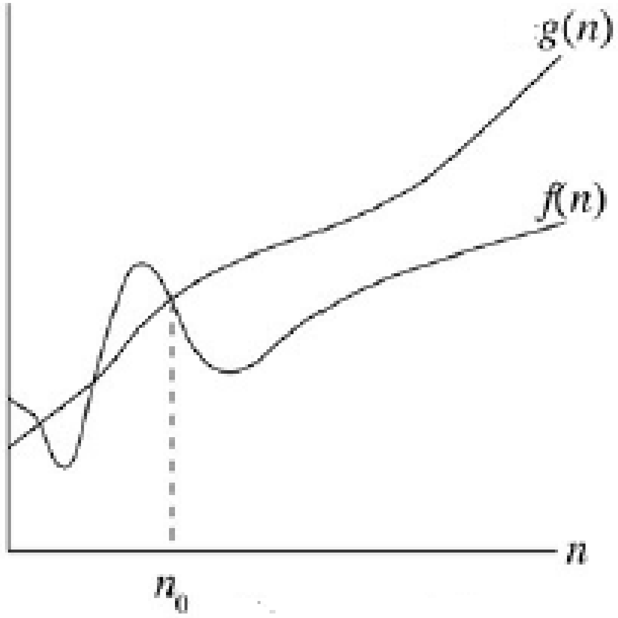
Várias comparações de complexidade assintótica podem ser definidas, mas as mais comumente usadas são O, Θ e Ω.

A s funções às quais aplicamos a notação assintótica, normalmente caracterizarão os tempos de execução de algoritmos.
Porém, a notação assintótica pode se aplicar a funções que caracterizam algum outro aspecto dos algoritmos (a quantidade de espaço que eles usam, por exemplo) .
Mesmo quando utilizamos a notação assintótica para o tempo de execução de um algoritmo, precisamos entender *a qual* tempo de execução estamos nos referindo.
Às vezes, estamos interessados no tempo de execução do pior caso.
Porém, frequentemente queremos caracterizar o tempo de execução, seja qual for a entrada. Em outras palavras, muitas vezes desejamos propor um enunciado abrangente que se aplique a todas as entradas, e não apenas ao pior caso.
As notações assintóticas prestam-se bem à caracterização de tempos de execução, não importando qual seja a entrada.

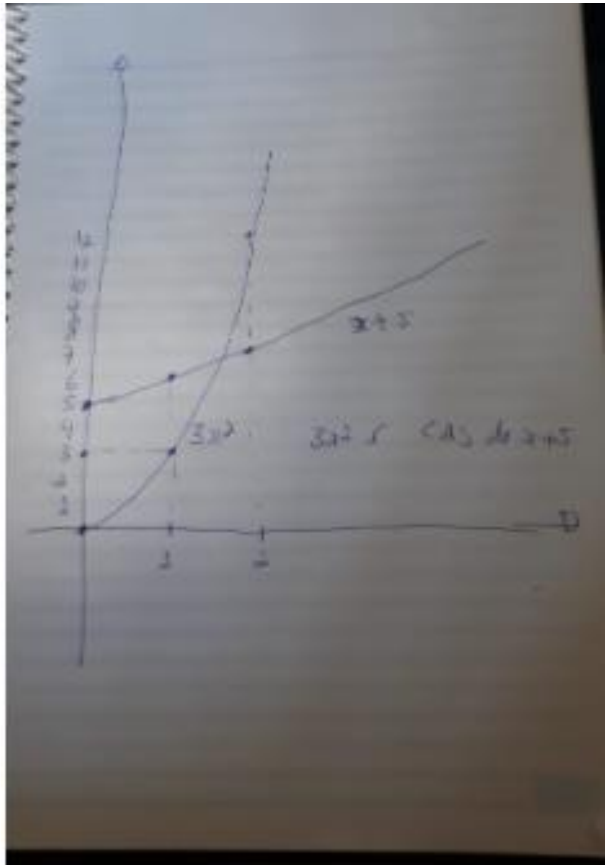
Uma ideia auxiliar útil é a de cota assintótica superior (CAS)

4.2 Cota Assintótica Superior

Uma CAS é uma função que cresce mais rapidamente do que outra: permanece acima a partir de certo ponto.
A função quadrática $3x^2$ cresce mais rapidamente que a função linear x +5 : dizemos que a função quadrática $3x^2$ é CAS para a linear x+5.



g(n) é CAS para f(n)



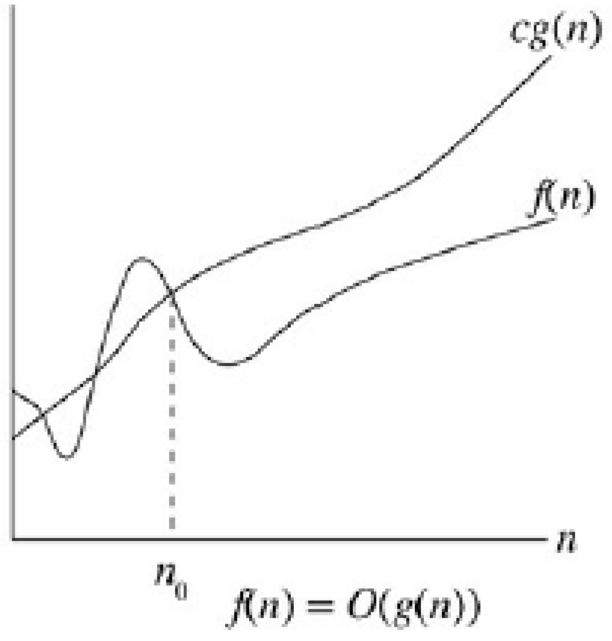
4.3 Notação O

A notação O define uma cota assintótica superior a menos de constantes.
Por exemplo, a função quadrática $g(n) = n^2$ cresce mais rapidamente do que a função linear $f(n) = 7n + 13$ (a partir de certo ponto). Dizemos que a linear f(n) é O(g(n)). Pode-se ler como f(n) é omicron de g(n), ou seja, f(n) é pequena para g(n). Porque a letra grega O chama-se omicron (O pequeno).

Para uma dada função *g*(*n*), denotamos por *O*(*g*(*n*)) (lê-se "*Ó* grande de *g* de *n*" ou, às vezes, apenas "*ó* de *g* de *n*") o conjunto de funções

$$O(g(n)) = \{f(n)\text{: existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}.$$

Usamos a notação *O* para dar um limite superior a uma função, dentro de um fator constante.



A notação *O* dá um limite superior para uma função dentro de um fator constante.
Escrevemos $f(n) = O(g(n))$ se existirem constantes positivas n_0 e c tais que, em n_0 e à direita de n_0
o valor de $f(n)$ sempre estiver abaixo de $cg(n)$.

Usando a notação O, podemos descrever frequentemente o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo. Por exemplo, a estrutura de loop duplamente aninhado do algoritmo de ordenação por inserção vista no Capítulo 2 produz imediatamente um limite superior $O(n_2)$ para o tempo de execução do pior caso: o custo de cada iteração do loop interno é limitado na parte superior por $O(1)$ (constante), os índices i e j são no máximo n, e o loop interno é executado no máximo uma vez para cada um dos n_2 pares de valores para i e j. Tendo em vista que a notação O descreve um limite superior, quando a empregamos para limitar o tempo de execução do pior caso de um algoritmo temos um limite para o tempo de execução do algoritmo em cada entrada — o enunciado abrangente do qual falamos anteriormente. Desse modo, o limite $O(n_2)$ para o tempo de execução do pior caso da ordenação por inserção também se aplica a seu tempo de execução para toda entrada.

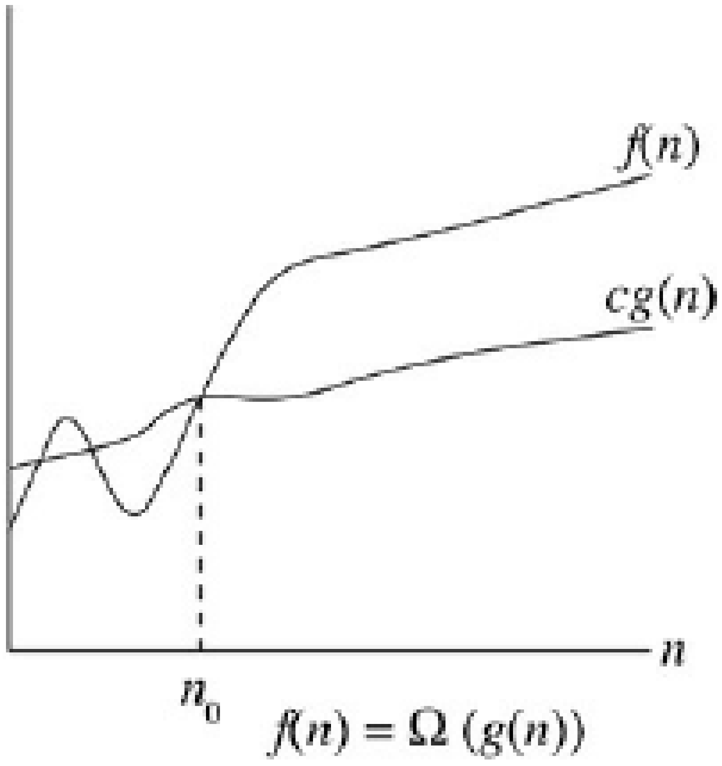
4.4 Notação Ω

A notação Ω define uma cota assintótica inferior a menos de constantes. Pro exemplo, a função cúbica $g(n) = 7.n^3 + 5$ cresce menos rapidamente do que a função exponencial $f(n) = 2^n$ (a partir de certo ponto). Nesse caso, diz-se que a exponencial f(n) é Ω (g(n)).

Porém, formalmente, definimos ω (*g*(*n*)) (lê-se "ômega pequeno de *g* de *n*") como o conjunto
$\omega(g(n)) = \{ f(n)$:
para qualquer constante positiva $c > 0$, existe uma constante

$n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0\}$.

A notação Ω dá um limite inferior para uma função dentro de um fator constante.



Escrevemos $f(n) = \Omega(g(n))$ se existirem constantes positivas n_0 e c tais que, em n_0 e à direita de n_0, o valor de $f(n)$ sempre estiver acima de $cg(n)$.

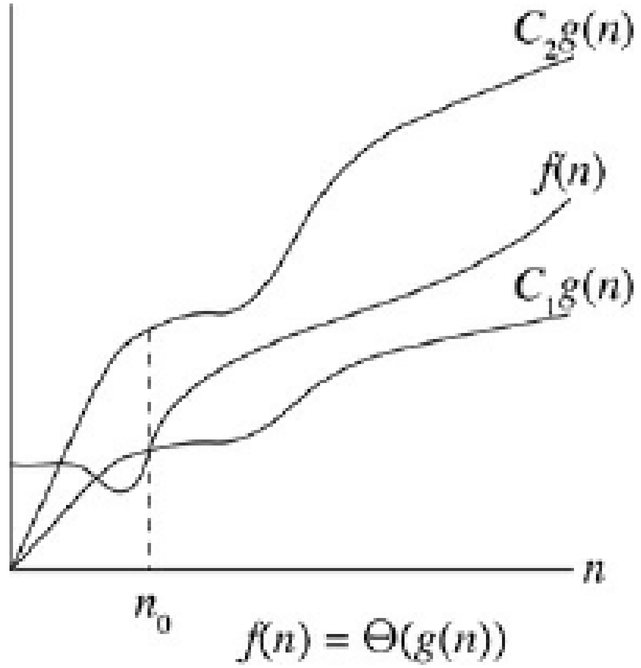
4.5 Notação Θ

Para uma dada função *g*(*n*), denotamos por Θ(*g*(*n*)) o *conjunto de funções*

$$\Theta(g(n)) = \{ f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0 \}.^1$$

Uma função *f*(*n*) pertence ao conjunto Θ(*g*(*n*)) se existirem constantes positivas *c*1 e *c*2 tais que ela possa ser "encaixada" entre *c*1*g*(*n*) e *c*2 *g*(*n*), para um valor de *n* suficientemente grande. Como Θ(*g*(*n*)) é um conjunto, poderíamos escrever "*f*(*n*) ∈ Θ(*g*(*n*))" para indicar que *f*(*n*) é um membro de (ou pertence a) Θ(*g*(*n*)). Em vez disso, em geral escreveremos "*f*(*n*) = Θ(*g*(*n*))" para expressar a mesma noção.

A notação Θ limita uma função entre fatores constantes



Escrevemos $f(n) = \Theta(g(n))$ se existirem constantes positivas n_0, $c1$ e $c2$ tais que, em n_0 e à direita de n_0 , o valor de $f(n)$ sempre encontrar-se entre $c1\ g(n)$ e $c2\ g(n)$ inclusive

4.6 Algoritmos lineares, linearítmicos e quadráticos.

Um algoritmo é **linear** se seu desempenho no pior caso está em $\Theta(n)$. É fácil entender o comportamento de um tal algoritmo: quando o tamanho de uma instância do problema dobra, o algoritmo consome duas vezes mais tempo. Algoritmos lineares são considerados muito rápidos pois o tempo que consomem é essencialmente igual ao necessário para a leitura dos dados de entrada.

Um algoritmo é **linearítmico** (ou **ene-log-ene**) se seu desempenho no pior caso está em Θ ($n \log n$). Se o tamanho n da instância do problema dobra, o consumo de tempo dobra e é acrescido de $2n$.

Um algoritmo é **quadrático** se consome $\Theta(n^2)$ unidades de tempo no pior caso. Se o tamanho da instância dobra, o consumo quadruplica.

Exercícios

1.1 O que é um algoritmo de tempo constante? Como ele se comporta quando o tamanho da instância do problema dobra?

1.2 O que é um algoritmo logarítmico? Como ele se comporta quando o tamanho da instância do problema dobra?

1.3 O que é um algoritmo cúbico? Como um algoritmo cúbico se comporta quando o tamanho da instância do problema dobra?

Funções e Taxas de Crescimento

- Tempo constante: O(1) (raro)
- Tempo sublinear (log(n)): muito rápido (ótimo)
- Tempo linear: (O(n)): muito rápido (ótimo)
- Tempo $n\log n$: Comum em algoritmos de divisão e conquista.
- Tempo polinomial n^k : Freqüentemente de baixa ordem ($k \leq$ 10), considerado eficiente.
- Tempo exponencial: 2^n , $n!$, n^n considerados intratáveis

Funções e Taxas de Crescimento

Exponential

· · · Polynomial

Superlinear · · ·

Linear

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n)$$

Linear

· · · Sublinear

Logarithmic

Constants

Calculando o Tempo de Execução

```
Início
  declare soma_parcial numérico;
  soma_parcial := 0;
  para i :=1 até n faça
    soma_parcial := soma_parcial+i*i*i;
  escreva(soma_parcial);
Fim
```

1 unidade de tempo

- 1 unidade para inicialização de i,
- n+1 unidades para testar se i<=n
- n unidades para incrementar i

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição)

1 unidade para escrita

Custo total: $6n + 4 = O(n)$

Calculando o Tempo de Execução

- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
- No exemplo anterior
  - A linha “soma_parcial := 0” é insignificante em termos de tempo
  - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha “soma_parcial := soma_parcial+i*i*i”
  - O que realmente dá a grandeza de tempo desejada é a repetição na linha “para i := 1 até n faça”

Em geral, não consideramos os termos de ordem inferior da complexidade de um algoritmo, apenas o termo predominante.

Exemplo: Um algoritmo tem complexidade $T(n) = 3n^2 + 100n$. Nesta função, o segundo termo tem um peso relativamente grande, mas a partir de $n_0 = 11$, é o termo n^2 que "dá o tom" do crescimento da função: uma parábola. A constante 3 também tem uma influência irrelevante sobre a taxa de crescimento da função após um certo tempo. Por isso dizemos que este algoritmo é da ordem de n^2 ou que tem complexidade $O(n^2)$.

Regras para o cálculo

- Repetições

O tempo de execução de uma repetição é o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

Regras para o cálculo

- Repetições aninhadas
  - A análise é feita de dentro para fora
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
  - O exemplo abaixo é $O(n^2)$

```
para i := 0 até n faça
  para j := 0 até n faça
    faça k := k+1;
```

Regras para o cálculo

- Comandos consecutivos
  - É a soma dos tempos de cada um bloco, o que pode significar o máximo entre eles
  - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$

```
para i := 0 até n faça
    k := 0;
para i := 0 até n faça
    para j := 0 até n faça
        faça k := k+1;
```

Regras para o cálculo

- Se... então... senão

  \- Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao então e os comandos relativos ao senão

  \- O exemplo abaixo é O(n)

```
se i < j
então i := i+1
senão para k := 1 até n faça
          i := i*k;
```

Exercício

- Estime quantas unidades de tempo são ecessárias para rodar o algoritmo abaixo

```
Início
  declare i e j numéricos;
  declare A vetor numérico de n posições;
  i := 1;
  enquanto i <= n faça
     A[i] := 0;
     i := i+1;
  para i := 1 até n faça
     para j := 1 até n faça
         A[i] := A[i]+i+j;
Fim
```

Regras para o cálculo

- Chamadas a sub-rotinas

  Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

Regras para o cálculo

- Sub-rotinas recursivas
  - Análise de recorrência
  - *Recorrência: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores*
  - Caso típico: algoritmos de **dividir-e-conquista**r, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original

Regras para o cálculo

- Exemplo de uso de recorrência
  - Calculo Fatorial n!

```
sub-rotina fat(n: numérico)
início
   declare aux numérico;
   aux := 1
   se n = 1
   então aux := 1
   senão aux := n*fat(n-1);
fim
```

Regras para o cálculo

```
sub-rotina fat(n: numérico)
início
  declare aux numérico;
  aux := 1
  se n = 1
  então aux := 1
  senão aux := n*fat(n-1);
fim
```

$$
\begin{aligned}
T(n) &= c+T(n-1) \\
&= 2c+T(n-2) \\
&= \ldots \\
&= nc + T(1) \\
&= O(n)
\end{aligned}
$$

Recorrência-Soluções

$T(n) = T(n-1) + c$ pior caso $\Theta(n)$

$T(n) = 2T(n/2) + n$ pior caso $\Theta(n \lg n)$

$T(n) = T(n-1) + n$ pior caso $\Theta(n^2)$

$T(n) = 2T(n-1) + 1$ pior caso $\Theta(2^n)$

$T(n) = T(n/2) + 1$ pior caso $\Theta(\lg n)$

$T(n) = 2T(n/4) + cn$ pior caso $\theta(n)$

$T(n) = T(3n/4) + c$ pior caso $\theta(\lg n)$