Complexidade de funções

Gonzalo Travieso[1]

2020

[1] gonzalo@ifsc.usp.br

Algumas definições (e restrições)

Funções exponenciais $f(x)$ é exponencial se $f(x) = ab^{cx}$, onde $a, b, c \in \mathbb{R}$ são constantes. No nosso caso, estamos interessados apenas em $a > 0$, $b > 1$ e $c > 0$.

Funções “polinomiais” (leis de potência) $f(x)$ é polinomial se $f(x) = ax^b$, com $a, b \in \mathbb{R}$ constantes. Estamos interessados em $a > 0$ e $b > 0$. Lembre-se que $a\sqrt[b]{x} = ax^{1/b}$ e portanto raizes também são polinomiais.

Funções logaritmicas $f(x)$ é logaritmica se $f(x) = a \log_b x$, com $a, b \in \mathbb{R}$ constantes. Estamos interessados em $a > 0$ e $b > 1$. Quando não indicamos a base b, assume-se que a base é 2.

Misturas Somas, produtos ou divisões arbitrárias de funções (em geral entre as indicadas acima). Por exemplo: $f(x) = 3x \log x$, ou $f(x) = 10e^{x/2}/\sqrt{x}$.

Ordem de funções

- Muitas vezes impossível e inútil derivar expressões precisas para as medidas de desempenho de algoritmos em dada arquitetura.
  - Algortimos são muito complexos
  - Detalhes podem ser desconhecidos
  - Resultado muito dependente da arquitetura
- Análise aproximada é mais simples e mais útil.

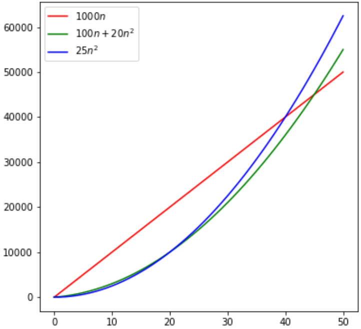
Exemplo

Suponha três programas com tempos de execução (em segundos) dados por (onde n é uma variável que representa o tamanho do problema):

P1 $T_1(n) = 1000n$

P2 $T_2(n) = 100n + 20n^2$

P3 $T_3(n) = 25n^2$



Comparação

- P2 melhor que P3 a partir de
  $T_2(n) < T_3(n) \implies 100 < 5n \implies n > 20.$
- P1 melhor que P3 a partir de
  $T_1(n) < T_3(n) \implies 1000 < 25n \implies n > 40.$
- P1 melhor que P2 a partir de
  $T_1(n) < T_2(n) \implies 900 < 20n \implies n > 45.$
- Para n grande, $T_3(n)/T_2(n) \approx 1.25$ (apenas um fator constante de diferença).
- Para n grande, $T_2(n)/T_1(n) \approx n/50$ (a diferença de desempenho cresce com n).

O-zão

A notação do O-zão (*big-O*) é usada para indicar uma limitação superior no crescimento de uma função. Formalmente temos a seguinte definição:

$O(\cdot)$

Dada uma função $g(x)$ dizemos que uma outra função $f(x)$ satisfaz $f(x) = O(g(x))$ se e somente se $\exists c > 0, x_0 \geq 0$ tais que $f(x) \leq cg(x)$ para todo $x \geq x_0$.

Exemplos

- De acordo com essa definição, se o tempo de execução é $T(n) = O(n^2)$, então sabemos que, para n suficientemente grande, o tempo de execução nunca irá crescer mais rapidamente que n^2 (fora um fator constante).
- Note que o $O(\cdot)$ dá apenas um limitante superior. Portanto, $100n^2 + 1000n = O(n^2)$, mas também $10n = O(n^2)$.
- No nosso exemplo anterior, temos $T_1(n) = O(T_3(n))$ e $T_2(n) = O(T_3(n))$ (descubra os valores de c e n_0 que fazem com que isso seja válido).

$\Omega(\cdot)$

Em alguns casos, queremos descobrir como será o crescimento mínimo do tempo de execução para n grande. Neste caso, usamos a notação $\Omega(\cdot)$, definida como segue.

$\Omega(\cdot)$

Dada uma função $g(x)$ dizemos que uma outra função $f(x)$ satisfaz $f(x) = \Omega(g(x))$ se e somente se $\exists c > 0, x_0 \geq 0$ tais que $f(x) \geq cg(x)$ para todo $x \geq x_0$.

Exemplos

- O limitante inferior dá uma idéia do mínimo necessário para a execução do programa. Muitas vezes esse mínimo é ditado por necessidades do problema ou características da máquina.
- Por exemplo, o tempo médio para encontrar um número em um vetor não-ordenado de n elementos é $\Omega(n)$ (pois em princípio precisamos testar os valores do vetor um a um).
- No nosso exemplo anterior, tempos $T_3(n) = \Omega(T_1(n))$ e $T_2(n) = \Omega(T_1(n))$.

$\Theta(\cdot)$

Por fim, temos a notação $\Theta(\cdot)$, que indica que duas funções são assintoticamente similares com exceção de um fator contante. Formalmente:

$\Theta(\cdot)$

Dada uma função $g(x)$ dizemos que uma outra função $f(x)$ satisfaz $f(x) = \Theta(g(x))$ se e somente se $\exists c_1, c_2 > 0, x_0 \geq 0$ tais que $c_1 g(x) \leq f(x) \leq c_2 g(x)$ para todo $x \geq x_0$.

Exemplos

- A diferença em constantes muitas vezes não é relevante na análise de algoritmos, pois as constantes são muito dependentes da arquitetura[2].
- No nosso exemplo anterior, $T_2(n) = \Theta(T_3(n))$.

[2] Mas tome cuidado! Ela pode ser relevante para o seu programa na sua máquina.

Constantes e fatores de mais baixa ordem

Ao usarmos qualquer uma dessas notações, podemos simplificar a função $g(x)$:

- Mantemos apenas o termo de mais alta ordem na função, pois termos de mais baixa ordem são desprezíveis para x muito grande.
- Desprezamos as constantes que multiplicam os termos restantes, pois as constantes são absorvidas pelas constantes usadas na definição.

Exemplo:

$T(n) = O(100n^2 + 20n\log n + 25)$ significa que $\exists c > 0, n_0 \geq 0$ tais que $T(n) \leq c(100n^2 + 20n\log n + 25)$ para $n \geq n_0$. Mas isto é equivalente a $T(n) \leq c'n^2(1 + 0.2\log n/n + 0.25/n^2)$ onde $c' = 100c$. Como estamos interessados apenas no caso assintótico de n grande, isso é equivalente a $T(n) \leq c'n^2$, o que indica que $T(n) = O(n^2)$ (onde usamos c' como a nova constante).

Algumas propriedades

- $n^a = O(n^b) \iff a \leq b$
- $\log_a(n) = \Theta(\log_b(n)), \forall a, b$
- $a^n = O(b^n) \iff a \leq b$
- $c = O(1), \forall c \in \mathbb{R}$
- $f(n) = O(g(n)) \implies f(n) + g(n) = O(g(n))$
- $f(n) = \Theta(g(n)) \implies f(n) + g(n) = \Theta(g(n)) = \Theta(f(n))$
- $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$.

Diversas complexidades

- Problemas. Ex: Ordenação por troca de posição.
- Algoritmos. Ex: Bubblesort e Quicksort.
- Código. Ex: A implementação de Quicksort da libc GNU.

Podemos falar da complexidade em qualquer deles.

Diversas complexidades

- Pior caso. Ex: Quicksort é $O(n^2)$.
- Caso "normal". Ex: Quicksort é $O(n \log n)$.
- Amortizada.