TEMPO DE EXECUÇÃO DE ALGORITMOS (Cormen, §2.2 e cap. 3)

Considere novamente o algoritmo de busca linear:

```
Algoritmo: busca_linear
Entrada: um vetor A[1..n] e um valor "v".
Saída: um número natural.
---------------------------------------------------
1. i := 1
2. ENQUANTO i <= n
3. | SE A[i] = v
4. | | RETORNE i.
5. | ++i
6. RETORNE i.
```

Para analisarmos quanto tempo demora a execução de um algoritmo como esse na prática, há várias considerações a fazer. Por exemplo:

1. Diferentes máquinas podem executar um mesmo programa em tempos diferentes. Logo, mesmo que soubéssemos o código de máquina exato de um programa, o tempo de execução do programa não dependeria apenas dele próprio.

2. Um mesmo programa, ao ser compilado, pode ser transformado em diferentes códigos-objeto, a depender do compilador, das opções de compilação, etc. Ou seja, um mesmo programa não corresponde a apenas um código-objeto.

3. Um mesmo algoritmo pode ser programado em diferentes linguagens, com diferentes otimizações, etc. Ou seja, um mesmo algoritmo não corresponde apenas a um código-fonte.

Por razões como essas acima, nós tipicamente não tentamos descobrir o tempo de execução exato de um algoritmo na prática. Ao invés disso, há análises que apenas estimam quantas comparações, atribuições e etc são feitas durante a execução de um algoritmo. Entretanto, mesmo tais análises não são levadas à frente na maior parte dos casos em análise de algoritmos, onde os objetivos são tipicamente os seguintes:

1. Comparar diferentes algoritmos.

2. Avaliar se um certo algoritmo é "escalável", isto é, se a performance por ele obtida se mantém à medida em que aumenta o tamanho da entrada recebida.

Para esses fins, uma análise de ORDEM DE CRESCIMENTO do tempo de execução de um algoritmo é frequentemente mais adequada.

CASOS DE EXECUÇÃO

Antes de discutirmos a ordem de crescimento do tempo de execução de algoritmos, é importante observar que mesmo algoritmos determinísticos não executam sempre no mesmo tempo, mesmo para entradas de mesmo tamanho; o algoritmo de busca linear, por exemplo, percorre diferentes extensões do vetor de entrada, dependendo de se e onde o elemento buscado se encontra no vetor.

Assim, pode-se analisar, por exemplo, o MELHOR CASO, o PIOR CASO e o CASO MÉDIO da execução de um algoritmo. Nesta disciplina, nós nos concentraremos na análise de pior caso dos algoritmos, a qual, em particular, nos fornece limites superiores para o tempo de execução de um algoritmo.

ORDEM DE CRESCIMENTO

No pior caso, o algoritmo de busca linear acima executa "n" iterações completas do laço da linha 2, mais um número de instruções antes e depois do laço que não depende de "n". Assim, é correto dizer que, em tais casos, o algoritmo executa

  a*n + b

instruções, para certos valores de "a" e "b" que nós não nos deteremos em avaliar.

Considere, agora, o algoritmo de busca binária:

```
  ========================================================
  Algoritmo: busca_binária
  Entrada: um vetor A[1..n] ORDENADO e um valor "v".
  Saída: um número natural.
  --------------------------------------------------------
  01. a := 1 , b := n
  02. ENQUANTO a <= b
  03. | m := a + (b-a)÷2
  04. | SE v = A[m]
  05. | | RETORNE m.
  06. | SE v < A[m]
  07. | | b := m-1
  08. | SENÃO
  09. | | a := m+1
  10. RETORNE n+1.
  ========================================================
```

Observe que, em uma iteração qualquer do algoritmo acima, o tamanho do trecho do vetor em consideração é "b -a +1", e que, de uma iteração para outra, esse tamanho sempre cai para a metade (ou menos). Em outras palavras, se, em uma iteração, o trecho em consideração tem "x" elementos, então, na iteração anterior (se houver), o trecho correspondente tinha pelo menos 2*x elementos. Como o trecho da última iteração tem pelo menos 1 elemento, então o trecho da penúltima iteração tem pelo menos 2 elementos, e o da antepenúltima 4 elementos. Em geral, o trecho da k-ésima última iteração tem pelo menos 2^(k-1) elementos. Como o vetor possui apenas "n" elementos, então temos:

  2^(k-1) <= n --> k-1 <= lg(n) --> k <= lg(n) + 1.

Nós concluímos portanto que o algoritmo de busca binária executa no máximo lg(n) + 1 iterações. Consequentemente, nós sabemos que o número de instruções executadas pelo algoritmo é no máximo

  c*lg(n) + d .

É importante observar então que a função "n" cresce significativamente mais rapidamente que a função "lg(n)", de forma que, para valores suficientemente grandes de "n", o número de instruções executadas pela busca linear é sempre maior que o número de instruções executadas pela busca binária. Em outras palavras, a ordem de crescimento do tempo de execução de pior caso da busca linear é estritamente maior que a ordem de crescimento do tempo de execução de pior caso da busca binária.

NOTAÇÃO ASSINTÓTICA

As definições a seguir nos permitem realizar de forma precisa e sucinta o tipo de análise feita acima.

1. DEFINIÇÃO: dadas duas funções "f" e "g", com domínio igual aos naturais e
   --------- contradomínio igual aos reais, nós dizemos que f = O(g) sse
   existem um natural n_0 e um real c, ambos positivos, tais que, para todo
   n >= n_0, temos 0 <= f(n) <= c*g(n).

2. EXERCÍCIO: mostre que 5n = O(n).

==========
PROVA:

Sejam c >= 5 e n0 >= 1 (naturais).
Seja n >= n0. Temos:

  0 <= 5*n <= c*n.

Logo, por definição, 5n = O(n), CQD.
==========

3. EXERCÍCIO: mostre que a*n + b = O(n), ∀ "a" e "b" naturais positivos.

==========
PROVA:

Seja c = a + b e n0 = 1.
Seja n >= n0. Temos:

c*n = (a+b)*n
     = a*n + b*n
    >= a*n + b   (pois n >= n0 = 1).

Logo, por definição, a*n + b = O(n).

OBSERVAÇÃO: c = a+1 e n0 = b também teriam sido escolhas suficientes.
==========

Nós dizemos então que o número de instruções executadas pela busca linear no pior caso é O(n), e O(lg n) pela busca binária.

4. EXERCÍCIO: mostre que a*n + b = O(n^2), ∀ "a" e "b" naturais positivos.

==========
PROVA:

Idem exercício anterior (c = a+1 e n0 = b são suficientes).
==========

5. DEFINIÇÃO: dadas duas funções f e g, com domínio igual aos naturais e
   --------- contradomínio igual aos reais, f = Ω(g) sse existem um natural n_0
   e um real c, ambos positivos, tais que, para todo n >= n_0,
   0 <= c*g(n) <= f(n).

6. EXERCÍCIO: mostre que a*n + b = Ω(1), ∀ "a" e "b" naturais positivos.

==========
PROVA:

Sejam f(n) = a*n + b e g(n) = 1. Nós mostraremos que f = Ω(g).
Sejam c = n0 = 1.
Seja n >= n0. Temos:

```
  0 <= c*g(n)
     = 1*1
     = 1
    <= a*n + b   (pois a,b >= 1 e n >= n0 = 1)
     = f(n).
```

Logo, por definição, a*n + b = Ω(1), CQD.
==========

7. EXERCÍCIO: prove ou apresente contraexemplo: n = Ω(5n).

==========
PROVA:

Sejam f(n) = n e g(n) = 5n. Nós mostraremos que f = Ω(g).
Sejam c = 1/5 e n0 = 1.
Seja n >= n0. Temos:

```
  0 <= 1 = n0 <= n = 1/5 * 5*n = c*g(n).
  c*g(n) = n <= n = f(n).
```

Logo, por definição, n = Ω(5n), CQD.
==========

8. EXERCÍCIO (ESSENCIAL): prove ou apresente contraexemplo: n = Ω(n²).

9. EXERCÍCIO: mostre que n^3 != O(n^2).

10. EXERCÍCIO: mostre que n = O(2^n).

11. EXERCÍCIO: mostre que n != Ω(2^n).

 REFERÊNCIAS ADICIONAIS

Clássico em análise de algoritmos, com análises rigorosas de tempo de execução:

  http://www-cs-faculty.stanford.edu/~uno/taocp.html

Análise de algoritmos hoje em dia:

  http://aofa.cs.purdue.edu/