
RECORRÊNCIAS - Aula 2 (Cormen, cap. 4)

Na aula anterior, nós mostramos que o algoritmo de busca binária recursivo executa em tempo O(lg t) obtendo uma expressão não-recorrente para a função

  f(t) = a + f(piso(t/2)), se t > 0
  f(0) = b,
  a,b > 0 (ambas sendo números naturais)

e então mostrando que essa expressão é O(lg t). Nós também podemos chegar a esse resultado sem precisar obter uma expressão não-recorrente exata para "f". PARA TANTO, NÓS FAZEMOS UMA PROVA POR INDUÇÃO DIRETAMENTE:

1. EXERCÍCIO: prove, por indução, que f(t) >= 0, ∀t >= 0.

O exercício acima é a parte mais fácil do que temos que provar, que é que

  0 <= f(n) <= c*(lg n)

para todo n >= m, para algum real "c" e algum natural "m" positivos. A inequação f(n) <= c*(lg n) também é demonstrada por indução, mas, na prática, acontece uma inversão: enquanto, no texto final da demonstração, as constantes "c" e "m" são escolhidas no início da demonstração e só então utilizadas no argumento, ocorre que, na elaboração da demonstração,

  NÓS TENTAMOS PROVAR O PASSO E A BASE DA INDUÇÃO DIRETAMENTE, E, DURANTE ESSE PROCESSO, DESCOBRIMOS AS PROPRIEDADES QUE AS CONSTANTES PRECISAM POSSUIR;

somente depois é que nós escolhemos os valores exatos das constantes, tendo em vista, naturalmente, as propriedades desejadas sobre elas.

2. LEMA: f(t) = O(lg t).

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

Temos que mostrar que existem c,m > 0, sendo "c" real e "m" natural, tais que, para todo n >= m, 0 <= f(n) <= c*(lg n). A primeira inequação segue do exercício anterior, e portanto resta apenas mostrar a segunda.

De fato, sejam m = 2 e c = 2a + b. Observe que, como a,b > 0, então m,c > 0. Assim, nós mostraremos abaixo que, ∀n >= m, f(n) <= c*(lg n), e nós o faremos por indução em "n":

  * BASE (2 <= n <= 3): nesse caso, temos:

      f(n) = 2a + b
            = c
           <= c*lg(n), CQD.   ---> pois lg(2) = 1 e lg(3) >= 1.

```
  * HIPÓTESE DE INDUÇÃO (H.I.): para todo 2 <= n' < n, f(n') <= c*(lg n').

  * PASSO (4 <= n): temos:

      f(n)  = a + f(piso(n/2))
            <= a + c*(lg piso(n/2))    ---> pela H.I., já que 2 <= piso(n/2) < n.
            <= a + c*(lg (n/2))        ---> lg é função crescente.
             = a + c*((lg n) - (lg 2))
             = a + c*((lg n) - 1)
             = a + c*(lg n) - c
             = c*(lg n) + (a - c)
            <= c*(lg n), CQD.          ---> pois a - c <= 0, já que c >= a.
```

==========

Observe que, na demonstração acima, as propriedades necessárias sobre "c" e "m" são as seguintes:

```
  * m >= 2         ---> como lg(1) = 0, não existe "c" tal que f(1) <= c*lg(1).
  * c >= 2a + b    ---> necessário na base  da indução.
  * c >= a         ---> necessário no passo da indução.
```

Na prática, essas propriedades foram descobertas em primeiro lugar, durante o desenvolvimento do argumento da indução, e só depois os valores exatos de "c" e "m" foram escolhidos.

RESOLVENDO PEQUENOS ENTRAVES NA INDUÇÃO -- PROVAR RESULTADO MAIS FORTE

Por vezes, a estratégia acima para a determinação de limites assintóticos não funciona imediatamente, devido ao argumento indutivo não servir para demonstrar o resultado desejado de forma direta; uma pequena modificação no argumento pode, porém, levar ao resultado desejado.

De fato, isso já foi exemplificado acima, onde foi necessário fazer a base da indução para n = 2 ou n = 3, e não n = 1, como se pretendia inicialmente. Em geral, se o resultado não é verdadeiro para certos valores pequenos de "n", nós sempre podemos tentar provar a base indutiva para valores maiores, tendo, porém, o cuidado de garantir que a base prova a hipótese de indução para todos os valores de "n" requeridos no passo indutivo e que não seguem pela argumentação do passo (acima, por exemplo, foi necessário incluir o caso "n = 3" na base).

Há casos, porém, em que é o passo indutivo que não procede. Nesses casos, uma alternativa é modificar adequadamente a tese a ser provada por indução; pode-se, por exemplo, tornar o resultado ligeiramente mais forte.

De fato, considere o seguinte algoritmo:

```
  ============================================================
  Algoritmo: soma_rec
  Entrada: vetor V[1..n] de números reais, índices "i" e "f".
  Saída: um número real.
  ------------------------------------------------------------
  1. SE f < i
  2. | RETORNE 0.
  3. SE i = f
  4. | RETORNE V[i].
  5. m := i + ⌊(f-i)/2⌋
  6. RETORNE soma_rec(V,i,m) + soma_rec(V,m+1,f).
  ============================================================
```

Para estimar o tempo de execução desse algoritmo, é útil descobrir os tamanhos dos intervalos sobre os quais operam as duas chamadas recursivas, em relação ao tamanho "t" -- mais precisam., t(i,f) -- do intervalo da chamada original, onde

  t(i,f) = f -i +1, se i <= f,
  t(i,f) = 0,   em caso contrário.

3. OBSERVAÇÃO: existe "x" tal que -⌊x⌋ != ⌊-x⌋!

==========
PROVA: -⌊2,5⌋ = -2 != -3 = ⌊-2,5⌋.
==========

4. LEMA: na linha 6 do algoritmo acima, m -i +1 = ⌈t/2⌉ e f-m = ⌊t/2⌋.

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

Observe primeiramente que, na linha 6 do algoritmo, t > 1. Agora, nós provaremos o resultado em cada um dos casos possíveis:

  * CASO 1: t = 2u, para algum u > 0 natural. Nesse caso, temos:

      * m -i +1 = i + ⌊(f-i)/2⌋ -i +1
                = ⌊(f-i)/2⌋ + 1
                = ⌊(f-i)/2 + 1⌋
                = ⌊(f -i +2)/2⌋
                = ⌊(f -i +1 +1)/2⌋
                = ⌊(t + 1)/2⌋
                = ⌊(2u + 1)/2⌋
                = ⌊u + 1/2⌋
                = u
                = ⌈u⌉
                = ⌈t/2⌉, CQD.

      * f-m = f - (i + ⌊(f-i)/2⌋)
            = f -i -⌊(f-i)/2⌋
            = f -i +1 -1 -⌊(f -i +1 -1)/2⌋
            = 2u -1 - ⌊(2u -1)/2⌋
            = 2u -1 - ⌊u - 1/2⌋
            = 2u -1 - (u - 1)
            = 2u -1 -u +1
            = u
            = ⌊u⌋
            = ⌊t/2⌋, CQD.

  * CASO 2: t = 2u + 1, para algum u > 0 natural. Nesse caso, temos:

      * m -i +1 = i + ⌊(f-i)/2⌋ -i +1
                = ⌊(f-i)/2⌋ +1
                = ⌊(f-i)/2 +1⌋
                = ⌊(f-i +2)/2⌋
                = ⌊(f-i +1 +1)/2⌋
                = ⌊(t +1)/2⌋
                = ⌊(2u +1 +1)/2⌋
                = ⌊(2u +2)/2⌋
                = ⌊u + 1⌋
                = u + 1
                = ⌈u + 1/2⌉
                = ⌈(2u + 1)/2⌉
                = ⌈t/2⌉, CQD.

```
      * f-m = f - (i + ⌊(f-i)/2⌋)
            = f -i - ⌊(f-i)/2⌋
            = f -i +1 -1 - ⌊(f -i +1 -1)/2⌋
            = t -1 - ⌊(t -1)/2⌋
            = 2u +1 -1 - ⌊(2u +1 -1)/2⌋
            = 2u - ⌊2u/2⌋
            = 2u - ⌊u⌋
            = 2u - u
            = u
            = ⌊u + 1/2⌋
            = ⌊(2u + 1)/2⌋
            = ⌊t/2⌋, CQD.
```

==========

Pelo lema acima, concluímos que o tempo de execução do algoritmo soma_rec, para um intervalo de tamanho "t", é limitado superiormente por g(t), onde

```
  g(t) = a + g(⌈t/2⌉) + g(⌊t/2⌋), se t >= 2,
  g(t) = b,                       se 0 <= t <= 1,
```

e "a" e "b" são naturais positivos grandes o suficiente.

Assim sendo, nós já sabemos que uma maneira de obter um limite superior assintótico para o tempo de execução de soma_rec é obter uma expressão não-recorrente exata para g(t):

5. EXERCÍCIO (opcional): obtenha uma expressão não-recorrente exata para g(t).

Aqui, porém, nós tentaremos obter um limite superior assintótico sem passar pela obtenção de uma expressão não-recorrente exata para g(t). Nesse sentido, é útil observar que, apesar da forma incomum do algoritmo, tudo o que ele faz é somar os elementos do vetor. Assim, é plausível cogitar que ele execute em tempo linear:

6. EXERCÍCIO: prove que g(t) >= 0, ∀t >= 0.

7. LEMA: g(t) = O(t).

==========

PROVA:

Temos que mostrar que existem um real "c" e um natural "m" positivos tais que, ∀n >= m, 0 <= g(t) <= c*t.

Sejam então m = 1 e c = a + b.

Nós mostraremos agora que 0 <= g(t) <= c*t, ∀n >= m. De fato, a inequação 0 <= g(t) segue do exercício anterior. Logo, resta apenas provar a segunda inequação, e nós o faremos por indução em "n":

  * TESE: ∀n >= m, g(n) <= c*n - a.

  * BASE (n = 1): temos:

```
      g(1)  = b
           <= c - a          ---> pois c >= a + b
            = c*1 - a, CQD.
```

  * HIPÓTESE DE INDUÇÃO (H.I.): ∀ 1 <= l < n, g(l) <= c*l - a.

  * PASSO (n >= 2): temos:

      g(n) = a + g(⌈n/2⌉) + g(⌊n/2⌋)
            <= a + c*⌈n/2⌉ -a + c*⌊n/2⌋ -a ---> H.I., pois 1 <= ⌈n/2⌉,⌊n/2⌋ < n.
             = c*(⌈n/2⌉ + ⌊n/2⌋) - a
             = c*n - a, CQD.

Pela demonstração acima, temos que, ∀n >= m, g(n) <= c*n - a <= c*n, CQD.

==========

 RESOLVENDO PEQUENOS ENTRAVES NA INDUÇÃO -- APROVEITAR DIFERENÇAS

Considere esta versão do algoritmo de ordenação por entrelaçamento (mergesort):

```
  ============================================================
  Algoritmo: mergesort
  Entrada: vetor V[1..n], índices "i" e "f".
  Saída: nenhuma.
  ------------------------------------------------------------
  1. SE i < f
  2. | m := i + ⌊(f-i)/2⌋   // = ⌊i + (f-i)/2⌋ = ⌊(i+f)/2⌋.
  3. | mergesort(V,i,m)
  4. | mergesort(V,m+1,f)
  5. | entrelaçar(V,i,m,f)  // executa em tempo θ(t).
  ============================================================
```

É fácil perceber que, sendo

```
  t(i,f) = f -i +1, se i <= f,
  t(i,f) = 0,       em caso contrário,

  h(t) = a + h(⌈t/2⌉) + h(⌊t/2⌋) + b*t, se t >= k,
  h(t) = d,                             se 0 <= t < k,

  a,b,d,k naturais positivos grandes o suficiente,
```

então h(t) limita superiormente o tempo de execução de uma chamada mergesort(V,i,f).

8. EXERCÍCIO ESSENCIAL: mostre que h(n) = O(n²).