
PROGRAMAÇÃO DINÂMICA (Cormen, §15.1)

O seguinte é um algoritmo direto -- e que podemos entender como do gênero de divisão-e-conquista -- para calcular o n-ésimo número de Fibonacci:

```
==========================================================================
Algoritmo:    fib
Entrada:      um natural "n".
Pré-condição: n >= 1.
Saída:        um natural "x".
Pós-condição: "x" é o n-ésimo número de Fibonacci.
--------------------------------------------------------------------------
1. SE n <= 2
2. | RETORNE 1.
3. RETORNE fib(n-2) + fib(n-1).
==========================================================================
```

1. EXERCÍCIO: observando que, a cada aumento de duas unidades em "n", fib(n)
   --------- (mais que) duplica de valor, mostre que o algoritmo acima executa
   em tempo Ω(2^⌊n/2⌋).

O problema com o algoritmo acima é óbvio -- ele repete cálculos --, e a solução também -- memorizar o resultado de cálculos já realizados. Isso nos leva ao seguinte algoritmo:

```
==========================================================================
Algoritmo:    fib_it  // fib iterativo
Entrada:      um natural "n".
Pré-condição: n >= 1.
Saída:        um natural "x".
Pós-condição: "x" é o n-ésimo número de Fibonacci.
--------------------------------------------------------------------------
1. SE n <= 2
2. | RETORNE 1.
3. fibim2 := 1 , fibim1 := 1  // "fib i menos 2", "fib i menos 1".
4. PARA i DE 3 A n
5. | fibi := fibim2 + fibim1 , fibim2 := fibim1 , fibim1 := fibi
6. RETORNE fibi.
==========================================================================
```

2. OBSERVAÇÃO: observe que é fácil provar a correção total do algoritmo acima
   ---------- por meio de invariantes e variantes, bem como o fato de que ele
   executa em tempo θ(n).

O algoritmo acima embute a essência da técnica de PROGRAMAÇÃO DINÂMICA: resolver uma instância "grande" pela combinação das soluções de instâncias "menores" (assim como na divisão-e-conquista), MAS ARMAZENANDO E REUTILIZANDO OS RESULTADOS DE CÁLCULOS INTERMEDIÁRIOS que sejam usados na solução de mais de um subproblema.

MEMOIZAÇÃO

Pode-se argumentar que um algoritmo de programação dinâmica, apesar de executar mais rapidamente que um algoritmo de divisão-e-conquista que repita cálculos, não é igualmente claro. A técnica da MEMOIZAÇÃO tenta juntar o melhor de cada abordagem, mantendo a forma recursiva do algoritmo e também um registro dos cálculos já realizados. A seguinte é, por exemplo, uma versão "memoizada" do algoritmo de Fibonacci:

```
Algoritmo:    fib_memo
Entrada:      um natural "n".
Pré-condição: n >= 1.
Saída:        um natural "x".
Pós-condição: "x" é o n-ésimo número de Fibonacci.
```

```
1. SE n <= 2
2. | RETORNE 1.
3. Obtenha vetor auxiliar A[1..n] de inteiros.
4. A[1] := A[2] := 1
5. PARA i DE 3 A n
6. | A[i] := 0
7. RETORNE fib_rec(n,A).
```

```
Algoritmo:    fib_rec
Entrada:      um inteiro "n" e um vetor "A" de inteiros.
Pré-condição: (1) n >= 1;
              (2) existe 2 <= k <= n tal que,
                  para todo 1 <= j <= k, A[j] = fib(j), e tal que
                  para todo k <  j <= n, A[j] = 0.
Saída:        um inteiro "x".
Pós-condição: (1) "x" é o n-ésimo número de Fibonacci;
              (2) para todo 1 <= j <= n, A[j] = fib(j).
```

```
1. SE A[n] = 0
2. | A[n] := fib_rec(n-2,A) + fib_rec(n-1,A)
3. RETORNE A[n].
```

3. EXERCÍCIO (opcional): como você argumentaria que fib_memo também executa em tempo O(n)?

Embora não evidente no exemplo acima, a técnica de memoização possui a vantagem de somente fazer os cálculos que são realmente necessários para se obter a solução desejada, ao passo que a técnica de programação dinâmica "padrão" (ou "de baixo para cima", "bottom-up", que vai diretamente da solução de instâncias pequenas para a solução das instâncias maiores) por vezes envolve a solução de instâncias intermediárias que não são efetivamente utilizadas na obtenção da solução final. Por outro lado, quando todos os cálculos intermediários são utilizados, a solução direta "de baixo para cima" geralmente leva a um algoritmo um pouco mais rápido (por um fator constante), pois nela os subproblemas são "atingidos" somente uma vez, e não duas como na memoização (uma "de cima para baixo", e outra "de baixo para cima"), e além disso não há o trabalho de inicializar a tabela com valores indicando "cálculo ainda não realizado".

A técnica de programação dinâmica pode então ser assim resumida:

  a) Solução de instâncias grandes por meio da solução de instâncias menores, como na divisão-e-conquista.

  b) Se as soluções das instâncias menores compartilham cálculos, então aplicar divisão-e-conquista diretamente leva a cálculos repetidos; a ideia, portanto, é contornar essa repetição, resolvendo do pequeno para o grande, ao invés de do grande para o pequeno.

  c) Caso desejemos, nós ainda podemos manter a forma recursiva do algoritmo; para tanto, porém, nós devemos manter um registro dos cálculos que já foram feitos, e só fazer um cálculo se ele ainda não tiver sido feito (MEMOIZAÇÃO).

 O PROBLEMA DA LINHA DE MONTAGEM (Cormen, 2ª edição, §15.1)

Veja a definição completa do problema no Cormen (2ª edição):

  http://books.google.com.br/books?id=NLngYyWFl_YC&pg=PA324&lpg=PA324&dq=assembly+line+problem+dynamic+programming&source=bl&ots=ByOmHH1jJa&sig=-u4NpvBcN4DXCegSh8PE-5oM1qo&hl=pt-BR&sa=X&ei=JPwWVL_8NeHFigKmuYGIBQ&sqi=2&ved=0CFgQ6AEwBw .

Os dados de entrada do problema são:

  * n: número de setores de cada uma das duas linhas de montagem.

  * a_{i j}: tempo gasto no j-ésimo setor da i-ésima linha de montagem (a: "assembly").

  * e_i: tempo gasto na entrada da i-ésima linha de montagem (e: "entry").

  * t_{i j}: tempo gasto para transferir a partir do j-ésimo setor da linha de montagem "i" (t: "transfer").

  * x_i: tempo gasto na saída da i-ésima linha de montagem (x: "exit").

Além disso, nós vamos calcular:

  * M_{i j}: melhor tempo para se chegar ao j-ésimo setor da i-ésima linha de montagem.

Observe que o problema pode ser resolvido em tempo Ω(2^n) por meio de uma enumeração explícita dos 2^n caminhos possíveis para o chassi na fábrica. É também possível resolver o problema por meio do algoritmo de caminhos mínimos de Dijkstra, em tempo O(n*lg n). O algoritmo abaixo, porém, utiliza a técnica de programação dinâmica para resolver o problema em tempo θ(n), que é ótimo, dado que a entrada do problema consiste em θ(n) números.

```
  ================================================================================
  Algoritmo: linha_de_montagem
  Entrada:   os dados "n", "a", "e", "t" e "x" acima.
  Saída:     um número real // o menor tempo de montagem completa.
  --------------------------------------------------------------------------------
  1. M_{1 1} := e_1 , M_{2 1} := e_2
  2. PARA j DE 2 A n
  3. | M_{1 j} := mín{ M_{1 j-1} + a_{1 j-1} ,
     |                M_{2 j-1} + a_{2 j-1} + t_{2 j-1} }
  4. | M_{2 j} := mín{ M_{2 j-1} + a_{2 j-1} ,
     |                M_{1 j-1} + a_{1 j-1} + t_{1 j-1} }
  5. RETORNE mín{ M_{1 n} + a_{1 n} + x_1 , M_{2 n} + a_{2 n} + x_2 }.
  ================================================================================
```

4. EXERCÍCIOS:
   ---

   a) O algoritmo acima utiliza uma matriz auxiliar M de 2 linhas e "n" colunas, ou seja, utiliza $\theta(n)$ de memória auxiliar. É possível diminuir esse uso de memória para $O(1)$?

   b) Escreva uma variação do algoritmo acima que retorne, além do tempo de montagem mínimo, também um vetor l[1..n] que informa, para cada setor j, a linha de montagem l[j] cuja estação de montagem é utilizada no setor j da solução ótima (logo, para todo j, l[j] vale 1 ou 2).

   c) Generalize o algoritmo acima para o caso de "m" linhas de montagem, ao invés de apenas duas.

   d) Escreva uma versão memoizada do algoritmo acima.

O PROBLEMA DO CORTE DE HASTE (Cormen, 3ª ed., §15.1)

---

5. EXERCÍCIO (ESSENCIAL): escreva um algoritmo eficiente para encontrar a melhor
   ---------- maneira de cortar uma haste de tamanho "n" em pedaços, de forma a maximizar o lucro obtido. Tanto o tamanho da haste original quanto o de cada pedaço final é um número natural, e, para todo "i" de 1 a "n", um pedaço de tamanho "i" tem preço p[i].