"Para fazer um procedimento recursivo é preciso ter fé."
— prof. Siang Wun Song

"Ao tentar resolver o problema, encontrei obstáculos dentro de obstáculos.
Por isso, adotei uma solução recursiva."
— aluno S.Y., 1998

"To understand recursion,
we must first understand recursion."
— anônimo

Recursão e algoritmos recursivos

Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm *estrutura recursiva*. Para resolver um tal problema podemos aplicar o seguinte método:

- se a instância em questão for pequena,
  resolva-a diretamente (use força bruta se necessário);
- senão,
  reduza-a a uma instância menor do mesmo problema,
  aplique o método à instância menor e
  volte à instância original.

A aplicação desse método produz um algoritmo *recursivo*. Para mostrar como isso funciona, examinaremos um exemplo concreto.

Um exemplo

Considere o seguinte problema: *Determinar o valor de um elemento máximo de um vetor* `v[0..n-1]`.

É claro que o problema só faz sentido se o vetor não é vazio, ou seja, se `n ≥ 1`. Para preparar o terreno, examine uma tradicional solução iterativa do problema:

```
int maximo( int n, int v[]) {
   int j, x;
   x = v[0];
   for (j = 1; j < n; j += 1)
      if (x < v[j]) x = v[j];
   return x;
}
```

Exercícios 1

1. Considere a função iterativa `maximo` acima. Faz sentido trocar "`x = v[0]`" por "`x = 0`", como fazem alguns programadores descuidados? Faz sentido trocar "`x = v[0]`" por "`x = INT_MIN`"? Faz sentido trocar "`x < v[j]`" por "`x <= v[j]`"? [Solução parcial]
2. A função abaixo promete encontrar o valor de um elemento máximo de `v[0..n-1]`. A função cumpre a

promessa?

```
int maxi( int n, int v[]) {
   int j, m = v[0];
   for (j = 1; j < n; ++j)
      if (v[j-1] < v[j]) m = v[j];
   return m;
}
```

Solução recursiva do problema

Eis uma função recursiva do problema da seção anterior:

```
int
maximoR( int n, int v[])
{
   if (n == 1)
      return v[0];
   else {
      int x;
      x = maximoR( n-1, v);  /* máximo de v[0..n-2] */
      if (x > v[n-1])
         return x;
      else
         return v[n-1];
   }
}
```

A análise do algoritmo tem a mesma forma que uma prova por indução. Se `n` vale `1` então `v[0]` é o único elemento relevante do vetor e portanto `v[0]` é o máximo. Agora suponha que `n` vale mais que `1`. Então nosso vetor tem duas partes: `v[0..n-2]` e `v[n-1]` e portanto o valor que procuramos é o maior dentre

`v[n-1]` e um máximo de `v[0..n-2]`.

(Eis um roteiro que pode ajudar a verificar se a função está correta: 1. Escreva *o que* a função deve fazer. 2. Verifique que a função de fato faz o que deveria quando `n` é pequeno, ou seja, quando `n` vale `1`. 3. Agora imagine que `n` é grande, ou seja, `n > 1`, e verifique que a função faz a coisa certa *supondo que faria a coisa certa se no lugar de* `n` *tivéssemos algo menor que* `n`.)

Para que uma função recursiva seja compreensível, é muito importante que o autor da função diga, explicitamente, *o que* a função faz. Portanto, eu deveria escrever o seguinte comentário antes do código:

```
/* Ao receber v e n >= 1, a função devolve o valor */
/* de um elemento máximo do vetor v[0..n-1].       */
```

Desempenho

Algumas pessoas acreditam que funções recursivas são inerentemente ineficientes e lentas, mas isso não passa de lenda. Talvez a lenda tenha origem em usos descuidadas da recursão, como num dos exercícios abaixo. (Nem tudo são flores, entretanto. É preciso lembrar do espaço de memória que a pilha de recursão consome.)

(A pergunta "Como o computador executa um algoritmo recursivo?", embora muito relevante, será ignorada por enquanto. Veja a página sobre pilhas.)

Exercícios 2

1. Critique a seguinte função recursiva; ela promete encontrar o valor de um elemento máximo de `v[0..n-1]`.

```
int maximoR1( int n, int v[]) {
   int x;
   if (n == 1) return v[0];
   if (n == 2) {
      if (v[0] < v[1]) return v[1];
      else return v[0];
   }
   x = maximoR1( n-1, v);
   if (x < v[n-1]) return v[n-1];
   else return x;
}
```

2. Critique a seguinte função recursiva; ela promete encontrar o valor de um elemento máximo de `v[0..n-1]`.

```
int maximoR2( int n, int v[]) {
   if (n == 1) return v[0];
   if (maximoR2( n-1, v) < v[n-1]) return v[n-1];
   else return maximoR2( n-1, v);
}
```

3. Escreva uma função recursiva `maxmin` que calcule o valor de um elemento máximo e o valor de um elemento mínimo de um vetor `v[0..n-1]`. Quantas comparações envolvendo os elementos do vetor sua função faz?

4. PROGRAMA DE TESTE. Escreva um pequeno programa para testar a função recursiva `maximoR`. O seu programa deve pedir ao usuário que digite uma sequência de números e em seguida devolver o valor do maior dos números digitados. [Solução] Agora faça uma nova versão do programa para determinar um elemento máximo de um vetor aleatório. Acrescente ao seu programa uma função que *confira a resposta* dada por `maximoR`. [Solução]

5. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor de inteiros `v[0..n-1]`. O problema faz sentido quando `n` é igual a `0`? Quanto deve valer a soma nesse caso? [Solução]

Outra solução recursiva

A função `maximoR` discutida acima aplica a recursão ao subvetor `v[0..n-2]`. É possível escrever uma versão que aplique a recursão ao subvetor `v[1..n-1]`:

```
/* Ao receber v e n >= 1, esta função devolve */
/* o valor de um elemento máximo do vetor     */
/* v[0..n-1].                                 */

int
maximo2( int n, int v[])
{
   return maxR( 0, n, v);
}

/* Recebe v, ini e fim tais que ini < fim. */
/* Devolve o valor de um elemento máximo   */
/* do vetor v[ini..fim-1].                 */

int
maxR( int ini, int fim, int v[])
{
   if (ini == fim-1) return v[ini];
   else {
      int x;
      x = maxR( ini + 1, fim, v);
      if (x > v[ini]) return x;
      else return v[ini];
   }
}
```

Observe que `maximo2` é apenas uma "embalagem": o serviço pesado é executado pela função recursiva `maxR`. A função `maxR` resolve um problema *mais geral* que o original. A necessidade de *generalizar* o

problema ocorre com frequência durante o projeto de algoritmos recursivos.

A título de curiosidade, eis outra maneira, talvez surpreendente, de aplicar recursão ao subvetor `v[1..n-1]`. Ela usa aritmética de endereços:

```
int maximo2R( int n, int v[]) {
   int x;
   if (n == 1) return v[0];
   x = maximo2R( n - 1, v + 1);
   if (x > v[0]) return x;
   return v[0];
}
```

Exercícios 3

1. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor `v[ini..fim-1]`. O problema faz sentido quando `ini` é igual a `fim`? Quanto deve valer a soma nesse caso?
2. Escreva uma função recursiva que calcule a soma dos dígitos de um inteiro positivo `n`. A soma dos dígitos de `132`, por exemplo, é `6`.
3. Escreva uma função recursiva que calcule o piso do logaritmo de *N* na base 2. (Veja uma versão iterativa do exercício.)

Exercícios 4

1. Qual o valor de `X( 4)`? [Solução]

   ```
   int X( int n) {
      if (n == 1 || n == 2) return n;
      else return X( n-1) + n * X( n-2);
   }
   ```

2. Qual é o valor de `f( 1,10)`? Escreva uma função equivalente que seja mais simples.

   ```
   double f( double x, double y) {
      if (x >= y) return (x + y)/2;
      else return f( f( x+2, y-1), f( x+1, y-2));
   }
   ```

3. Qual o resultado da execução do programa abaixo?

   ```
   int ff( int n) {
      if (n == 1) return 1;
      if (n % 2 == 0) return ff( n/2);
      return ff( (n-1)/2) + ff( (n+1)/2);
   }
   int main( void) {
      printf( "%d", ff(7));
      return EXIT_SUCCESS;
   }
   ```

4. Execute `fusc( 7,0)`.

   ```
   int fusc( int n, int profund) {
     int i;
     for (i = 0; i < profund; ++i)
        printf( "  ");
     printf( "fusc(%d,%d)\n", n, profund);
     if (n = 1)
        return 1;
     if (n % 2 == 0)
        return fusc( n/2, profund+1);
     return fusc( (n-1)/2, profund+1) + fusc( (n+1)/2, profund+1);
   }
   ```

5. Critique a seguinte função recursiva:

```
int XX( int n) {
   if (n == 0) return 0;
   else return XX( n/3+1) + n;
}
```

6. FIBONACCI. A função de Fibonacci é definida assim: `F(0) = 0`, `F(1) = 1` e `F(n) = F(n-1) + F(n-2)` para `n > 1`. Descreva a função `F` em linguagem C. Faça uma versão iterativa e uma recursiva.
7. Seja `F` a versão recursiva da função de Fibonacci. O cálculo do valor da expressão `F(3)` provocará a seguinte sequência de invocações da função:

```
F(3)
  F(2)
    F(1)
    F(0)
  F(1)
```

   Qual a sequência de invocações da função provocada por `F(5)`?
8. EUCLIDES. A seguinte função calcula o maior divisor comum dos inteiros positivos `m` e `n`. Escreva uma função recursiva equivalente.

```
int Euclides( int m, int n) {
   int r;
   do {
      r = m % n;
      m = n;
      n = r;
   } while (r != 0);
   return m;
}
```

9. EXPONENCIAÇÃO. Escreva uma função recursiva eficiente que receba inteiros positivos k e n e calcule k^n. (Suponha que k^n cabe em um `int`.) Quantas multiplicações sua função executa aproximadamente?

Veja o verbete *Recursion* na Wikipedia

Veja bons exemplos de recursão no capítulo sobre algoritmos de enumeração

---

Last modified: Sat Mar 21 11:59:01 BRT 2015
http://www.ime.usp.br/~pf/algoritmos/
Paulo Feofiloff
DCC-IME-USP