5. Recursividade

"E não sabendo que era impossível, foi lá e fez."
Jean Cocteau

Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular. Assim, pode-se dizer que o conceito de algo recursivo está dentro de si, que por sua vez está dentro de si e assim sucessivamente, infinitamente.

O exemplo a seguir define o ancestral de uma pessoa:

- Os pais de uma pessoa são seus ancestrais (caso base);
- Os pais de qualquer ancestral são também ancestrais da pessoa inicialmente considerada (passo recursivo).

Definições como estas são normalmente encontradas na matemática. O grande apelo que o conceito da recursão traz é a possibilidade de dar uma definição finita para um conjunto que pode ser infinito [15]. Um exemplo aritmético:

- O primeiro número natural é zero.
- O sucessor de um número natural é um número natural.

Na computação o conceito de recursividade é amplamente utilizado, mas difere da recursividade típica por apresentar uma condição que provoca o fim do ciclo recursivo. Essa condição deve existir, pois, devido às limitações técnicas que o computador apresenta, a recursividade é impedida de continuar eternamente.

5.1. Função para cálculo de Fatorial

Na linguagem C, as funções podem chamar a si mesmas. A função é recursiva se um comando no corpo da função a chama. Para uma linguagem de computador ser recursiva, uma função deve poder chamar a si mesma. Um exemplo simples é a função fatorial, que calcula o fatorial de um inteiro. O fatorial de um número N é o produto de todos os números inteiros entre `1` e `N`. Por exemplo, 3 fatorial (ou `3!`) é `1 * 2 *3 = 6`. O programa 5.1 apresenta uma versão iterativa para cálculo do fatorial de um número.

Programa 5.1: Fatorial (versão iterativa)

```
1  int fatorialc ( int n )
   {
       int t, f;
       f = 1;
       for ( t = 1; t<=n; t++ )
6          f = f * t
       return f;
   }
```

Mas multiplicar `n` pelo produto de todos os inteiros a partir de `n-1` até 1 resulta no produto de todos os inteiros de `n` a 1. Portanto, é possível dizer que fatorial:

- 0! = 1
- 1! = 1 * 0!
- 2! = 2 * 1!
- 3! = 3 * 2!
- 4! = 4 * 3!

Logo o fatorial de um número também pode ser definido recursivamente (ou por recorrência) através das seguintes regras (representação matemática) [15, 1]:

- n! = 1, se n = 0
- n! = n * (n-1)! , se n > 0

O programa 5.2 mostra a versão recursiva do programa fatorial.

Programa 5.2: Fatorial (versão recursiva)

```
int fatorialr( int n)
{
    int t, f;
    /* condição de parada */
    if( n == 1 || n == 0)
    {
        return 1;
    }
    f = fatorialr(n-1)*n; /* chamada da função */
    return f;
}
```

A versão não-recursiva de fatorial deve ser clara. Ela usa um laço que é executado de 1 a `n` e multiplica progressivamente cada número pelo produto móvel.

A operação de fatorial recursiva é um pouco mais complexa. Quando `fatorialr` é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto de `fatorialr(n-1)*n`. Para avaliar essa expressão, `fatorialr` é chamada com `n-1`. Isso acontece até que `n` se iguale a 1 e as chamadas à função comecem a retornar.

Calculando o fatorial de 2, a primeira chamada a `fatorialr` provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original e `n`). A resposta então é 2.

Para melhor entendimento, é interessante ver como o programa é executado internamente no computador. No caso do programa iterativo (programa 5.1) é necessário duas variáveis `f` e `t` para armazenar os diversos passos do processamento. Por exemplo, ao calcular fatorial de 6, o computador vai passar sucessivamente pelos seguintes passos (tabela 5.1).

Tabela 5.1: Cálculo de fatorial de 6

t	f
1	1
2	2
3	6
4	24
5	120
6	720

No programa recursivo (5.2) nada disto acontece. Para calcular o fatorial de 6, o computador tem de calcular primeiro o fatorial de 5 e só depois é que faz a multiplicação de 6 pelo resultado (120). Por sua vez, para calcular o fatorial de 5, vai ter de calcular o fatorial de 4. Resumindo, aquilo que acontece internamente é uma expansão seguida de uma contração:

```
fatorialr(6)
6 * fatorialr(5)
6 * 5 * fatorialr(4)
6 * 5 * 4 * fatorialr(3)
6 * 5 * 4 * 3 * fatorialr(2)
6 * 5 * 4 * 3 * 2 * fatorialr(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada à função dentro da função.

5.2 Número triangular

Pitágoras, matemático e filósofo grego, demonstrou várias propriedades matemáticas, entre elas a propriedade dos números triangulares. Um número triangular é um número natural que pode ser representado na forma de triângulo equilátero. Para encontrar o *n-ésimo* número triangular a partir do anterior basta somar-lhe *n* unidades. Os primeiros números triangulares são 1, 3, 6, 10, 15, 21, 28. O *n-ésimo* termo pode ser descoberto pela fórmula a seguir:

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2} [17]$$

Estes números são chamados de triangulares pois podem ser visualizados como objetos dispostos na forma de um triângulo (figura 5.1).

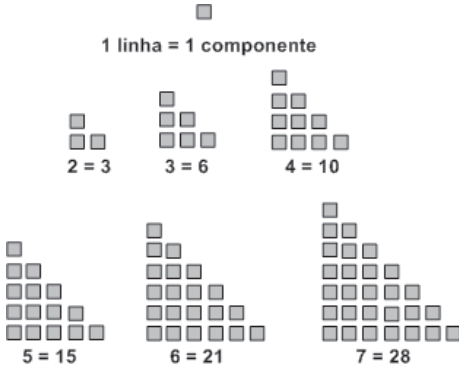

Figura 5.1: Números triangulares

Supondo que se esteja buscando o quinto elemento (dado pelo número 15), como descobrir este elemento? Basta distribuir entre as linhas e colunas conforme a figura 5.2 (5+4+3+2+1 = 15).

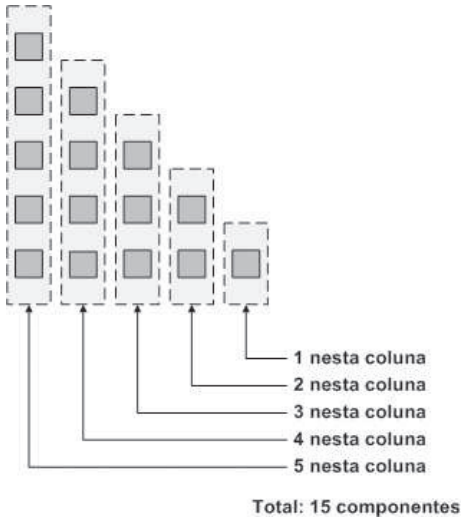

Figura 5.2: Descobrindo o quinto elemento triangular

Este é um processo repetitivo, dado por um programa simples (programa 5.3).

Programa 5.3: Descobrindo o número triangular (iterativo)

```
int triangulo(int n)
{
  int iTotal = 0;
  while( n > 0 )
  {
    iTotal += n;
    n--;
  }
  return iTotal;
}
```

Este é um processo recursivo [8] (figura 5.3) , pois:

1. Primeira coluna tem *n* elementos.
2. Soma-se a próxima coluna com *n*-1 elementos até que reste apenas 1 elemento.

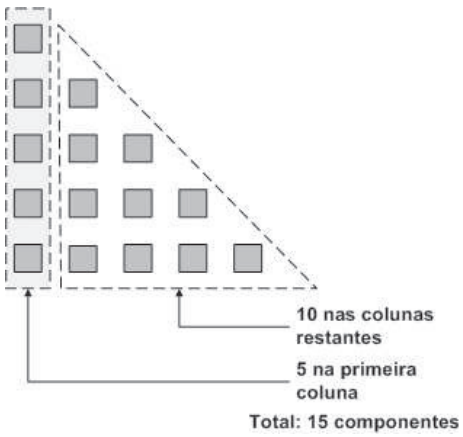

Figura 5.3: Descobrindo o quinto elemento triangular de forma recursiva

O programa 5.4 implementa a solução recursiva do problema. A figura 5.4 demostra o que ocorre a cada chamada da função *triangulo*, nela pode ser observado o retorno de cada execução da função.

Programa 5.4: Descobrindo o número triangular (recursivo)

```
int triangulo(int n)
{
    if( n == 1 )
```

```
    {
5       return n;
    }
    return n + triangulo(n-1);
}
```

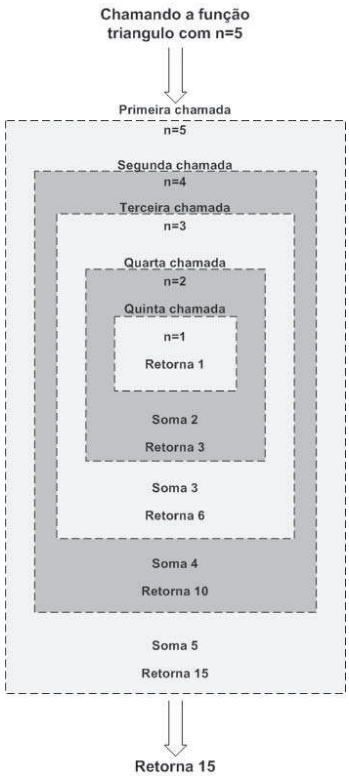

Figura 5.4: O que ocorre a cada chamada

5.3 Números de Fibonacci

Fibonacci (matemático da Renascença italiana) estabeleceu uma série curiosa de números para modelar o número de casais de coelhos em sucessivas gerações. Assumindo que nas primeiras duas gerações só existe um casal de coelhos, a seqüência de Fibonacci é a seqüência de inteiros: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

No programa 5.5 é mostrada uma versão iterativa para calcular o n-ésimo termo da seqüência de Fibonacci.

Programa 5.5: Cálculo do n-ésimo termo de Fibonacci (versão iterativa)

```
int fibc(int n)
{
    int l,h, x, i;
    if( n <= 2)
        return 1;
    l = 0;
    h = 1;
    for(i=2; i<= n; i++)
    {
        /* Cálculo do próximo número da seqüência. */
        x = l;
        l = h;
        h = x + l;
    }
    return h;
}
```

O n-ésimo número é definido como sendo a soma dos dois números anteriores. Logo, fazendo a definição recursiva:

- fib(n) = n se n <= 2
- fib(n) = fib(n-2) + fib(n-1) se n > 2

A sua determinação recursiva impõe o cálculo direto do valor para dois elementos de base (a primeira e a segunda geração). No programa 5.6 é mostrada a versão recursiva para calcular o n-ésimo termo da seqüência de Fibonacci.

Programa 5.6: Cálculo do n-ésimo termo de Fibonacci (versão recursiva)

```
int fibr( int n )
{
    if( n <= 2)
    {
    return 1;
    }
    /* chama a si próprio 2 vezes!!! */
    return fibr(n-1) + fibr(n-2);
}
```

Esta solução (programa 5.6) é muito mais simples de programar do que a versão iterativa (programa 5.5). Contudo, esta versão é ineficiente, pois cada vez que a função `fibr` é chamada, a dimensão do problema reduz-se apenas uma unidade (de n para n-1), mas são feitas duas chamadas recursivas. Isto dá origem a uma explosão combinatorial e o computador acaba por ter de calcular o mesmo termo várias vezes.

Para calcular `fibr(5)` é necessário calcular `fibr(4)` e `fibr(3)`. Conseqüentemente, para calcular `fibr(4)` é preciso calcular `fibr(3)` e `fibr(2)`. E assim sucessivamente. Este tipo de processamento é inadequado, já que o computador é obrigado a fazer trabalho desnecessário. No exemplo, usando o programa 5.6, para calcular `fibr(5)` foi preciso calcular `fibr(4)` 1 vez, `fibr(3)` 2 vezes, `fibr(2)` 3 vezes e `fibr(1)` 2 vezes. No programa iterativo (programa 5.5), apenas era necessário calcular `fibc(5)`, `fibc(4)`, `fibc(3)`, `fibc(2)` e `fibc(1)` 1 vez. A figura 5.5 demonstra como ficaria a chamada do programa 5.6 para cálculo do sétimo termo.

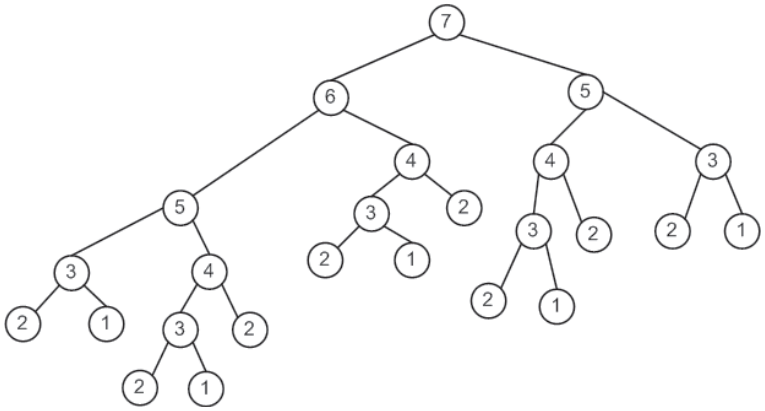

Figura 5.5: Cálculo de Fibonacci recursivo para o sétimo termo

5.4 Algoritmo de Euclides

O algoritmo de Euclides busca encontrar o máximo divisor comum (MDC) entre dois números inteiros diferentes de zero. O procedimento é simples:

1. Chame o primeiro número de *m* e o segundo número de *n*;
2. Divida *m* por *n* e chame o resto de *r*;
3. Se *r* for igual a zero, então o MDC é *n* e o procedimento termina, se não o procedimento continua;
4. Atribua *n* para *m* e *r* para *n*;
5. Recomece o procedimento do segundo passo.

Estes passos podem ser descritos conforme o algoritmo 5.1.

No programa 5.7 é vista uma versão iterativa do algoritmo de Euclides para cálculo do MDC.

Algoritmo 5.1: Algoritmo de Euclides

```
   Input: M e N
   Output: MDC calculado
1  begin
2      r ⟵ resto da divisão m por n
3      while r ≠ 0 do
4          m ⟵ n
5          n ⟵ r
6          r ⟵ resto da divisão m por n
7      endw
8      return n;
9  end
```

Programa 5.7: Cálculo do MDC iterativo

```
1  #include <stdio.h>

   int mdc(int m, int n)
   {
       int r;
6      while( m % n != 0)
       {
           r = m % n;
           m = n;
           n = r;
11     }
       return n;
   }

   int main(void)
16 {
       printf("60,36 = %d\n", mdc(60,36));
       printf("36,24 = %d\n", mdc(36,24));
       return 0;
   }
```

O programa 5.8 implementa o algoritmo de Euclides de forma recursiva.

Programa 5.8: Cálculo do MDC recursivo

```
   #include <stdio.h>
   int mdc(int m, int n)
5  {
       if( n == 0)
```

```
    {
        return m;
    }
    return mdc(n, m % n);
}

int main(void)
{
    printf("60,36 = %d\n", mdc(60,36));
    printf("36,24 = %d\n", mdc(36,24));
    return 0;
}
```

O MDC entre dois números *m* e *n* é também um divisor da sua diferença, *m-n*. Por exemplo: o MDC de 60 e 36 é 12, que divide 24 = 60-36. Por outro lado, o MDC dos dois números *m* e *n* é ainda o MDC do menor número (*n*) com a diferença (*m-n*). Se houvesse um divisor comum maior, ele seria igualmente divisor de *n*, contrariamente à hipótese. Portanto, é possível determinar o MDC de dois números através da determinação do MDC de números cada vez menores. O programa termina quando os números forem iguais e, neste caso, o MDC é este número. Exemplo: 60-36 = 24; 36-24 = 12; 24-12 = 12; 12 = 12. No programa 5.9 é vista uma versão do programa MDC utilizando os passos descritos (*m* sempre tem que ser maior que *n*).

Programa 5.9: Cálculo do MDC recursivo

```
#include <stdio.h>
int mdc(int m, int n)
{
    if( m == n )
    {
        return m;
    }
    if( m-n >= n)
    {
        return mdc(m-n, n);
    }
    return mdc(n, m-n);
}

int main(void)
{
    printf("60,36 = %d\n", mdc(60,36));
    printf("36,24 = %d\n", mdc(36,24));
```

```
  return 0;
}
```

5.5 Torres de Hanoi

No grande templo de Benares, embaixo da cúpula que marca o centro do mundo, repousa uma placa de latão onde estão presas três agulhas de diamante, cada uma com 50 cm de altura e com espessura do corpo de uma abelha. Em uma dessas agulhas, durante a criação, Deus colocou sessenta e quatro discos de ouro puro, com o disco maior repousando sobre a placa de latão e os outros diminuindo cada vez mais ate o topo. Essa é a torre de Brahma. Dia e noite, sem parar, os sacerdotes transferem os discos de uma agulha de diamante para outra de acordo com as leis fixas e imutáveis de Brahma, que exigem que o sacerdote em vigília não mova mais de um disco por vez e que ele coloque este disco em uma agulha de modo que não haja nenhum disco menor embaixo dele. Quando os sessenta e quatro discos tiverem sido assim transferidos da agulha em que a criação de Deus as colocou para uma das outras agulhas, a torre, o templo e os brâmanes virarão pó, e com um trovejar, o mundo desaparecerá.

Várias são as histórias que contam a origem da Torre de Hanoi. A história anterior foi utilizada pelo matemático francês Édouard Lucas em 1883 como inspiração para o jogo criado por ele [18].

A Torre de Hanoi é um quebra-cabeça que consiste em uma base contendo três pinos, onde em um deles são dispostos sete discos uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo. O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação. O número de discos pode variar sendo que o mais simples contém apenas três (figura 5.6).

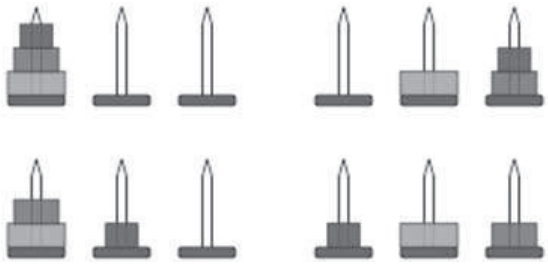

Figura 5.6: Torre de Hanoi

A solução para o problema da Torre de Hanoi com recursividade baseia-se no seguinte:

1. A única operação possível de ser executada é mover um disco de um pino para outro;
2. Uma torre com (N) discos, em um pino, pode ser reduzida ao disco de baixo e a torre de cima com (N-1) discos;
3. A solução consiste em transferir a torre com (N-1) discos do pino origem para o pino auxiliar, mover o disco de baixo do pino origem para o pino destino e transferir a torre com (N-1) discos do pino auxiliar para o pino destino. Como a transferência da torre de cima não é uma operação possível de ser executada, ela deverá ser reduzida sucessivamente até transformar-se em um movimento de disco.

O algoritmo 5.2 demostra os passos necessários para desenvolver a função recursiva. Os passos podem ser observados na figura 5.7.

Algoritmo 5.2: Passar n peças de uma torre (A) para outra (C)

```
1 begin
2 |  Passar n-1 peças da torre inicial (A) para a torre livre (B)
3 |  Mover a última peça, para a torre final (C)
4 |  Passar n-1 peças da torre B para a torre (C)
5 end
```

Baseado no algoritmo 5.2 é possível determinar o número de movimentos necessários e determinar os movimentos necessários.

O número de movimentos necessário é simples de determinar:

```
hanoi_count(n) = hanoi_count(n-1) + 1 + hanoi_count(n-1)
```

Neste caso, é possível evitar dupla recursividade (como ocorre com Fibonacci) de uma forma simples:

```
hanoi_count(n) = 2 * hanoi_count(n-1) + 1
```

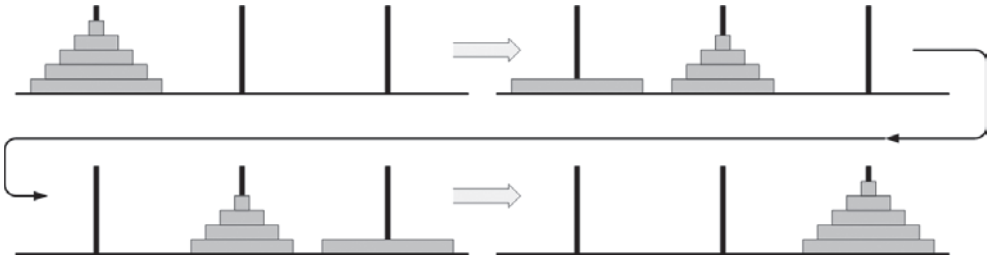

Figura 5.7: Movimentos conforme algoritmo

Para conseguir transferir todos os discos da primeira estaca à terceira é $2^n - 1$, sendo n o número de discos, portanto:

- Para solucionar um hanoi de 3 discos, são necessários 23 - 1 movimentos = 7 movimentos.
- Para solucionar um hanoi de 7 discos, são necessários 127 movimentos (27 - 1).
- Para solucionar um hanoi de 15 discos, são necessários 32767 movimentos (215 - 1).
- Para solucionar um hanoi de 64 discos, como diz a lenda, são necessários 18446744073709551615 movimentos (264 - 1) ou 585 bilhões de anos (considerando um movimento por segundo).

No programa 5.10 é vista a solução para o problema da Torre de Hanoi seguindo as definições recursivas.

Programa 5.10: Torre de Hanoi recursivo

```
/* programa_recursividade_hanoi.c */

#include <stdio.h>

void hanoi (int discos , char origem, char destino, char ajuda);

void hanoi (int discos, char origem, char destino, char ajuda)
{
    if( discos == 1)
    {
        printf("\t Mova o disco %d de %c para %c \n",discos,origem, destino);
    }
    else
```

```
    {
        hanoi(discos-1,origem,ajuda,destino);
        printf("\t Mova o disco %d de %c para %c \n",discos,origem,destino);
        hanoi(discos-1,ajuda,destino,origem);
    }
    return;
}

int main (void)
{
    int total_discos;
    printf("Informe o numero de discos:");
    scanf("%d",&total_discos);
    hanoi(total_discos,'A','B','C');
    printf("\n");
    return 0;
}
```

5.6 Curiosidades com Recursividade

O programa 5.11 utiliza de recursão para imprimir a fase da lua (cheia, minguante, crescente ou nova). Este programa foi um dos concorrentes do 15th International Obfuscated C Code Contest[1].

Programa 5.11: Natori - Imprimindo as fases da lua

```
/* programa_recursividade_curiosidade_01.c */

#include <stdio.h>
#include <math.h>
double l;

main(_,o,O)
{
    return putchar((_--+22&&_+44&&main(_,-43,_),_&&o) ?
        ( main(-43,++o,O),
        ( (l=(o+21)/sqrt(3-O*22-O*O),l*l<4 &&
            (fabs(((time(0)-607728)%2551443)/405859.-4.7 +
                acos(l/2))<1.57))["  #"]))
        :10 );
}
```

1. Disponível em http://www.iocc.org/years.html

O programa 5.12 participou do mesmo concurso[2] e usa recursividade em cima de ponteiros. O código irá gerar outro código que, quando compilado, irá gerar outro código e assim sucessivamente.

Programa 5.12: Dhyanh - Saitou, aku, soku e zan

```
/* programa_recursividade_curiosidade_01.c */

                    #define/**/X
                  char*d="X0[!4cM,!"
5               "4cK`*!4cJc(!4cHg&!4c$j"
              "8f'!&~]9e)!'|:d+!)rAc-!*m*"
             ":d/!4c(b4e0!1r2e2!/t0e4!-y-c6!"
            "+|,c6!)f$b(h*c6!(d'b(i)d5!(b*a'`&c"
            ")c5!'b+`&b'c)c4!&b-_$c'd*c3!&a.h'd+"
10        "d1!%a/g'e+e0!%b-g(d.d/!&c*h'd1d-!(d%g)"
        "d4d+!*l,d7d)!,h-d;c'!.b0c>d%!A`Dc$![7)35E"
      "!'1cA,,!2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)"
      "f%2g*!?f5f,!=f-*e/!<d6e1!9e0'f3!6f)-g5!4d*b"
      "+e6!0f%k)d7!+~^'c7!)z/d-+!'n%a0(d5!%c1a+/d4"
15    "!2)c9e2!9b;e1!8b>e/! 7cAd-!5fAe+!7fBe(!"
     "8hBd&!:iAd$![7S,Q0!1 bF 7!1b?'_6!1c,8b4"
     "!2b*a,*d3!2n4f2!${4 f.    '!%y4e5!&f%"
    "d-^-d7!4c+b)d9!4c-a 'd     :!/i('`&d"
    ";!+l'a+d<!)l*b(d=!' m-    a &d>!&d'"
20  "`0_&c?!$dAc@!$cBc@!$ b     <   ^&d$`"
   ":!$d9_&l++^$!%f3a' n1       _     $ !&"
  "f/c(o/_%!(f+c)q*c %!      *     f &d+"
  "f$s&!-n,d)n(!0i- c-      k)     ! 3d"
  "/b0h*!H`7a,![7*  i]       5     4  71"
25 "[=ohr&o*t*q*`*d  *v      *r      ; 02"
 "7*~=h./}tcrsth  &t       :       r  9b"
"].,b-725-.t--//  #r      [        <  t8-"
"752793? <.~;b   ].t--+r   /       #   53"
"7-r[/9~X .v90   <6/<.v;-52/={      k  goh"
30 "./}q; u vto   hr `.i*$engt$       $   ,b"
";$/   =t ;v;   6   ='it.';7=`      :   ,b-"
"725  = / o`.  .d     ;b]`--[/+    55/   }o"
"`.d  :  - ?5  /        }o`.'   v/i]q    - "
"-[;  5 2 =` it         .       o;53-    . "
35 "v96  <7 /    =o        :          d     =o"
"--/i ]q--    [;       h.          /     = "
"i]q--[ ;v    9h       ./          <     - "
```

2. http://www.iocc.org/2000/dhyang.hint

```
"52={cj u     c&'       i  t       . o      ; "
"?4=o:d=      o--       / i        ]q       - "
"-[;54={ cj    uc&       i]q       -        -"
"[;76=i]q[;6   =vsr      u.i       /        ={"
"=),BihY_gha   ,)\0      "         ,         o [
 3217];int i,        r,w,f        ,         b        ,x ,
 p;n(){return        r <X         X         X        X X
 768?d[X(143+    X  r++           +         *d       ) %
  768]:r>2659    ?   59:          (         x        = d
  [(r++-768)%    X 947            +        768]      ) ?
  x^(p?6:0):(p = 34               X         X        X )
  ;}s(){for(x= n     ();          (        x^        ( p
 ?6:0))==32;x= n     ()           )   ;return x       ; }
 void/**/main X     ()           {         r         = p
 =0;w=sprintf (X     X           X        X X       X o
 ,"char*d="); for                (      f=1;f <     * d
 +143;)if(33-( b=d               [        f++ X     ] )
 ){if(b<93){if    X(!            p         )       o
  [w++]=34;for X(i               =        35       +
   (p?0:1);i<b;    i++           )         o
   [w++]=s();o[    w++                     ]
    =p?s():34;}     else                   X
     {for(i=92;      i<b;                  i
      ++)o[w++]=    32;}                   }
          else o      [w++                 ]
                       =10;o               [
                         w]=0              ;
                           puts(o);}
```

5.7 Cuidados com Recursividade

Ao escrever funções recursivas, deve-se ter um comando `if` em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se não existir, a função nunca retornará quando chamada (equivalente a um loop infinito). Omitir o comando `if` é um erro comum quando se escrevem funções recursivas.

Isto garante que o programa recursivo não gere uma seqüência infinita de chamadas a si mesmo. Portanto, todo programa deve ter uma condição de parada não recursiva. Nos exemplos vistos, as condições de paradas não recursivas eram:

- Fatorial: 0! = 1
- Números Triangulares: n = 1
- Seqüência de Fibonacci: fib(1) = 1 e fib(2) = 1

- Euclides: n = 0
- Hanoi: discos = 1

Sem essa saída não recursiva, nenhuma função recursiva poderá ser computada. Ou seja, todo programa recursivo deve ter uma condição de parada não recursiva.

5.8 Vantagens

A maioria das funções recursivas não minimiza significativamente o tamanho do código ou melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas à função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro da pilha. Como o armazenamento para os parâmetros da função e variáveis locais está na pilha e cada nova chamada cria uma nova cópia dessas variáveis, a pilha pode provavelmente escrever sobre outra memória de dados ou de programa. Contudo, não é necessário se preocupar com isso, a menos que uma função recursiva seja executada de forma desenfreada. A principal vantagem das funções recursivas é a possibilidade de utilizá-las para criar versões mais claras e simples de vários algoritmos, embora uma solução não recursiva envolvendo outras estruturas (pilhas, filas etc.) seja mais difícil de desenvolver e mais propensa a erros. Dessa forma, ocorre um conflito entre a eficiência da máquina e a do programador. O custo da programação está aumentando e o custo da computação está diminuindo. Isto leva a um cenário que não vale a pena para um programador demandar muito tempo para elaborar programas iterativos quando soluções recursivas podem ser escritas mais rapidamente. Somente deve-se evitar o uso de soluções recursivas que utilizam recursão múltipla (Fibonacci, por exemplo).

5.9 Exercícios

1. Determine o que a seguinte função recursiva em C calcula. Escreva uma função iterativa para atingir o mesmo objetivo:

```
int func(int n)
{
    if( n == 0)
        return 0;
    return (n+ func(n-1));
}
```

2. Imagine vet como um vetor de inteiros. Apresente programas iterativos e recursivos para calcular:

   a) o elemento máximo do vetor;

   b) o elemento mínimo do vetor;

   c) a soma dos elementos do vetor;

   d) o produto dos elementos do vetor;

   e) a média dos elementos do vetor.

3. Uma cadeia *s* de caracteres é palíndrome se a leitura de *s* é igual da esquerda para a direita e da direita para a esquerda. Por exemplo, as palavras seres, arara e ama são palíndromes, assim como a sentença "A torre da derrota". Faça um programa que fique lendo palavras do usuário e imprima uma mensagem dizendo se as palavras são palíndromes ou não. O seu programa deve ter uma função recursiva com o seguinte protótipo: `int palindrome( char * palavra, int first, int last);`. Esta função recebe como parâmetros a palavra que está sendo testada se é palíndrome ou não e os índices que apontam para o primeiro e último caracteres da palavra. Talvez seja mais fácil fazer uma função com o seguinte protótipo: `int checaPalindrome(char * palavra, int last);`. Esta função recebe a palavra a ser verificada e o tamanho da palavra.

4. A função de Ackermann [2, 19] é definida recursivamente nos números não negativos como segue:

   a) a(m,n) = n + 1 Se m = 0,

   b) a(m,n) = a(m-1,1) Se m <> 0 e n = 0,

   c) a(m,n) = a(m-1, a(m,n-1)) Se m <> 0 e n <> 0

Faça um procedimento recursivo para computar a função de Ackermann. Observação: Esta função cresce muito rápido, assim ela deve ser impressa para valores pequenos de m e n.