Capítulo 3
Decrementar para Conquistar

"Para fazer um algoritmo recursivo é necessário ter fé"

- Siang Wun Song -

Sumário:

3.1- Conceitos

A estratégia de **Decrementar para Conquistar**, também conhecida como **Recursão Linear**, caracteriza-se pela execução de uma única chamada recursiva para reduzir o problema em subproblemas mais simples. Essa redução é feita pela decomposição da amostra de dados por duas amostras mais pequenas. Uma unitária e outra cuja dimensão é menor do que a amostra inicial.

Segundo P. Feofiloff, muitos problemas de computação possuem a seguinte propriedade: cada entrada do problema contém uma entrada menor do mesmo problema. Para resolver um problema como essa propriedade, devemos utilizar a seguinte estratégia:

```
se a entrada do problema é pequena então
  resolva diretamente;
senão
   início
      reduza a uma entrada menor do mesmo problema;
      aplique este método à entrada menor;
      volte à entrada original.
    fim.
```

A aplicação dessa estratégia produz um **algoritmo recursivo**, que implementado numa linguagem de programação, torna-se num **programa recursivo** ou num programa que contém uma ou mais **funções recursivas**.

Uma **função recursiva** é aquela que possui, em seu corpo, uma ou mais chamadas a si mesma. Uma chamada de uma função a si mesma é dita uma **chamada recursiva**.

Se a função só possui chamadas feitas por outras funções, então essa função é **iterativa**.

3.2- Recursão e Indução

3.3.1- Potência de 2^n

Pela matemática elementar, sabemos que

$2^0 = 1$
$2^n = 2 \times 2 \times 2 \ldots\ldots \times 2$ (n vezes)

Esta fórmula permite-mos desenvolver um algoritmo iterativo, que consiste em: declarar uma variável pot que irá calcular o valor da potência de dois elevado a n. Inicialmente essa variável deverá receber o valor 1. Em seguida, percorrer todos os números inteiros que vão de n até 1. Para cada número percorrido, multiplicar o conteúdo de pot por dois. Ao terminar o percurso, retornar o valor de pot.

```
int potencia ( int n )
{
  int  pot = 1;
  for ( ; n > 0 ; n --)  pot * = 2;
  return pot;
 }
```

Agora, vamos determinar uma formula recorrente para expressar a potência de 2 elevado a n como uma combinação de potências de 2 anteriores. Sabemos que:

Caso base: n = 0, temos que $2^0 = 1$.

Passo indutivo:

n = 1, temos que $2^1 = 2$
$= 2 \times 1$
$= 2 \times 2^0$

n = 2, temos que $2^2 = 4$
$= 2 \times 2$

$= 2 \times 2^1$

n = 3, temos que $2^3 = 2 \times 2 \times 2$
$= 2 \times 4$
$= 2 \times 2^2$

Podemos concluir que, que para todo n > 0, a seguinte fórmula é válida.

$$2^n = 2 \times 2^{n-1}$$

Logo, estamos em condições de desenvolver uma função recursiva para calcular a potência de 2 elevado a n para qualquer n ≥ 0.

```
int Potencia2nRec ( int n )
{
   if ( n == 0 )  return 1;
   else  return 2* Potencia2nRec (n - 1);
}
```

Contudo, essa função só pode ser invocada se n ≥ 0. Essa restrição, determina o início do processo recursivo, e é conhecida pelo nome de **início da recursão**. O início da recursão deve ser incluído numa função que irá invocar a função Potencia2n().

3.3.2- Quadrados Perfeitos

um número natural é um quadrado perfeito quando este pode ser escrito como o quadrado de um número natural. Por exemplo, são quadrados perfeitos:

$1 = 1^2$
$4 = 2^2$
$9 = 3^2$
$16 = 4^2$
$25 = 5^2$
:

Pela sequência, facilmente se pode constactar que o **caso base** acontece quando n = 1.

Vamos determinar o **passo recursivo** com a definição de uma fórmula recorrente que exprima um número perfeito como combinação de números perfeitos anteriores. Sabemos que:

$Q(1) = 1$
$= 1^2$

$Q(2) = Q(1) + 3$
$= 1 + 3$
$= 4$
$= 2^2$

$$Q(3) = Q(2) + 5$$
$$= 1 + 3 + 5$$
$$= 9$$
$$= 3^2$$

Podemos concluir que, para todo n > 1, a seguinte fórmula é válida.

$$Q(n) = Q(n-1) + (2n-1)$$

Logo, estamos em condições de desenvolver uma função recursiva que calcula o número perfeito de n.

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

3.3.2- Factorial de um Número Natural

Pela matemática elementar, sabemos que:

n! = 1x2x3x . . . x (n-1) x n

ou

n! = n x (n-1) x (n-2) x …x1

Esta fórmula permite-nos desenvolver um algoritmo iterativo, que caracteriza-se por possuir um ciclo que percorre todos os números inteiros positivos que vão de n até 1. Para cada número percorrido, multiplicar o seu conteúdo por uma variável que inicialmente foi inicializada com uma unidade. Quando terminarmos do percurso, essa variável possuirá o valor do factorial de n.

```
int factorial (int n)
{
   int  i , f = 1;
   for ( ;  n  >  0 ; n --)  f *= i;
   return f;
}
```

Agora vamos tentar escrever uma fórmula recorrente que expresse o factorial de n em função de termos anteriores. Sabemos que:

Caso base: 0! = 1 e que 1! = 1. (Temos dois casos base.)

Passo recursivo:

n = 2, temos 2! = 2x1
= 2x1!

= 2

n = 3, temos 3! = 3x2x1
= 3x2
= 3x2!
= 6

n = 4, temos 4! = 4x3x2x 1
= 4x6
= 4x3!
=24

Podemos concluir que, para todo n > 1, a fórmula a seguir é válida.

n! = n x (n-1)!

Logo, estamos em condições de implementar uma função recursiva que calcula o factorial de um número natural.

```
int factorialRec (int n)
{
  if ( n == 0 )  return 1;
  else if ( n == 1 ) return 1;
       else  return n* factorialRec (n - 1);
 }
```

É importante salientar que esta função que receber como parâmetro um valor maior ou igual a zero. Logo, não temos a necessidade de verificar se o parâmetro é igual a zeros ou igual a um, basta verificar, se ele é menor ou igual a um. Essa optimização de código pode ser descrita pela seguinte função.

```
int factorialRec (int n)
 {
   if ( n <= 1 )  return 1;
   else  return n* factorialRec (n - 1);
  }
```

3.3.4- Soma dos Dígitos de um Número Inteiro Positivo

A estratégia para implementar um algoritmo iterativo para essa função é um pouco mais complexa. Ela consiste em separar os dígitos do número inteiro enviado como parâmetro até que este seja menor do que 10. Durante esse processo de separação, devemos adicionar cada dígito á uma variável auxiliar. Quando terminarmos de realizar essa operação teremos nessa variável a soma de todos os dígitos desse número.

```
int somaDigitos ( int n )
{
  int  s = 0;
```

```
    do
     {
        s += n % 10;
        n =  n  / 10;
     }
    while ( n < 10)
    return s;
}
```

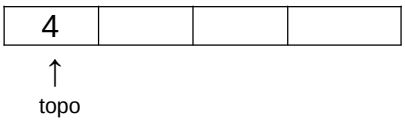
Observe que este problema é iminentemente recursivo. Se o número for menor do que 10, então ele possui um único dígito e a sua soma é o próprio número. Mas se o número possuir mais do que um algarismo (dígito), então ele é maior ou igual a 10, e o processo de separação garante que no próximo passo esse número possui menos um algarismo (dígito). Como em cada passo ele obtêm um número com menos dígitos, então esse processo é finito. Isso que dizer que chegaremos a um instante que teremos um único dígito e nessa altura a solução é trivial.

Logo, estamos em condições de escrever uma função recursiva para calcular a soma dos dígitos de um número inteiro positivo.

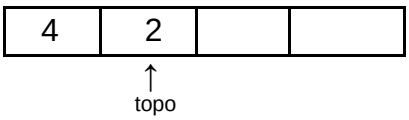
```
int somaDigitosRec (int n)
{
   if (n < 10)  return n;
   else return   (n %10 ) + somaDigitosRec  (n / 10) ;
 }
```

Para compreender o funcionamento desta função, veremos a seguir uma simulação descritiva. Não vamos considerar nessa simulação, os valores do parâmetro que serão armazenados automaticamente na pilha.

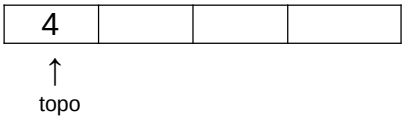
Suponhamos sem perda da generalidade que n é igual à 324. Como n não é menor do que 10, caso base, a execução do programa é direccionada para a cláusula **else** e o cálculo do resto da divisão de 324 por 10 é automaticamente inserido numa pilha em memória.



Em seguida, a função volta a ser chamada com o parâmetro igual à 32. Mais uma vez o fluxo de execução é direcionado para a clausula **else** e o cálculo do resto da divisão de 32 por 10 é automaticamente inserido na pilha. Nesse instante a pilha possui dois elementos.



A função volta a ser chamada para com o parâmetro igual à 3. Mas como esse valor é menor do que 10, caso base, a função retorna o valor 3. Como a chamada foi concluída, iniciamos o encerramento da última chamada suspensa. Esse encerramento consiste em remover o elemento que está no topo da pilha, e adicionar em seguida, esse valor ao número que foi retornado anteriormente, ou seja, a efectuar a operação 3 + 2. Esse resultado é apurado e a pilha passa a ter um único elemento.



Como a pilha não está vazia, temos algumas chamadas que foram suspensas. Iniciamos desse modo, o encerramento da penúltima chamada suspensa. Esse encerramento consiste em remover o elemento que está no topo da pilha, e adicionar a esse valor o resultado do calculo feito anteriormente, ou seja, efectuar a operação 4 + (3 + 2) = 4 + 5 = 9.

Agora a pilha está vazia, isso quer dizer que não temos mais chamadas não concluídas. Então, a função termina a execução e retorna o valor 9.

Um erro muito comum na implementação de algoritmos recursivos está na utilização de comandos de repetição, em vez de comandos de desvio condicional. Como princípio de boas prácticas de programação, os comandos de repetição, não devem ser utilizados nos algoritmos recursivos.

3.3 - Simulação da Recursão

Vimos na secção anterior, uma descrição narrativa da simulação de uma função recursiva. Também vimos que nessa simulação utilizamos um dispositivo de pilha, denominado de pilha de execução recursiva.

Sempre que uma função recursiva é chamada, são armazenados na pilha de execução recursiva, os parâmetros da função, as variáveis locais, o valor de retorno e o endereço da função. Enquanto a função não for encerrada, essa informação permanece em memória.

A gestão da pilha de execução recursiva é feita pelas operações de inserção (empilhar) e remoção (desempilhar).

Sempre que uma função recursiva é chamada, são empilhados os seguintes elementos:

1- Os valores dos parâmetros da chamada são armazenados na pilha;
2- As variáveis locais são armazenadas na pilha;
3- O endereço do valor de retorno é armazenado na pilha;
4- O indicador de topo da pilha é incrementado;

Em seguida, é feito um desvio do fluxo do programa para a primeira instrução da função.

Sempre que uma chamada recursiva termina, são desempilhados os seguintes elementos:

1- Os valores dos parâmetros da chamada são removidos da pilha;
2- Os valores das variáveis locais são removidos da pilha;
3- O endereço do valor de retorno é removido da pilha;
4- O indicador de topo da pilha é decrementado;

Em seguida é feito um desvio para o local onde foi feira a chamada da função.

Agora veremos um diagrama, que mostra visualmente o funcionamento desse dispositivo. Tomemos sem perda da generalidade o exemplo da função factorial, vista na secção anterior, e suponhamos que n = 4.

```
fatorial (4)
  4 x fatorial (3)
     4 x ( 3 x fatorial (2) )
        4 x ( 3 x (2  x  fatorial (1) ) )
                          1
        4 x ( 3 x ( 2 x 1 ) )
     4 x ( 3 x  2 )
  4 *  6
24
```

Este diagrama mostra que durante o processo de cálculo temos duas fases distintas:

1º fase (Expansão): a execução das operações de multiplicação foram suspensas até que o valor do término da decomposição fosse igual a 1, ou seja os valores das chamadas da função foram armazenados na pilha.

2ª fase (Contracção): as operações de multiplicação que foram suspensas são executadas, ou seja, os valores da chamada da função são desempilhados e combinados com o operador multiplicação.

3.4- Iteração e Recursão

Conforme vimos na secção anterior, que a recursão é uma técnica de programação que normalmente requere mais espaço em memória e mais tempo de processamento do que as soluções iterativas equivalentes. Também vimos, que normalmente a chamada recursiva deve ser a última instrução a ser executada. Essa técnica, dá-se o nome de recursão caudal, e facilita a transformação de algoritmos recursivos em algoritmos iterativos e vice-versa. Dado um esquema geral de uma função recursiva:

```
void  p ( int n )
 {
   int i ;
   for ( i = n ; i  >  x ; i = i - 1)
         /*  resolva o problema   */
 }
```

onde x é uma constante. Podemos obter uma função recursiva equivalente se utilizarmos o seguinte esquema:

```
void  pRec (  int n )
{
   if ( n = x )
      /*   resolva o problema    */
   else
      p ( n – 1)
}
```

É importante salientar que recursão caudal ocorre quando a última instrução executada por uma função é uma chamada recursiva e essa chamada encerra essa função. Contudo, é importante observar que a última instrução a ser executada não necessita de estar na última linha de instrução da função.

Transformar uma função que não apresenta recursão caudal para uma função iterativa é um processo mais complicado. Essa transformação, requere a utilização explícita de uma pilha, e a função iterativa resultante, normalmente é mais difícil de ler e de entender.

3.4.1- Multiplicação de x por y

A multiplicação de dois números inteiros positivos, x e y, pode ser implementada por uma função iterativa que utiliza apenas operações de adição.

```
int   mult ( int x , int y)
 {
    int result = 0, i;
    for ( ; y > 0 ; y -- )  result += x ;
    return result;
 }
```

Pelo esquema anterior, temos:

```
int   multRec ( int x, int y)
 {
    If ( y == 0 ) return  0;
    else return x + multRec ( x , y -1);
 }
```

Vejamos em seguida, uma outra função iterativa que resolve o mesmo problema.

```
int   mult ( int x , int y)
 {
    int result = 0, i;
    for ( i = 0 ; i  <= y > 0 ; i -- ) result += x
```

```
    return result;
}
```

Como o processo iterativo é feito com uma variável local, essa variável deve ser declarada como parâmetro.

```
int   multRec ( int x, int y , int  i )
 {
    If ( i == 0 ) return  0;
    else return x + multRec ( x , y , i + 1);
 }
```

No corpo das funções recursivos, normalmente, observe que digo, normalmente, não aparecem comandos de repetição.

3.5- Vantagens de Desvantagens

Vamos discutir algumas vantagens e desvantagens de cada tipo de estratégia. A utilização de algoritmos recursivos tem a vantagem de, em geral, serem muito simples de ler, e o seu código ser claro e conciso. Assim, alguns problemas que podem parecer complexos no início, acabam por ter uma solução simples e muito elegante, ao passo que, nos algoritmos iterativos essas soluções são em geral longas e requerem que o programador tenha alguma experiência para serem entendidos. Por outro lado, as soluções recursivas podem ocupar muita memória, porque o computador necessita de armazenar o conteúdo dos parâmetros e das variáveis locais, em todas as chamadas recursivas numa pilha de execução do programa.

3.6 - Recomendações Bibliográficas

Para o leitor aprofundar os temas abordados neste capítulo, recomendamos os seguintes livros

Jeri R. Hanly , Elliot B. Koffman; - *Problem Solving ang Program Design in C*, 4th Edition, Addison-Wesley, 2004.

Robert E. S.;- *Programming Abstraction in C: a Second Course in Computer Science*, Addison-Wesley, 1998

Sedgewick R. ;- Algoritms in C, Addison-Wesley, 1990

3.7 - Exercícios Propostos

3.7.1-Desenvolva uma função recursiva que recebe um número inteiro positivo n e Calcula o valor da série Harmônica:

Hn = 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + …+ 1/n

3.7.2-Dada a seguinte soma:

0 + 1 + 2 + 3 + 4 + ... + n

Desenvolva uma função recursiva e uma função iterativa para calcular os n primeiros termos.

3.7.3-Desenvolva uma função recursiva e uma função iteratica, que recebe como parâmetro dois números inteiros positivos e retorna o valor da multiplicação desses números, com base na seguinte fórmula recorrente.

$$a \times b = \begin{cases} a \; x \; b & se \; b = 1 \\ a \; x \; (b-1) + a & se \; b > 1 \end{cases}$$

3.7.4-Desenvolva uma função recursiva que recebe um número na numeração decimal e o converte para a numeração binária.

3.7.5-Desenvolva uma função recursiva que recebe como parâmetro um número inteiro positivo e mostre a representação desse número na notação binária.

3.7.6-Desenvolva uma função recursiva que recebe como parâmetro um número real x e um número natural n, e calcula a potência de x elevado a n, com base na seguinte fórmula recorrente:

3.7.7-Desenvolva uma função recursiva que recebe como parâmetro um número inteiro positivo retorna o valor dos n primeiros termos a seguinte soma:

$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n$

3.7.8-Desenvolva uma função recursiva que recebe como parâmetro dois números inteiros positivos e calcula o valor da fórmula recorrente de Ackermann:

$$A(m,n) = \begin{cases} n+1 & se \; m = 0 \\ A(m-1,1) & se \; n = 0 \\ A\big(m-1, \; A(m,n-1)\big) & se \; n > 0 \; e \; m > 0 \end{cases}$$

Mostre através de um diagrama que A(2,2) = 7.

3.7.9-Desenvolva uma função recursiva para calcular o MDC de dois números inteiros passados como parâmetros, através da seguinte relação de recorrência.

$$MDC\ (x, y) = \begin{cases} 0 & se \; x = y \\ x & se \; x < y \\ MDC\ (x-y, x) & se \; x > y \end{cases}$$

MDC (10,6) = MDC (4,6) = MDC (6,4) = MDC (2,4) = MDC(4,2) = MDC (2,2) = 2

3.7.10-Desenvolva uma função recursiva que recebe como parâmetro um número inteiro positivo e devolve o número de dígitos desse número.

3.7.11-Desenvolva uma função recursiva para verificar se um número inteiro positivo é do tipo capicua, ou seja, se ele pode ser lido da mesma forma da esquerda para direita, quanto da direita para esquerda. Por 121, 34543.

3.7.12- Dado um número inteiro n > 0, gerar todas as possíveis combinações com as n primeiras letras do alfabeto. Por exemplo: ABC; ACB; BAC; BCA; CAB; CBA.

3.7.13- Desenvolva uma função recursiva que recebe um número inteiro positivo e retorna esse número na ordem inversa. Por exemplo, para n = 123, a sua função deve mostrar o número 321.

3.7.14-Faça a simulação da seguinte função recursiva. Suponha que n = 8

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

3.7.15- Desenvolva uma função recursiva que recebe um número inteiro positivo e mostra uma pirâmide. Por exemplo, para n = 5 a sua função deve mostrar na tela a seguinte pirâmide:

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

3.7.16- Desenvolva uma função recursiva para somar os n primeiros números inteiros.

3.7.17- A função que descrevemos em seguida, calcula o MDC (m, n) pelo algoritmo de Euclides, para m e n estritamente positivos. Desenvolva 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;
```

}

3.7.18-Desenvolva uma função recursiva que calcula quantas vezes um algarismo k está contudo num número inteiro positivo.

3.7.19- Desenvolva uma função recursiva que recebe um número inteiro positivo e verifica se esse número é primo. A função deve retornar 1 se o número for primo e zero no caso contrário.

3.7.20- Dada a série harmônica.

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$$

Desenvolva uma função recursiva para calcular o valor de H(n).

3.7.21-A Torre de Hanoi é um “quebra-cabeça" que consiste em uma base contendo três pinos, em um dos quais são dispostos alguns discos uns sobre os outros, em ordem crescente de diâmetro, de tal forma que o maior esteja na parte de baixo do pino. O problema consiste em passar todos os discos (um a um) de um pino para outro qualquer, usando o terceiro pino como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor.
Considere que os pinos são numerados, da esquerda para direita, como A, B, C e que os discos são numerados de 1 a n, sendo 1 o disco de menor diâmetro. Escreva um programa recursivo que leia a quantidade de discos n e transfira todos os discos de A para C, utilizando B como auxiliar, e descrevendo todos os movimentos.