

Algoritmos e Funções Recursivas

Considere a definição da função fatorial:

$$n! = 1 \text{ se } n \leq 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 \text{ se } n > 0$$

Considere agora a seguinte definição equivalente:

$$n! = 1 \text{ se } n \leq 0 \\ n \cdot (n-1)! \text{ se } n > 0$$

Dizemos que essa última definição é uma definição recursiva, pois usa a função fatorial para definir a função fatorial. Note que chamando a função de f temos:

$$f(n) = 1 \text{ se } n \leq 0 \\ n \cdot f(n-1) \text{ se } n > 0$$

A princípio parece estranho usar uma função para definir a si própria, mas vejamos como se calcula o fatorial usando a definição recursiva.

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$$

A definição acima faz sentido, pois tem uma regra de parada, isto é, tem um caso (n menor ou igual a zero) onde a função não é usada para calcular a si própria e que sempre vai ocorrer.

Muitas outras funções admitem definições recorrentes deste tipo, ou seja, usa-se a função para definir a si própria, com um caso particular não recorrente.

Funções recursivas em C

Já vimos que uma função em C pode chamar outras funções. Em particular pode chamar a si mesma. Quando isso ocorre dizemos que a função é recursiva.

A função fatorial pela definição recursiva acima ficaria:

```
int fatrec(int n) {
    if (n <= 0) return 1;
    return n * fatrec(n-1);
}
```

Vamos agora ilustrar as chamadas recorrentes da função `fatrec` com o programa abaixo.

Para isso, fizemos uma pequena modificação na `fatrec` e colocamos dois comandos `printf`. O primeiro mostra qual o parâmetro com o qual a função foi chamada e o segundo mostra o mesmo parâmetro após a multiplicação.

Observe na saída do programa abaixo a ordem das chamadas e a ordem das multiplicações. As multiplicações ficam pendentes até que a última chamada da função `fatrec` (**`fatrec(0)`**) seja feita.

Dado n inteiro, calcular o valor de $n!$, usando a função recursiva `fatrec`.

```
#include <stdio.h>
#include <stdlib.h>

int fatrec(int n) {
    int f;
    printf("\nchamando fatorial de %10d", n);
    if (n <= 0) return 1;
    f = n * fatrec(n-1);
    printf("\nfeita a multiplicacao por %10d", n);
    return f;
}

/* dado n>=0 calcular o fatorial de n usando fatrec */
int main() {
    int n; /* numero dado */
    /* ler o n */
    printf("digite o valor de n:");
    scanf("%d", &n);
    /* calcule e imprima o resultado de fatorial de n */
    printf("\n\nfatorial de %10d = %10d\n", n, fatrec(n));
    system("PAUSE");
    return 0;
}
```

Veja o que será impresso:

```
digite o valor de n:5

chamando fatorial de      5
chamando fatorial de      4
chamando fatorial de      3
chamando fatorial de      2
chamando fatorial de      1
chamando fatorial de      0
feita a multiplicacao por      1
feita a multiplicacao por      2
feita a multiplicacao por      3
feita a multiplicacao por      4
feita a multiplicacao por      5

fatorial de      5 =      120
```

Outro exemplo:

```
digite o valor de n:10

chamando fatorial de      10
chamando fatorial de      9
chamando fatorial de      8
chamando fatorial de      7
chamando fatorial de      6
chamando fatorial de      5
chamando fatorial de      4
chamando fatorial de      3
chamando fatorial de      2
```

```
chamando fatorial de      1
chamando fatorial de      0
feita a multiplicacao por  1
feita a multiplicacao por  2
feita a multiplicacao por  3
feita a multiplicacao por  4
feita a multiplicacao por  5
feita a multiplicacao por  6
feita a multiplicacao por  7
feita a multiplicacao por  8
feita a multiplicacao por  9
feita a multiplicacao por 10

fatorial de      10 =      3628800
```

Números Harmônicos

Considere agora a função que calcula o n-ésimo número harmônico:

$$H(n) = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/n \quad (n \geq 1)$$

Outra maneira de pensar a definição da soma acima é a seguinte:

$$H(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1/n + H(n-1) & \text{se } n > 1 \end{cases}$$

É uma definição recursiva. Usando-a para calcular $H(4)$:

$$H(4) = 1/4 + H(3) = 1/4 + 1/3 + H(2) = 1/4 + 1/3 + 1/2 + H(1) = 1/4 + 1/3 + 1/2 + 1$$

Análogo ao fatorial, a função acima também tem o caso de parada (n igual a 1), onde o valor da função não é recorrente.

Portanto, usando a definição acima:

```
double harmrec(int n) {
    if (n == 1) return 1;
    return 1.0 / (double)n + harmrec(n-1);
}
```

Para ilustrar o funcionamento de `harmrec`, veja o programa abaixo, no qual adicionamos um `printf` dentro de `harmrec`, para esclarecer qual a chamada em andamento:

Dado $n > 0$ inteiro, calcular o número harmônico de ordem n , usando a função recursiva `harmrec`.

```
#include <stdio.h>

double harmrec(int n) {
    printf("\nchamando harmrec de %10d", n);
    if (n == 1) return 1;
    return 1.0 / (double)n + harmrec(n-1);
}
```

```
/* dado n > 0 calcular o número harmônico de ordem n */  
int main() {  
    int n;    /* numero dado */  
    /* ler o n  
    printf("digite o valor de n:");  
    scanf("%d", &n);  
    /* calcule e imprima o n-ésimo número harmônico */  
    printf("\n\nharmonico de %10d = %lf", n, harmrec(n));  
}
```

Veja o que será impresso:

digite o valor de n:10

```
chamando harmrec de      10  
chamando harmrec de      9  
chamando harmrec de      8  
chamando harmrec de      7  
chamando harmrec de      6  
chamando harmrec de      5  
chamando harmrec de      4  
chamando harmrec de      3  
chamando harmrec de      2  
chamando harmrec de      1
```

harmonico de 10 = 2.928968

Recursão ou recorrência versus iteração ou repetição

Nos dois casos acima, não parece existir nenhuma vantagem, da solução recursiva para a solução iterativa. De fato, a solução recursiva é até pior em termos de gasto de memória e de tempo, pois a cada chamada é necessário guardar o contexto da chamada anterior, até ocorrer o caso de parada, ou o caso não recorrente. Isto é, as chamadas todas ficam pendentes, esperando a chamada sem recorrência, só ocorrendo o retorno de cada uma delas seqüencialmente após esse evento.

Um ponto positivo para uma solução recursiva é que a estrutura da função em C, fica análoga à estrutura da definição da função.

Outro ponto que merece ser ressaltado é que quando vamos construir um algoritmo para resolver um problema podemos pensar de maneira repetitiva (iterativamente” ou de maneira recorrente (recursiva).

A seqüência de Fibonacci

Vejam os outros exemplos:

A seqüência de Fibonacci, assim conhecida porque foi proposta pelo matemático italiano do século XI, Leonardo Fibonacci, é tal que cada elemento (com exceção dos dois primeiros que são 0 e 1), é igual a soma dos dois anteriores.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . .

A seqüência possui algumas propriedades matemáticas que não são objeto de análise neste curso.

Dado $n > 1$, imprimir os $n+1$ primeiros números de Fibonacci.

```
#include <stdio.h>

/* dado n > 1, imprime os n+1 primeiros elementos da sequência
   de fibonacci (f0, f1, f2, ... fn) */
int main() {
    int n,      /* numero dado */
        i, atual = 1, anterior = 0, auxiliar;
    /* ler o n */
    printf("digite o valor de n:");
    scanf("%d", &n);
    /* imprima os 2 primeiros f0 e f1 */
    printf("\nfibonacci de %10d = %10d", 0, 0);
    printf("\nfibonacci de %10d = %10d", 1, 1);
    /* calcule e imprima f2, f3, . . ., fn */
    for (i = 2; i <= n ; i++) {
        auxiliar = atual;
        atual = atual + anterior;
        anterior = auxiliar;
        printf("\nfibonacci de %10d = %10d", i, atual);
    }
}
```

Veja o que será impresso:

```
digite o valor de n:20
fibonacci de      0 =          0
fibonacci de      1 =          1
fibonacci de      2 =          1
fibonacci de      3 =          2
fibonacci de      4 =          3
fibonacci de      5 =          5
fibonacci de      6 =          8
fibonacci de      7 =         13
fibonacci de      8 =         21
fibonacci de      9 =         34
fibonacci de     10 =         55
fibonacci de     11 =         89
fibonacci de     12 =        144
fibonacci de     13 =        233
fibonacci de     14 =        377
fibonacci de     15 =        610
fibonacci de     16 =        987
fibonacci de     17 =       1597
fibonacci de     18 =       2584
fibonacci de     19 =       4181
fibonacci de     20 =       6765
```

Uma função não recursiva para determinar o n -ésimo número de Fibonacci:

```
int fibonacci(int n) {
    int i, atual = 1, anterior = 0, auxiliar;
    if (n == 0) return 0;
    if (n == 1) return 1;
```

```
    for (i = 2; i <= n ; i++) {  
        auxiliar = atual;  
        atual = atual + anterior;  
        anterior = auxiliar;  
    }  
    return atual;  
}
```

Por outro lado, a definição da função para o n-ésimo número de Fibonacci é claramente recursiva:

$f(0) = 0$
 $f(1) = 1$
 $f(n) = f(n-1) + f(n-2)$ se $n > 1$

Vejam agora uma versão recursiva da mesma função:

```
int fibonaccirec(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fibonaccirec(n - 1) + fibonaccirec(n - 2);  
}
```

Para ilustrar o funcionamento de fibonaccirec, veja o program abaixo, onde inserimos na função o printf, para esclarecer qual a chamada corrente:

Dado n, calcular o número de Fibonacci de ordem n, usando fibonaccirec e fibonacci.

```
#include <stdio.h>  
  
int fibonaccirec(int n) {  
    printf("\nchamando fibonacci recursiva de %10d", n);  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fibonaccirec(n - 1) + fibonaccirec(n - 2);  
}  
  
int fibonacci(int n) {  
    int i, atual = 1, anterior = 0, auxiliar;  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    for (i = 2; i <= n ; i++) {  
        auxiliar = atual;  
        atual = atual + anterior;  
        anterior = auxiliar;  
    }  
    return atual;  
}  
  
/* dado n > 0 calcular o n-ésimo número de Fibonacci */  
int main() {  
    int n;    /* numero dado */  
    /* ler o n */  
    printf("digite o valor de n:");  
    scanf("%d", &n);  
    /* calcule e imprima o n-ésimo número de Fibonacci */
```

```
printf("\n\nfibonacci de %10d recursivo = %10d  nao recursivo = %10d",  
      n, fibonaccirec(n), fibonacci(n));  
}
```

Veja o que será impresso:

digite o valor de n:5

```
chamando fibonacci recursiva de      5  
chamando fibonacci recursiva de      4  
chamando fibonacci recursiva de      3  
chamando fibonacci recursiva de      2  
chamando fibonacci recursiva de      1  
chamando fibonacci recursiva de      0  
chamando fibonacci recursiva de      1  
chamando fibonacci recursiva de      2  
chamando fibonacci recursiva de      1  
chamando fibonacci recursiva de      0  
chamando fibonacci recursiva de      3  
chamando fibonacci recursiva de      2  
chamando fibonacci recursiva de      1  
chamando fibonacci recursiva de      0  
chamando fibonacci recursiva de      1
```

```
fibonacci de          5 recursivo =          5  nao recursivo =          5
```

A versão recursiva é menos eficiente que a versão iterativa, embora mais elegante. Note que cada chamada gera duas novas chamadas. No exemplo acima, para calcular `fibonaccirec(5)`, foram geradas 14 chamadas adicionais.

Exercício – Quantas chamadas são necessárias, em função de n , para se calcular `fibonaccirec(n)`.

Máximo Divisor Comum – Algoritmo de Euclides

Um outro exemplo que converge rapidamente para o caso de parada, mas também não é melhor que a versão iterativa, já vista anteriormente, é o cálculo do mdc entre dois números, usando o algoritmo de Euclides.

Quociente		1	1	2
dividendo/divisor	30	18	12	6
Resto	12	6	0	

Veja a versão iterativa:

```
int mdc(int a, int b) {  
    int r;  
    r = a % b;  
    while (r != 0)  
        {a = b; b = r; r = a % b; }  
    return b;  
}
```

Estamos usando a seguinte definição (recursiva) para o mdc:

$$\text{mdc}(a, b) = b \text{ se } b \text{ divide } a, \text{ ou seja } a \% b = 0$$
$$\text{mdc}(b, a \% b) \text{ caso contrário}$$

Portanto a versão recursiva ficaria:

```
int mdc_recursiva(int a, int b) {
    if (a % b == 0) return b;
    return mdc_recursiva (b, a % b);
}
```

Veja agora no programa abaixo, o funcionamento de mdc_recursiva. Também acrescentamos um printf na função.

Dados a e b inteiros > 0, calcular mdc entre a e b, usando mdc_recursiva.

```
#include <stdio.h>

int mdc_recursiva(int a, int b) {
    printf("\nchamando mdc entre %5d e %5d", a, b);
    if (a % b == 0) return b;
    return mdc_recursiva (b, a % b);
}

/* dados a e b > 0 calcular o mdc entre a e b usando mdc_recursiva */
int main() {
    int a, b;    /* dados */
    /* ler a e b */
    printf("digite os valores de a e b:");
    scanf("%d%d", &a, &b);
    /* calcule e imprima o resultado de fatorial de n */
    printf("\n\nmdc entre %5d e %5d = %5d", a, b, mdc_recursiva(a, b));
}
```

Veja o que será impresso:

digite os valores de a e b:14 568

```
chamando mdc entre    14 e    568
chamando mdc entre   568 e    14
chamando mdc entre    14 e     8
chamando mdc entre     8 e     6
chamando mdc entre     6 e     2
```

```
mdc entre    14 e    568 =     2
```

Busca Sequencial

Outra função que já conhecemos é a busca sequencial numa tabela, cuja versão iterativa é:

```
/* procura x no vetor a, devolvendo o índice do elemento igual ou -1 */
int busca (int a[], int n, int x) {
    int i;
    for (i = 0; i < n; i++)
```



```
    if (a[i] == x) return i;  
    return -1; /* foi até o fim do for e não encontrou */  
}
```

Vejam agora a seguinte definição:

Procurar um elemento numa tabela de n elementos, é o mesmo que comparar com o primeiro e se não for igual, voltar a procurar o mesmo elemento na tabela restante de $n-1$ elementos. Ou seja:

busca (a, k, n, x) = -1 se $k == n$ (não encontrou)
 k se $a[k] == x$
 busca (a, k+1, n, x) caso contrário

A chamada inicial teria que ser: busca (a, 0, n, x).

```
/* procura x no vetor a, devolvendo o índice do elemento igual ou -1 */  
int busca (int a[], int k, int n, int x) {  
    if (k == n) return -1;  
    if (a[k] == x) return k;  
    return busca (a, k + 1, n, x);  
}
```

Veja agora um programa que usa a busca e o que imprime:

Programa que testa a função busca recursiva.

```
#include <stdio.h>  
  
int busca (int a[], int k, int n, int x) {  
    if (k == n) return -1;  
    if (a[k] == x) return k;  
    return busca (a, k + 1, n, x);  
}  
  
int main() {  
    int tabela[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};  
    int n = 10, zz, r;  
    zz = 23;  
    /* procura o valor zz na tabela */  
    r = busca(tabela, 0, n, zz);  
    printf("\nprocura %d resultado = %d", zz, r);  
  
    zz = 26;  
    /* procura o valor zz na tabela */  
    r = busca(tabela, 0, n, zz);  
    printf("\nprocura %d resultado = %d", zz, r);  
}
```

```
procura 23 resultado = 8  
procura 26 resultado = -1
```

Na solução acima, adicionamos um parâmetro k , cujo objetivo na verdade é funcionar como um contador. Outra forma é usar o número de elementos como um contador e nesse caso seriam necessários apenas 3 parâmetros:

```
busca (a, k, x) =  -1 se k == 0 (não encontrou)
                  k-1 se a[k-1] == x
                  busca (a, k-1, x) caso contrário
```

A chamada inicial teria que ser `busca (a, n, x)`. Veja que na primeira chamada o valor do parâmetro `k` é `n`. Enquanto na versão anterior os elementos são comparados na ordem `a[0]`, `a[1]`, ..., `a[n-1]`, nessa as comparações são feitas na ordem `a[n-1]`, `a[n-2]`, ..., `a[0]`.

Exercício – Reescreva a função `busca` com essa nova definição.

Se não for necessário devolver o índice do elemento encontrado e apenas 0(sim) ou 1 não, há outra forma de resolver fazendo a busca na ordem `a[0]`, `a[1]`, ..., `a[n-1]`. As soluções abaixo fazem uso do fato que ponteiro é vetor e vice-versa.

```
int BuscaRecF(int * a, int x, int n) {
    if (n == 0) return -1;
    if (a[0] == x) return 0;
    return BuscaRecF(&a[1], x, n - 1);
}
```

Ou ainda:

```
int BuscaRecFF(int * a, int x, int n) {
    if (n == 0) return -1;
    if (*a == x) return 0;
    return BuscaRecFF(a + 1, x, n - 1);
}
```

Ou ainda:

```
int BuscaRecFF(int a[], int x, int n) {
    if (n == 0) return -1;
    if (a[0] == x) return 0;
    a = &a[1]; /* ou a = a + 1; */
    return BuscaRecFF(a, x, n - 1);
}
```

Outra forma de pensar a função `busca` de forma recursiva é usar como parâmetro o índice inicial e o final da tabela:

```
busca (a, x, i, f) = -1 se i > f (não encontrou)
                   i se a[i] igual a x
                   busca (a, x, i + 1, f) caso contrário
```

A chamada inicial seria: `busca(a, x, 0, n - 1)`

Fica como exercício construir a função desta forma.

Busca Binária

Quando a tabela está ordenada, fazemos busca binária.

Comparamos com o elemento médio da tabela. Se for igual, a busca termina. Se for maior, fazemos a busca na metade superior, senão na metade inferior.

```
/* procura x no vetor a, devolvendo o índice do elemento igual ou -1 */
int busca_binaria (int a[], int inicio, int final, int x) {
    int k;
    if (inicio > final) return -1;
    k = (inicio + final) / 2;
    if (a[k] == x) return k;
    if (a[k] > x) return busca_binaria(a, inicio, k - 1, x);
    if (a[k] < x) return busca_binaria(a, k + 1, final, x);
}
```

A chamada inicial tem como parâmetros os índices inicial e final da tabela, zero e n-1:

busca_binaria(a, 0, n-1, x)

Maior elemento da tabela

Outro exemplo no mesmo estilo é uma função que calcular o máximo entre os elementos de um vetor, cuja versão iterativa está abaixo:

```
int maximo (int a[], int n) {
    int i, max;
    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
```

Uma forma recursiva de entender a função maximo:

- O máximo de uma tabela de n elementos, é o maior entre o último e o máximo entre os $n-1$ elementos anteriores.

Usando a forma (ii):

$\text{maximo}(a, n) = a[0]$ se $n = 1$
maior entre $a[n-1]$ e $\text{maximo}(a, n-1)$ se $n > 1$

```
int maior (int x, int y) {
    if (x >= y) return x; else return y;
}
```

```
int maximo_recursiva (int a[], int n) {
    if (n == 1) return a[0];
    return maior (a[n-1], maximo_recursiva(a, n-1));
}
```

Agora, um programa e o que é impresso usando a função. Acrescentamos também um printf, tanto na `maximo_rekursiva` como na `maior`.

Verifique pela saída a seqüência de chamadas. As chamadas da função **maior** ficam pendentes, até que a última chamada da **maximo_rekursiva** seja resolvida, isto é, até chegar o caso em que a função não é recorrente.

Programa que dado `n` inteiro > 0 , gera uma seqüência de `n` elementos aleatórios e calcula o máximo dos elementos, usando a função `maximo_rekursiva` acima.

```
#define nmax 100
#include <stdio.h>
#include <stdlib.h>

int maior (int x, int y) {
    printf("\n>>>>chamada de maior com x = %5d e y = %5d", x, y);
    if (x >= y) return x; else return y;
}

int maximo_rekursiva (int a[], int n) {
    printf("\n####chamada de maximo_rekursiva com n = %3d", n);
    if (n == 1) return a[0];
    return maior (a[n-1], maximo_rekursiva(a, n-1));
}

int main() {
    int tabela[nmax];
    int n, i;
    /* ler n */
    printf("**** entre com n:");
    scanf("%d", &n);
    /* gerar e imprimir n números aleatórios inteiros entre 0 e 9999 */
    srand(1234);
    printf("\n***** numeros gerados\n");
    for (i = 0; i < n; i++)
        printf("%5d", tabela[i] = rand() % 10000);
    /* calcular e imprimir o máximo */
    printf("\n\n***** maximo dos elementos gerados:%5d\n",
        maximo_rekursiva(tabela, n));
}
```

**** entre com n:7

```
***** numeros gerados
4068 213 2761 8758 3056 7717 5274
####chamada de maximo_rekursiva com n = 7
####chamada de maximo_rekursiva com n = 6
####chamada de maximo_rekursiva com n = 5
####chamada de maximo_rekursiva com n = 4
####chamada de maximo_rekursiva com n = 3
####chamada de maximo_rekursiva com n = 2
####chamada de maximo_rekursiva com n = 1
>>>>chamada de maior com x = 213 e y = 4068
>>>>chamada de maior com x = 2761 e y = 4068
>>>>chamada de maior com x = 8758 e y = 4068
```

```
>>>>chamada de maior com x = 3056 e y = 8758  
>>>>chamada de maior com x = 7717 e y = 8758  
>>>>chamada de maior com x = 5274 e y = 8758
```

```
***** maximo dos elementos gerados: 8758
```

Outra forma recursiva de entender a função `maximo`:

- O máximo de uma tabela de n elementos, é o maior entre o primeiro e o máximo entre os $n-1$ elementos posteriores.

Exercício – refaça a função recursivamente usando a definição acima. Neste caso, como no exemplo acima de busca sequencial, use ponteiros para a tabela.

Forma recursiva versus forma repetitiva

Em todos os exemplos acima, a versão recursiva, embora às vezes mais elegante, pois é mais aderente à definição da função, não é mais eficiente que a versão iterativa. Em alguns casos, no entanto, pensar na solução de forma recursiva, facilita muito.

Listas ligadas e recursão

Listas ligadas simples (não circulares e sem sentinela) são estruturas claramente recursivas. Por isso, admitem também algoritmos recursivos. Cada nó de uma lista ligada pode ser entendido como um ponteiro para uma nova lista ligada. Vejamos uma definição informal:

Definição:

```
Lista Ligada = {  
    NULL ou  
    Elemento contendo um campo de informação e um ponteiro para uma Lista Ligada  
}
```

Algumas funções recursivas com lista ligada

```
/* Função conta(x) que conta o número de elementos de uma  
   lista apontada por x. */  
int conta(link x) {  
    if (x == NULL) return 0;  
    return 1 + conta(x -> prox)  
}
```

```
/* Função procura(x,v) que procura elemento com info = v  
   na lista apontada por x. */  
link procura(link x, int v) {  
    if (x == NULL) return NULL;  
    if (x -> info == v) return x;  
    return procura(x -> prox, v);  
}
```

```
}

/* Função compara(x, y ) que compara o conteúdo (campo info)
   de duas listas ligadas apontas por x e y respectivamente.
   Devolve 0 se forem iguais, 1 se x>y e -1 se x<y. */
int compara(link x, link y) {
    if (x == NULL && y == NULL) return 0;
    if (x == NULL) return -1; /* x < y */
    if (y == NULL) return 1; /* x > y */
    /* ambos são não NULL */
    if (x -> info == y -> info)
        return compara(x -> prox, y -> prox);
    if (x -> info > y -> info) return 1;
    return -1;
}

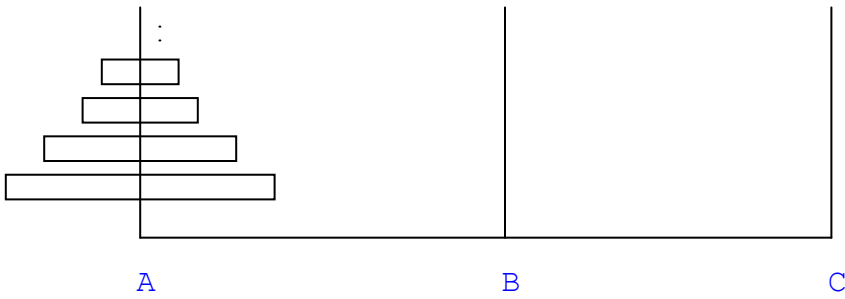
/* Função remove(x,v) que remove item com info = v da lista
   apontada por x. Solução um pouco artificial. */
link remove(link x, int v) {
    if (x == NULL) return NULL;
    if (x -> info == v) {
        link t;
        /* retorna o próximo elemento da lista */
        t = x -> prox;
        free(x);
        return t;
    }
    x -> prox = remove(x -> prox, v);
    return x;
}
```

Torres de Hanói

Um exemplo é o problema conhecido como Torres de Hanói.

O problema consiste em mover n discos empilhados (os menores sobre os maiores), de uma haste de origem (A), para uma haste de destino (B), na mesma ordem, respeitando as seguintes regras:

- 1) Apenas um disco pode ser movido por vez
- 2) Não colocar um disco maior sobre um menor
- 3) Pode usar uma haste auxiliar (C)



Vamos numerar os discos 1, 2, 3, ..., n-1, n. O disco 1 é o menor.

A solução para 2 discos seria:

```

move disco 1 da torre A para a torre C
move disco 2 da torre A para a torre B
move disco 1 da torre C para a torre B
    
```

Correspondente aos movimentos abaixo:

1				1	
2		2	1	2	
A	B	C	A	B	C

Vamos agora resolver para 3 discos “pensando de forma recursiva”.
 Já sabemos fazer para 1 (trivial) e para 2 discos (solução acima).

Bastam as seguintes operações:

- 1) Mover os 2 discos superiores de A para C usando B como auxiliar (já sabemos fazer isso)
- 2) Mover o disco 3 de A para B
- 3) Mover os 2 discos superiores de C para B usando A como auxiliar (já sabemos fazer isso)

Correspondem aos movimentos abaixo:

1						1		
2				1		2		
3			3	2	3	3		
A	B	C	.	.	.	A	B	C

Detalhando os movimentos:

```

move disco 1 da torre A para a torre B
move disco 2 da torre A para a torre C
move disco 1 da torre B para a torre C
move disco 3 da torre A para a torre B
move disco 1 da torre C para a torre A
    
```

```
move disco 2 da torre C para a torre B
move disco 1 da torre A para a torre B
```

Vamos agora resolver para n discos “pensando de forma recursiva”. Já sabemos fazer para 1 (trivial) e para 2 e para 3 discos (acima). Vamos supor que sabemos resolver para $n-1$ discos.

Bastam as seguintes operações:

- 1) Mover os $n-1$ discos superiores de A para C usando B como auxiliar (já sabemos fazer isso)
- 2) Mover o disco n de A para B
- 3) Mover os $n-1$ discos superiores de C para B usando A como auxiliar (já sabemos fazer isso)

Correspondem aos movimentos abaixo:

1						1		
2			1		1	2		
3			2		2	3		
...			3		3	...		
$n-1$			$n-1$		
n		n	$n-1$	n	$n-1$	n		
A	B	C	.	.	.	A	B	C

Temos então uma solução recursiva para o problema, ou seja, se sabemos como mover $n-1$ discos, podemos construir a solução para n discos. O caso trivial ou a regra de parada ocorre quando n é 1.

Usando a solução acima a função ficaria:

```
void hanoi(int n, char a, char b, char c) {
    if (n == 1) printf("\n move disco %3d da torre %3c para a torre %3c", 1, a, b);
    else {
        hanoi(n - 1, a, c, b);
        printf("\n move disco %3d da torre %3c para a torre %3c", n, a, b);
        hanoi(n - 1, c, b, a);
    }
}
```

O programa abaixo usa a função `hanoi` recursiva:

Dado o número de discos n e o nome das torres de origem, destino e auxiliar, resolver o problema das Torres de Hanói para n discos.

```
#include <stdio.h>

void hanoi(int n, char a, char b, char c) {
    if (n == 1) printf("\n move disco %3d da torre %3c para a torre %3c", 1, a, b);
    else {
        hanoi(n - 1, a, c, b);
    }
}
```



```
        printf("\n move disco %3d da torre %3c para a torre %3c", n, a, b);  
        hanoi(n - 1, c, b, a);  
    }  
}  
  
int main() {  
    int n;  
  
    printf("entre com o numero de discos:");  
    scanf("%d", &n);  
  
    /* chama a função para movimentar os discos */  
    hanoi(n, 'A', 'B', 'C');  
    system("pause");  
    return 0;  
}
```

Veja o que será impresso:

entre com o numero de discos:4

```
move disco 1 da torre A para a torre C  
move disco 2 da torre A para a torre B  
move disco 1 da torre C para a torre B  
move disco 3 da torre A para a torre C  
move disco 1 da torre B para a torre A  
move disco 2 da torre B para a torre C  
move disco 1 da torre A para a torre C  
move disco 4 da torre A para a torre B  
move disco 1 da torre C para a torre B  
move disco 2 da torre C para a torre A  
move disco 1 da torre B para a torre A  
move disco 3 da torre C para a torre B  
move disco 1 da torre A para a torre C  
move disco 2 da torre A para a torre B  
move disco 1 da torre C para a torre B
```

Exercício: Mostre que para N discos, o algoritmo acima precisa de $2^N - 1$ movimentos.

A Regra de Horner

Considere o cálculo do valor de um polinômio. Dados x , $a[n]$, $a[n-1]$, ..., $a[0]$ calcular $P(n, x) = a[n]x^n + a[n-1]x^{n-1} + \dots + a[1]x + a[0]$.

Por exemplo, para $n = 4$ o cálculo seria:

$a[4]x^4 + a[3]x^3 + a[2]x^2 + a[1]x + a[0]$ que pode ser escrito pela regra de Horner como:

$((a[4]x + a[3])x + a[2])x + a[1])x + a[0]$

Com quatro multiplicações e quatro somas calculamos o valor. No caso geral com n multiplicações e com n somas se calcula o valor do polinômio para um dado x .

Abaixo uma definição iterativa:

```
double P(double x, double a[], int n) {  
    double s = a[n];  
    while (--n >= 0) s = s * x + a[n];  
    return s;  
}
```

Observe que:

$a[n]x^n + \dots + a[0] = x(a[n]x^{n-1} + \dots + a[1]) + a[0]$ **se $n > 0$**
 $a[0]$ **se $n = 0$**

Ou seja:

$P(x, a, 0, n) = x * P(x, a, 1, n-1) + a[0]$ se $n > 0$
 $a[0]$ se $n = 0$

Note que foi necessária a inclusão de um parâmetro adicional em P para refletirmos a recursão. Com isso a versão recursiva da função fica:

```
double P(double x, double a[], int k, int n) {  
    if (n > 0) return x * P(x, a, k+1, n-1) + a[k];  
    return a[n-1];  
}
```

Usando ponteiros para vetores, podemos economizar um parâmetro:

```
double P(double x, double * a, int n) {  
    if (n > 0) return x * P(x, a + 1, n - 1);  
    return *a;  
}
```

Potência com expoente inteiro

Considere o cálculo de x^n com n inteiro e não negativo. Embora a função intrínseca **pow(x, y)** calcule x^y , o faz para x e y genéricos. Vamos tentar achar maneiras melhores para o caso particular acima.

Seja $f(x, n) = x^n$ ($n \geq 0$ inteiro).

$f(x, n) = 1$ se $n = 0$ e $x * f(x, n-1)$ se $n > 0$

Exercício: Escreva a função acima nas versões iterativa e recursiva.

Observe que são feitas n multiplicações na definição acima. Será que é possível diminuir? Veja que:

$$x^4 = x \cdot x \cdot x \cdot x = x^2 \cdot x^2 = (x^2)^2$$
$$x^5 = x \cdot x \cdot x \cdot x \cdot x = x^2 \cdot x^2 \cdot x = (x^2)^2 \cdot x$$

Podemos fazer menos multiplicações usando produtos anteriores.

Veja a seguinte definição:

$$x^n = 1 \text{ se } n = 0$$
$$(x^h)^2 \text{ se } n > 1 \text{ e par}$$
$$(x^h)^2 \cdot x \text{ se } n > 1 \text{ e ímpar}$$

Onde h é o quociente de n/2 truncado.

Veja a seguir:

```
double Pot(double x, int n) {
    if (n == 0) return 1;
    double k = Pot(x, n / 2);
    if (n % 2 == 0) return k * k;
    return k * k * x;
}
```

Exercício: Quantas multiplicações são feitas para se calcular x^n no algoritmo acima ?

Exercício: Refaça os 2 exercícios anteriores expandindo a definição para n negativo.

$$x^n = 1 / x^{-n} \text{ se } n < 0. \text{ Se } n < 0 \text{ e } x = 0 \text{ a função não está definida.}$$

Exercício: Dado n e uma seqüência com n números, imprimir a seqüência na ordem inversa a que foi lida. Fazer isso sem usar vetor. Sugestão: faça uma função recursiva `imprime`, que lê um número, chama a si própria se não chegou ao fim da seqüência e imprime o número lido.

Exercício: Idem para uma seqüência terminada por zero.

Exercício: Considere o cálculo do número de combinações de n elementos k a k, isto é o número de subconjuntos de k elementos dentre n, denominado $C(n, k)$.

$$C(n, k) = n! / (k! \cdot (n-k)!)$$

Outra forma de calcular $C(n, k)$ é $C(n, k) = C(n-1, k-1) + C(n-1, k)$ quando n e k são ambos maiores que zero. Essa é a forma do triângulo de Pascal e é claramente recursiva. Faça uma função recursiva para calcular $C(n, k)$.