


## Funções Recursivas

- **Recursividade** é uma forma interessante de resolver problemas. Essa forma se aplica quando um problema pode ser dividido em problemas menores (**subproblemas**) de **mesma natureza** que o problema original.
- Como os subproblemas têm a mesma natureza do problema original, o **mesmo método** usado para reduzir o problema pode ser usado para reduzir os subproblemas.
- Mas, até quando devemos dividir o problema em subproblemas? **Quando parar de reduzir?** Quando o subproblema obtido for um caso trivial, para o qual se conhece a solução.
- Se o método usado para reduzir o problema for implementado como uma **função**, usar o mesmo método é chamar a função dentro da própria função.

- **Exemplo:** Calcular  $10!$

$$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$




O que é isso?

É **9!**

- Então:  $10! = 10 \times 9!$
- O problema de calcular  $10!$  reduziu-se ao problema de calcular  $9!$
- Mas  $9! = 9 \times 8!$
- Até quando devemos reduzir o problema? Qual é o caso trivial de cálculo do fatorial? É a definição de  $0! = 1$ .
- Então podemos escrever:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

- Como escrever a função `int fat(int n)`?

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

Observar que a função **fat** **chama** a si mesma!

- Uma função que chama (usa) a si mesma é denominada **função recursiva**.
- Como funciona essa função fat? A função usa um mecanismo conhecido como **pilha de execução**!

- Numa pilha de execução, os **resultados parciais** são **empilhados** e são desempilhados somente quando é possível realizar o cálculo do resultado (ou quando se chega ao caso trivial).

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

### Pilha de Execução

<b>fat(0)</b>
<b>fat(1)</b>
<b>fat(2)</b>
<b>fat(3)</b>
<b>fat(4)</b>

return 1 (caso trivial)

return 1\*fat(0)

return 2\*fat(1)

return 3\*fat(2)

return 4\*fat(3)

### Resultados

<b>1</b>
<b>1*1 = 1</b>
<b>2*1 = 2</b>
<b>3*2 = 6</b>
<b>4*6 = 24</b>

- **Exercício:** Escrever uma função recursiva `int soma(int n)`, que retorna a soma dos `n` primeiros números inteiros positivos.

- Definição recursiva:

$$\text{soma}(n) = \begin{cases} 0 & \text{se } n = 0 \\ n + \text{soma}(n-1) & \text{se } n > 0 \end{cases}$$

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

int main()
{
    int n,s;
    printf("n = ");
    scanf("%d",&n);
    s = soma(n);
    printf("Soma = %d\n",s);
    return 1;
}
```

Algumas execuções:

n = 5  
Soma = 15

n = 10  
Soma = 55

n = 20  
Soma = 210

- **Exercício:** Escrever a função recursiva `int soma(int n)`, que retorna a soma dos  $n$  primeiros inteiros ímpares.

- Definição recursiva: 
$$\text{soma}(n) = \begin{cases} 1 & \text{se } n = 1 \\ (2 * n - 1) + \text{soma}(n - 1) & \text{se } n > 1 \end{cases}$$

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

int main()
{
    int n,s;
    printf("n = ");
    scanf("%d",&n);
    s = soma(n);
    printf("Soma = %d\n",s);
    return 1;
}
```

Algumas execuções:

n = 6  
Soma = 36

n = 10  
Soma = 100

n = 20  
Soma = 400

## Exercícios

1. Escrever a função recursiva `float pot(float x, int n)`, que retorna  $x^n$ .

- Definição recursiva:

$$\text{pot}(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ x * \text{pot}(x, n - 1) & \text{se } n > 0 \end{cases}$$

2. Usando as funções recursivas `fat` e `pot`, escrever a função recursiva `float somatoria(float x, int n)` que retorna:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Comparar o resultado com a versão iterativa do cálculo da somatória.

- Definição recursiva:

$$\text{somatoria}(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ \text{pot}(x, n) / \text{fat}(n) + \text{somatoria}(x, n - 1) & \text{se } n > 0 \end{cases}$$

```
float pot(float x, int n)
{
    if (n == 0)
        return 1;
    else
        return x*pot(x,n-1);
}
```

```
float somaiter(float x, int n)
{
    int i;
    float s = 0;
    for (i = 0; i <= n; i++)
        s = s + pot(x,i)/fat(i);
    return s;
}

int main()
{
    int n;
    float x,s;
    printf("x e n: ");
    scanf("%f %d",&x,&n);
    s = somaiter(x,n);
    printf("ite = %.4f\n",s);
    return 1;
}
```

```
float somatoria(float x, int n)
{
    if (n == 0)
        return 1;
    else
        return pot(x,n)/fat(n)+somatoria(x,n-1);
}

int main()
{
    int n;
    float x,s;
    printf("x e n: ");
    scanf("%f %d",&x,&n);
    s = somatoria(x,n);
    printf("rec = %.4f\n",s);
    return 1;
}
```

Execuções:

```
x e n: 0.5 10
rec = 1.6487
ite = 1.6487
```



- **Exercício:** Escrever a função recursiva `int fib(int n)`, que retorna o n-ésimo número da sequência de Fibonacci:

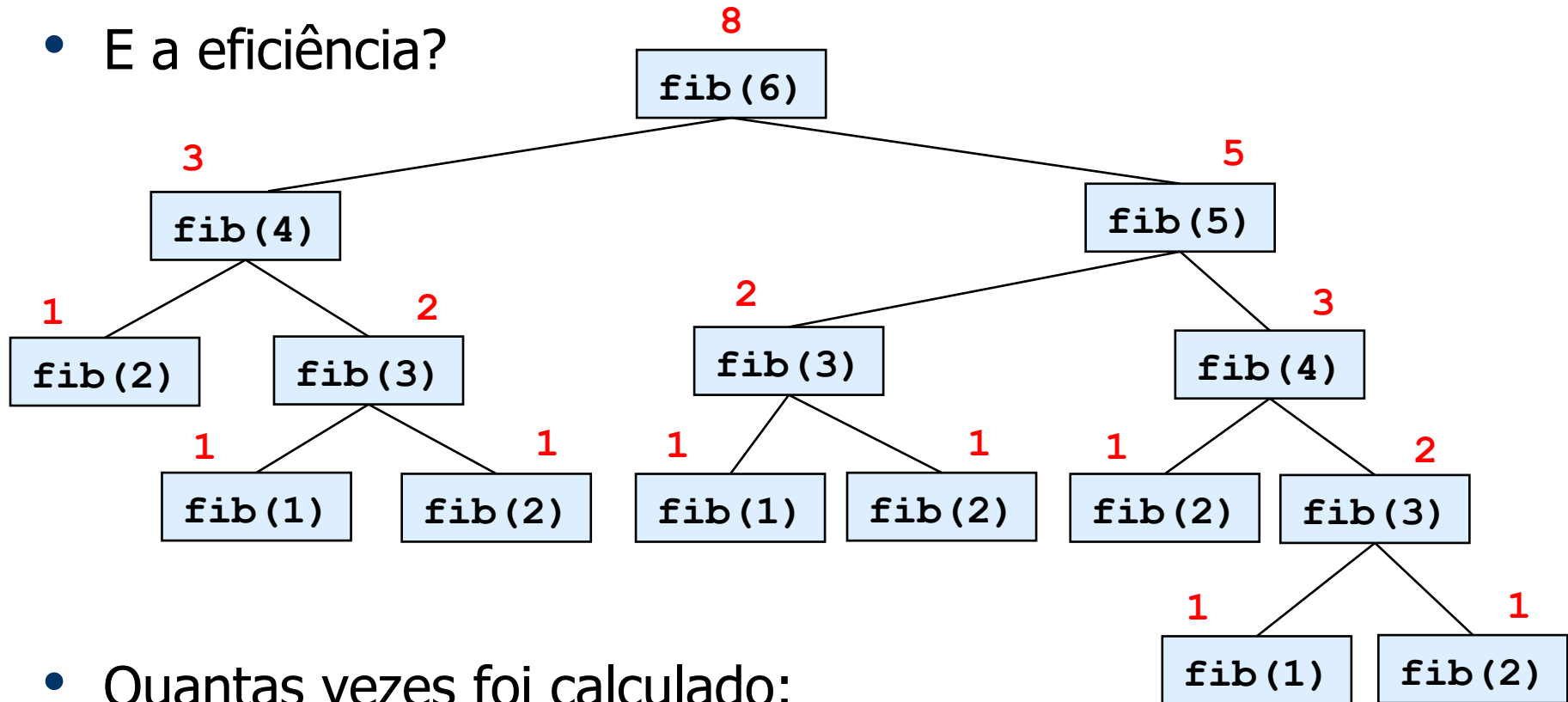
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- Definição recursiva:

$$\text{fib}(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{se } n > 2 \end{cases}$$

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

- E a eficiência?



- Quantas vezes foi calculado:
  - `fib(1)`: 3 vezes
  - `fib(2)`: 5 vezes
  - `fib(3)`: 3 vezes
  - `fib(4)`: 2 vezes

**Conclusão:** A versão recursiva da função `fib` é **muito ineficiente**, uma vez que recalcula o mesmo valor várias vezes.

- A versão iterativa é mais eficiente:

```
int fib(int n)
{
    int i,a,b,c;
    if ((n == 1) || (n == 2))
        return 1;
    else
    {
        a = 0;
        b = 1;
        for (i = 3; i <= n; i++)
        {
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

- Alguns resultados:

n	10	20	30	50	100
<b>recursivo</b>	8 ms	1 s	2 min	21 dias	10 <sup>9</sup> anos
<b>iterativo</b>	0.17 ms	0.33 ms	0.50 ms	0.75 ms	1.50 ms

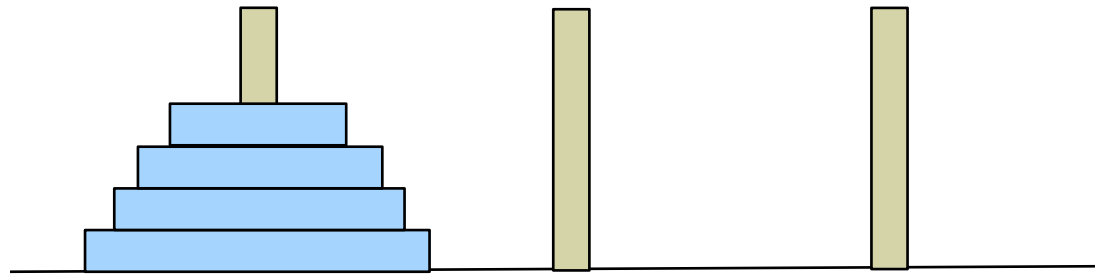
**Fonte:** Livro [Fundamentals of Algorithmics](#) (p. 73), dos autores Gilles Brassard e Paul Bradley. Editora Pearson, 1996.



- Portanto, um algoritmo recursivo nem sempre é o melhor caminho para se resolver um problema.
- Na maioria das vezes, no entanto, a recursividade torna o algoritmo mais simples.

## Desafio:

- O jogo Torre de Hanoi surgiu na Europa no final do século XIX. O jogo consiste em um conjunto de  $n$  discos de tamanhos distintos dispostos em 3 pinos. Por exemplo, para  $n = 4$ :



- O jogo consiste em transferir todos os discos do pino 0 (mais à esquerda) para o pino 2 (mais à direita), obedecendo as seguintes regras:
  - Somente um disco pode ser movido de cada vez;
  - Em nenhum estágio durante o jogo, pode-se ter um disco maior colocado sobre um disco menor.

- Seja a função `void mover(int n, int a, int b, int c)` para mover  $n$  discos do pino  $a$  para o pino  $b$  usando o pino  $c$  como intermediário. Com essa função, para resolver o problema da figura anterior basta chamar: `mover(4,1,3,2)`.
- A função `mover` pode ser escrita de forma recursiva como:
  - a) Mover  $n-1$  discos do pino  $a$  para o pino  $c$ ;
  - b) Mover 1 disco do pino  $a$  para o pino  $b$ ;
  - c) Mover  $n-1$  discos do pino  $c$  para o pino  $b$ .
- Observar que o passo (b) corresponde a um **problema trivial** e os passos (a) e (c) são subproblemas (passos **recursivos**).

```
void mover(int n, int a, int b, int c)
{
    if (n > 0)
    {
        mover(n-1, a, c, b);
        printf("%d -> %d\n", a, b);
        mover(n-1, c, b, a);
    }
}
```

- Versão recursiva:

```
void mover(int n, int a, int b, int c)
{
    if (n > 0)
    {
        mover(n-1, a, c, b);
        printf("%d -> %d\n", a, b);
        mover(n-1, c, b, a);
    }
}

int main()
{
    int n;
    printf("Numero de discos: ");
    scanf("%d", &n);
    printf("Solucao para %d discos:\n", n);
    mover(n, 1, 3, 2);
    return 1;
}
```

Numero de discos: 4  
Solucao para 4 discos:

1 -> 2  
1 -> 3  
2 -> 3  
1 -> 2  
3 -> 1  
3 -> 2  
1 -> 2  
1 -> 3  
2 -> 3  
2 -> 1  
3 -> 1  
2 -> 3  
1 -> 2  
1 -> 3  
2 -> 3

Para 4 discos foram necessários **15 movimentos** para resolver o problema.

Discos	Movimentos
10	1023
20	1048575
30	1073741823

- **Desafio:** Implementar a função **mover** de forma iterativa.

## O Algoritmo QuickSort

- É um dos melhores algoritmos de ordenação conhecidos. Consiste nos seguintes passos:
  1. Seja  $v$  um vetor. Sejam  $i$  e  $j$  os índices do primeiro e do último elementos de  $v$ .
  2. Seja  $vm = v[(i+j)/2]$ , o elemento "central" de  $v$ ;
  3. Enquanto ( $i < j$ ) fazer:
    - a) Enquanto ( $v[i] < vm$ ) aumentar  $i$ ;
    - b) Enquanto ( $v[j] > vm$ ) diminuir  $j$ ;
    - c) Trocar os elementos  $v[i]$  e  $v[j]$ .
- Ao final desses passos, teremos em relação à  $vm$ :
  - À esquerda, somente elementos menores;
  - À direita, somente elementos maiores.



Exemplo:

0	1	2	3	4	5	6	7	8	9
12	9	17	22	<b>15</b>	10	5	21	3	18

- $i = 0; j = 9; vm = v[(0 + 9)/2] = v[4] = 15;$

<b>i</b>				<b>j</b>					
12	9	3	22	<b>15</b>	10	5	21	17	18

<b>i</b>				<b>j</b>					
12	9	3	5	<b>15</b>	10	22	21	17	18

<b>i</b>			<b>j</b>						
12	9	3	5	10	<b>15</b>	22	21	17	18

<b>i</b>					<b>j</b>				
12	9	3	5	10	<b>15</b>	22	21	17	18

Elementos **menores**  
do que 15.

Elementos **maiores**  
do que 15.

**E agora?**  
Como ordenar  
os menores e  
os maiores?  
**Chamada  
recursiva.**

- Algoritmo QuickSort:

```
void quicksort(int v[], int L, int R)
{
    int i,j,vm;
    i = L;
    j = R;
    vm = v[(i+j)/2];
    do
    {
        while (v[i] < vm) i++;
        while (v[j] > vm) j--;
        if (i <= j)
        {
            troca(v,i,j);
            i++;
            j--;
        }
    } while (i <= j);
    if (L < j) quicksort(v,L,j);
    if (R > i) quicksort(v,i,R);
}
```

```
void troca(int v[], int i, int j)
{
    int aux;
    aux = v[i];
    v[i] = v[j];
    v[j] = aux;
}
```

- Exercício: Ordenação por seleção.

```
void selecao(int v[], int n)
{
    int k,p;
    k = 0;
    while (k < n)
    {
        p = menor(v,k,n-1);
        troca(v,k,p);
        k++;
    }
}
```

```
int menor(int v[], int L, int R)
{
    int i,m,vm;
    m = L;
    vm = v[m];
    for (i = L+1; i <= R; i++)
    {
        if (v[i] < vm)
        {
            vm = v[i];
            m = i;
        }
    }
    return m;
}
```

- Escrever a versão recursiva do algoritmo de ordenação por seleção.

```
void rselecao(int v[], int L, int R)
{
    int p = menor(v,L,R);
    troca(v,L,p);
    if (L < R)
        rselecao(v,L+1,R);
}
```

## Exercícios

1. Escrever a função recursiva `int max(int n, int *v)` que retorna o maior elemento do vetor `v` (de `n` elementos).
2. Escrever a função `float vpol(int n, int *c, float x)`, de forma recursiva, para calcular o valor do polinômio  $p_n(x) = c_0x^n + c_1x^{n-1} + \dots + c_{n-1}x + c_n$  para um dado `x`.
3. Dado `n` e uma sequência com `n` números inteiros, imprimir a sequência na **ordem inversa** a que foi lida, sem usar um vetor.
4. Escrever uma função recursiva que retorna o produto dos elementos estritamente positivos de um vetor `v` de `n` elementos inteiros. Considerar que `v` tem pelo menos um elemento  $> 0$ .
5. Qual é o valor retornado por `f(10)`?

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

## Exercício 1:

```
int maximo(int n, int *v)
{
    int m;
    if (n == 1)
        return v[0];
    else
    {
        m = maximo(n-1,v);
        if (m > v[n-1])
            return m;
        else
            return v[n-1];
    }
}
```

```
int main()
{
    int i,n,m;
    int *v;
    printf("n = ");
    scanf("%d",&n);
    v = (int *)calloc(n,sizeof(int));
    printf("v = ");
    for (i = 0; i < n; i++)
    {
        scanf("%d",&v[i]);
    }
    m = maximo(n,v);
    printf("Maximo = %d\n",m);
    return 1;
}
```

## Exercício 2:

Observar que um polinômio  $P_n(x)$  pode ser escrito como:

$$P_n(x) = c_0x^n + c_1x^{n-1} + c_2x^{n-2} + \dots + c_{n-1}x + c_n$$

Um método usual para o cálculo de  $P_n(x)$ , para um dado  $x$ , é a **regra de Horner**:

$$P_n(x) = (((\dots (c_0x + c_1)x + c_2)x + \dots)x + c_{n-1})x + c_n$$

**Exemplo:**  $P_3(x) = c_0x^3 + c_1x^2 + c_2x + c_3$

pode ser escrito como:  $P_3(x) = ((c_0x + c_1)x + c_2)x + c_3$

A regra de Horner pode ser expressa de **forma recursiva** como:

$$P_n(x) = P_{n-1}(x) * x + c_n$$

```
float vpol(int n, int *c, float x)
{
    if (n > 0)
        return x*vpol(n-1,c,x) + c[n];
    else
        return c[0];
}
```

```
int main()
{
    int i,n;
    int *c;
    float v,p;
    printf("n = ");
    scanf("%d",&n);
    c = (int *)calloc(n+1,sizeof(int));
    printf("c = ");
    for (i = 0; i <= n; i++)
    {
        scanf("%d",&c[i]);
    }
    printf("v = ");
    scanf("%f",&v);
    p = vpol(n,c,v);
    printf("p(%f) = %f\n",v,p);
    return 1;
}
```

## Exercício 3:

```
void sequencia(int n)
{
    int x;

    if (n == 0)
        return;

    printf("x = ");
    scanf("%d",&x);
    sequencia(n-1);
    printf("%d ",x);
}
```

```
int main()
{
    int n;

    printf("n = ");
    scanf("%d",&n);
    sequencia(n);
    printf("\n");
    return 1;
}
```



## Exercício 4:

```
int prod(int n, int *v)
{
    if (n == 1)
    {
        if (v[0] > 0)
            return v[0];
        else
            return 1;
    }
    else
    {
        if (v[n-1] > 0)
            return v[n-1]*prod(n-1,v);
        else
            return prod(n-1,v);
    }
}
```

```
int main()
{
    int i,n,p;
    int *v;
    printf("n = ");
    scanf("%d",&n);
    v = (int *)calloc(n,sizeof(int));
    printf("v = ");
    for (i = 0; i < n; i++)
    {
        scanf("%d",&v[i]);
    }
    p = prod(n,v);
    printf("p = %d\n",p);
    return 1;
}
```

Fim do curso!  
Obrigado pela atenção.  
Boa sorte a todos!