# Programação Estruturada

Recursão

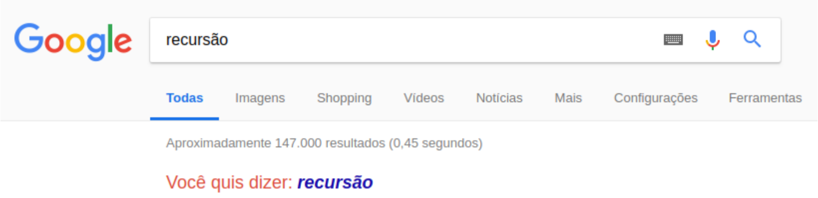

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

# Recursão

# Recursão


Google
recursão
Todas
Imagens
Shopping
Vídeos
Notícias
Mais
Configurações
Ferramentas
Aproximadamente 147.000 resultados (0,45 segundos)
Você quis dizer: recursão

# Recursão

# Recursão

O que os seguintes problemas têm em comum?

- **Fatorial**

$$F_i = \begin{cases} 1 & \text{se } i = 0 \\ i \times F_{i-1} & \text{caso contrário} \end{cases}$$

- O $i$-ésimo elemento da **Sequência de Fibonacci ($F_i$)**

$$F_i = \begin{cases} i & \text{se } i < 2 \\ F_{i-1} + F_{i-2} & \text{caso contrário} \end{cases}$$

- Máximo divisor comum (**MDC**)

$$MDC(a, b) = \begin{cases} a & \text{se } b = 0 \\ MDC(b, a \bmod b) & \text{caso contrário} \end{cases}$$

# Recursão

- **Fatorial**

```c
int fatorial(int i) {
    if (i == 0) return 1;
    return i * fatorial(i - 1);
}
```

- O *i*-ésimo elemento da **Sequência de Fibonacci ($F_i$)**

```c
int fib(int i) {
    if (i < 2) return i;
    return fib(i - 1) + fib(i - 2);
}
```

- Máximo divisor comum (**MDC**)

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

# Recursão

- Um objeto é denominado recursivo quando sua definição é parcialmente feita em termos dele mesmo.
- Em programação, a recursividade é um mecanismo útil e poderoso que permite a uma função chamar a si mesma direta ou indiretamente.
- A ideia básica de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que o possamos resolver o problema reduzido de forma direta.
    - Quando isso ocorre, atingimos uma **condição de parada**.

# Indução

# Indução

- Usando o método de indução, a solução de um problema pode ser expressa da seguinte forma:
  - Primeiramente, definimos a solução para casos básicos;
  - Em seguida, definimos como resolver o problema para um caso geral, utilizando-se de soluções para instâncias menores do problema.

# Indução

- **Indução:** Técnica de demonstração matemática onde algum parâmetro da proposição a ser demonstrada envolve números naturais.
- Seja $T(n)$ uma proposição que desejamos provar como verdadeira para todos valores naturais $n$.
- Ao invés de provar diretamente que $T(n)$ é válida para todos os valores de $n$, basta:
  1. **Caso base:** Provar que $T(1)$ é válido.
  2. **Hipótese de Indução:** Assumir que $T(n-1)$ é válida.
  3. **Passo de indução:** Provar que $T(n)$ é válida.

- Por que a indução funciona?
  - Mostramos que $T(1)$ é valida.
  - Com o passo da indução, automaticamente mostramos que $T(2)$ é válida.
  - Como $T(2)$ é válida, pelo passo de indução, $T(3)$ também é válida.
  - E assim por diante...

# Indução

- **OBS:** O caso base não precisa ser necessariamente com $n = 1$.
- Você pode considerar um caso inicial $n = c$ para uma constante $c$ qualquer.
- Se você mostrar que este caso base é valido e o passo também é valido: sua proposição é verdadeira para todo $n \geq c$.

# Exemplo

**Teorema**

$2^{2n} - 1$ é múltiplo de 3 para $n \geq 0$.

**Base:** Para $n = 0$ temos que $2^{2n} - 1 = 0$, que é múltiplo de 3.

**Hipótese:** O teorema é válido para $n - 1$, ou seja, $2^{2(n-1)} - 1$ é múltiplo de 3.

**Passo:** Devemos provar que $2^{2n} - 1$ é múltiplo de 3. Para tanto, vamos usar a hipótese. Note que

$2^{2n} - 1 = 2^{2n-2}2^2 - 1 = 4(2^{2(n-1)}) - 1 = 3(2^{2(n-1)}) + 2^{2(n-1)} - 1.$

Note que $3(2^{2(n-1)})$ é múltiplo de 3 e, por hipótese, $2^{2(n-1)} - 1$ também é múltiplo de 3. Portanto,

$$3(2^{2(n-1)}) + 2^{2(n-1)} - 1 = 2^{2n} - 1$$

é múltiplo de 3.

# Exemplo

**Teorema**

A soma $S(n)$ dos primeiros $n$ números naturais é $n(n+1)/2$

**Base:** Para $n = 1$ devemos mostrar que $n(n+1)/2 = 1$. Isto é verdade: $1(1+1)/2 = 1$.

**Hipótese:** Vamos assumir que é válido para $(n-1)$, ou seja, $S(n-1) = (n-1)((n-1)+1)/2$.

**Passo:** Devemos mostrar que é válido para $n$, ou seja, devemos mostrar que $S(n) = n(n+1)/2$. Por definição, $S(n) = S(n-1) + n$ e por hipótese $S(n-1) = (n-1)((n-1)+1)/2$. Logo,

$$
\begin{aligned}
S(n) &= S(n-1) + n \\
&= (n-1)((n-1)+1)/2 + n \\
&= n(n-1)/2 + 2n/2 \\
&= n(n+1)/2
\end{aligned}
$$

# Recursão

# Recursão

- Definições recursivas de funções funcionam como o *princípio matemático da indução* que vimos anteriormente.
- A ideia é que a solução de um problema pode ser expressa da seguinte forma:
    - Definimos a solução para casos básicos;
    - Definimos como resolver o problema geral utilizando soluções do mesmo problema só que para casos menores.

# Exemplo: fatorial

**Problema**

Calcular o fatorial de um número $n$ ($n!$).

Qual o caso base? Se $n$ é igual a 1, então o fatorial é 1.

Qual seria o passo indutivo?

Temos que expressar a solução para $n > 1$, supondo que já sabemos a solução para algum caso mais simples:

$$n! = n \times (n - 1)!$$

# Fatorial

Portanto, a solução do problema **pode ser expressa de forma recursiva** como:

- Se $n = 1$, então $n! = 1$.
- Se $n > 1$, então $n! = n \times (n-1)!$.

Note como aplicamos o princípio da indução:

- Sabemos a solução para um caso base: $n = 1$.
- Definimos a solução do problema geral $n!$ em termos do mesmo problema só que para um caso menor $(n-1)!$.

# Fatorial em C

```c
long int fatorial(int n) {
    long int r, x;

    /* caso base: */
    if (n == 1)
        return 1;
    else {
        /* sabendo o fatorial de n-1: */
        x = n-1;
        r = fatorial(x);
        /* calculamos o fatorial de n: */
        return n * r;
    }
}
```

# Funções recursivas

- Para solucionar o problema, é feita uma chamada para a própria função.
- Por isso, esta função é chamada *recursiva*.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

# O que acontece na memória

# O que acontece na memória

- Precisamos entender como é feito o controle sobre as variáveis locais em chamadas recursivas.
- A memória de um sistema computacional é dividida em alguns segmentos:
  - **Espaço Estático**: Contém as variáveis globais e código do programa.
  - **Heap:** Para alocação dinâmica de memória.
  - **Pilha:** Para execução de funções.

# O que acontece na memória

O que acontece na pilha:

- Toda vez que uma função é invocada, suas variáveis locais são armazenadas no topo da pilha.
- Quando uma função termina a sua execução, suas variáveis locais são removidas da pilha.

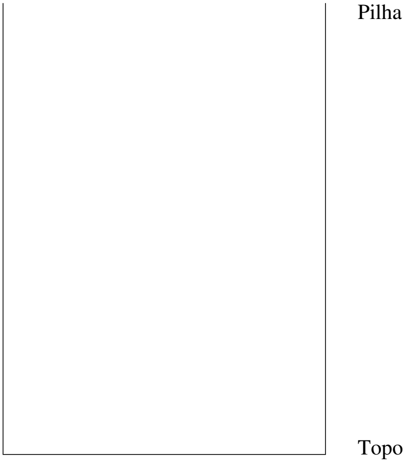
# O que acontece na memória

Considere o exemplo:

```c
int f1(int a, int b) {
    int c = 5;
    return c + a + b;
}

int f2(int a, int b) {
    int c;
    c = f1(b, a);
    return c;
}

int main() {
    f2(2, 3);
    return 0;
}
```

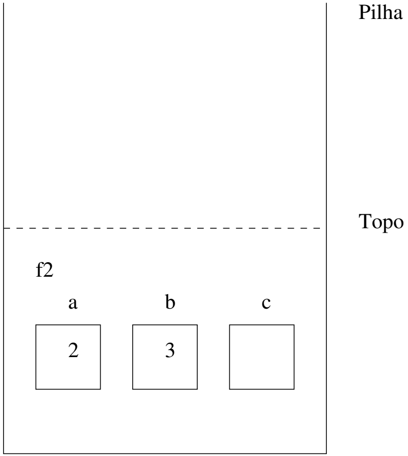
# O que acontece na memória

Inicialmente a pilha está vazia.



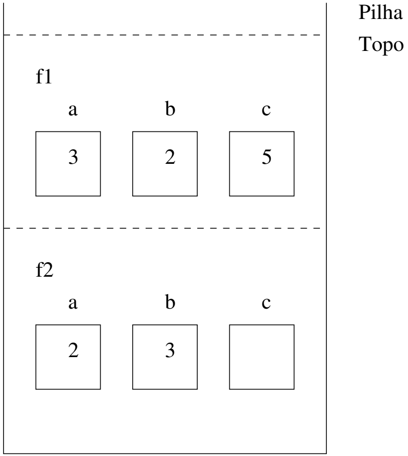
# O que acontece na memória

Quando `f2(2, 3)` é invocada, as variáveis locais de `f2` são alocadas no topo da pilha.



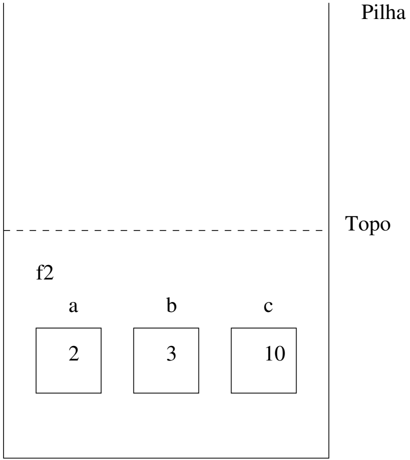
# O que acontece na memória

A função `f2` invoca a função `f1(b, a)` e as variáveis locais desta são alocadas no topo da pilha, sobre as de `f2`.



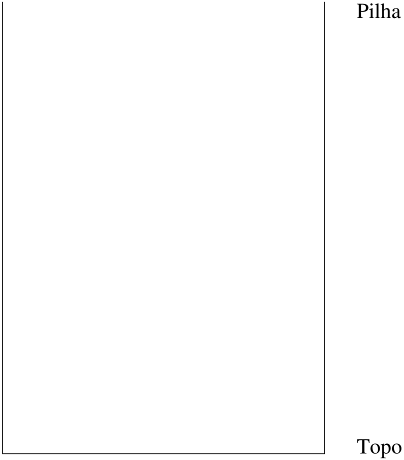
# O que acontece na memória

A função `f1` termina, devolvendo 10. As variáveis locais de `f1` são removidas da pilha.



## O que acontece na memória

Finalmente, `f2` termina a sua execução devolvendo 10. Suas variáveis locais são removidas da pilha.



# O que acontece na memória

No caso de chamadas recursivas para uma mesma função, é como se cada chamada correspondesse a uma função distinta.

- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.
- O último conjunto de variáveis alocadas na pilha, que está no topo, corresponde às variáveis da última chamada da função.
- Quando termina a execução de uma chamada da função, as variáveis locais desta são removidas da pilha.
- Sem uma condição de parada, o algoritmo não para de chamar a si mesmo, até estourar a capacidade da pilha.

# Usando recursão em programação

Considere novamente a solução recursiva para se calcular o fatorial e assuma que seja feito a chamada `fatorial(4)`.

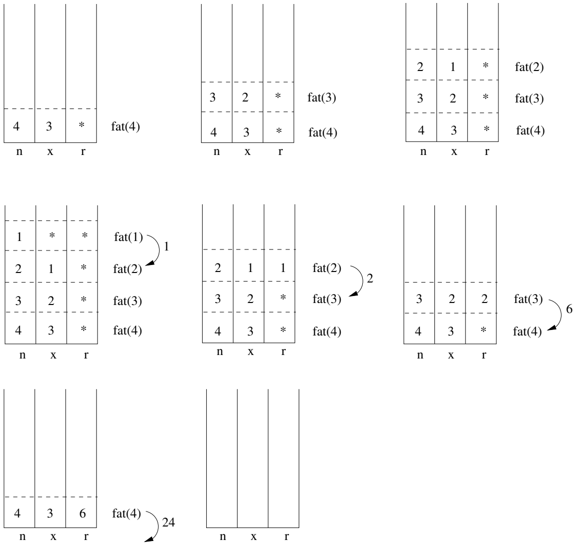
```
long int fatorial(int n) {
    long int r, x;

    /* caso base: */
    if (n == 1)
        return 1;
    else {
        /* sabendo o fatorial de n-1: */
        x = n-1;
        r = fatorial(x);
        /* calculamos o fatorial de n: */
        return n * r;
    }
}
```

# O que acontece na memória

- Cada chamada à função `fatorial` cria novas variáveis locais de mesmo nome (`n`, `x` e `r`).
- Portanto, várias variáveis `n`, `x` e `r` podem existir em um dado momento.
- Em um dado instante, o nome `n` (ou `r`, ou `x`) refere-se à variável local ao corpo da função que está sendo executada naquele instante.

# O que acontece na memória

Estado da pilha de execução para `fatorial(4)`:



# O que acontece na memória

- É claro que as variáveis `r` e `x` são desnecessárias.
- E você também deveria testar se `n` não é negativo!

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

```
long int fatorial(int n) {
    if (n <= 1)
        return 1;
    return n * fatorial(n-1);
}
```

# Recursão $\times$ Iteração

# Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas.
- Soluções iterativas em geral têm a memória limitada enquanto as recursivas, não.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.

# Recursão × Iteração

No caso do cálculo do fatorial, uma solução iterativa é mais eficiente. Por quê?

```c
long int fatorial(int n) {
    long int result = 1;
    int i;

    for (i = 1; i <= n; i++)
        result = result * i;

    return r;
}
```

## Recursão com várias chamadas

- Não há necessidade da função recursiva ter apenas uma chamada para si própria.
- A função pode fazer várias chamadas para si própria.
- A função pode ainda fazer chamadas recursivas indiretas: a função 1, por exemplo, chama uma outra função 2 que por sua vez chama a função 1.

# Fibonacci

- A série de Fibonacci é a seguinte: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$.
- Queremos determinar qual é o $n$-ésimo número da série, que denotaremos por $F(n)$.
- Como descrever o $n$-ésimo número de Fibonacci de forma recursiva?

# Fibonacci

- No caso base, temos: se $n = 1$ ou $n = 2$, então $F(n) = 1$.
- Sabendo casos anteriores podemos computar $F(n)$:

$$F(n) = F(n-1) + F(n-2) \ .$$

# Fibonacci: algoritmo em C

A definição anterior é traduzida diretamente em um algoritmo em C:

```c
long int fibonacci(int n) {
    if (n <= 2)
        return 1;

    return fibonacci(n-1) + fibonacci(n-2);
}
```

# Cálculo de potências

Suponha que temos que calcular $x^n$ para $n$ inteiro positivo.

Como calcular de forma recursiva?

$x^n$ é:

- 1, se $n = 0$.
- $xx^{n-1}$, caso contrário.

# Cálculo de Potências

```
long int pot(long int x, long int n) {
    if (n == 0)
        return 1;

    return x * pot(x, n-1);
}
```

## Cálculo de Potências

Neste caso a solução iterativa é mais eficiente:

```
long int pot(long int x, long int n) {
    long int result = 1, i;

    for (i = 1; i <= n; i++)
        result = result * x;

    return result;
}
```

- O laço é executado *n* vezes.
- Na solução recursiva são feitas *n* chamadas recursivas, mas tem-se o custo adicional para criação/remoção de variáveis locais na pilha.

## Cálculo de Potências

Mas e se definirmos a potência de forma diferente?

$x^n$ é:

- se $n = 0$, então $x^n = 1$.
- se $n > 0$ e $n$ é par, então $x^n = (x^{n/2})^2$.
- se $n > 0$ e $n$ é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note que aqui também definimos a solução do caso maior em termos de casos menores.

# Cálculo de Potências

Este algoritmo é mais eficiente do que o iterativo. Por quê?

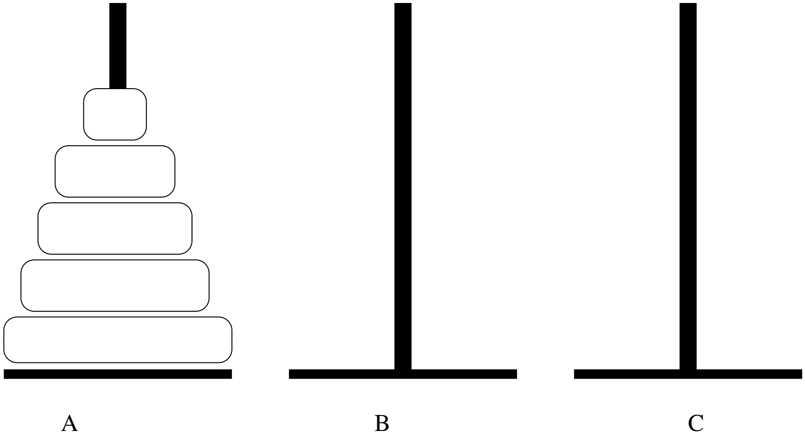
Quantas chamadas recursivas o algoritmo pode fazer?

```
long int pot(long int x, long int n) {
    long int aux;

    if (n == 0)
        return 1;

    else if (n % 2 == 0) {
        aux = pot(x, n/2);
        return aux * aux;
    }

    else {
        aux = pot(x, (n-1)/2);
        return x * aux * aux;
    }
}
```

# Cálculo de Potências

- No algoritmo anterior, a cada chamada recursiva o valor de $n$ é dividido por 2. Ou seja, a cada chamada recursiva, o valor de $n$ decai para pelo menos a metade.
- Usando divisões inteiras faremos no máximo $\lceil(\log_2 n)\rceil + 1$ chamadas recursivas.
- Enquanto isso, o algoritmo iterativo executa o laço $n$ vezes.

# Torres de Hanoi



## Torres de Hanoi

- Inicialmente temos 5 discos de diâmetros diferentes na estaca A.
- O problema das torres de Hanoi consiste em transferir os cinco discos da estaca A para a estaca C (pode-se usar a estaca B como auxiliar).
- Porém deve-se respeitar as seguintes regras:
    - Apenas o disco do topo de uma estaca pode ser movido.
    - Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.

# Torres de Hanoi

- Vamos considerar o problema geral onde há $n$ discos.
- Vamos usar indução para obtermos um algoritmo para este problema.

## Torres de Hanoi

- Base: $n = 1$. Neste caso temos apenas um disco. Basta mover este disco da estaca A para a estaca C.
- Hipótese: Sabemos como resolver o problema quando há $n - 1$ discos.
- Passo: Devemos resolver o problema para $n$ discos.
  - Por hipótese de indução, sabemos mover os $n - 1$ primeiros discos da estaca **A** para **B** usando **C** como auxiliar.
  - Depois de movermos estes $n - 1$ discos, movemos o maior disco (que continua na estaca **A**) para a estaca **C**.
  - Novamente pela hipótese de indução, sabemos mover os $n - 1$ discos da estaca **B** para **C** usando **A** como auxiliar.
- Com isso temos uma solução para o caso onde há $n$ discos.
- A indução nos fornece um algoritmo e ainda por cima temos uma demonstração formal de que ele funciona!

# Torres de Hanoi: algoritmo

Problema: Mover $n$ discos de **A** para **C**.

1. Se $n = 1$, então mova o único disco de **A** para **C** e pare.
2. Caso contrário ($n > 1$) desloque de forma recursiva os $n - 1$ primeiros discos de **A** para **B**, usando **C** como auxiliar.
3. Mova o último disco de **A** para **C**.
4. Mova, de forma recursiva, os $n - 1$ discos de **B** para **C**, usando **A** como auxiliar.

# Torres de Hanoi: algoritmo

A função que computa a solução (em C) terá o seguinte protótipo:

```
void hanoi(int n, char estacaInicio, char estacaFim, char
    estacaAux);
```

É passado como parâmetro o número de discos a ser movido (**n**) e três caracteres indicando de onde os discos serão movidos (**estacaInicio**), para onde devem ser movidos (**estacaFim**) e qual é a estaca auxiliar (**estacaAux**).

# Torres de Hanoi: algoritmo

A função que computa a solução é:

```c
void hanoi(int n, char estacaInicio, char estacaFim, char estacaAux) {
    if (n == 1) {
        /* Caso base: move o único disco diretamente */
        printf("Mova disco %d de %c para %c.\n", n, estacaInicio, estacaFim);
    } else {
        /* Move n-1 discos de Inicio para Aux usando Fim de auxiliar: */
        hanoi(n-1, estacaInicio, estacaAux, estacaFim);

        /* Move o maior disco para estacaFim: */
        printf("Mova disco %d de %c para %c.\n", n, estacaInicio, estacaFim);

        /* Move os n-1 discos de Aux para Fim usando Ini de auxiliar: */
        hanoi(n-1, estacaAux, estacaFim, estacaInicio);
    }
}
```

# Torres de Hanoi: algoritmo

```c
#include <stdio.h>

void hanoi(int n, char estacaInicio, char estacaFim, char estacaAux);

int main() {
    hanoi(4, 'A', 'C', 'B');
    return 0;
}

void hanoi(int n, char estacaInicio, char estacaFim, char estacaAux) {
    if (n == 1)
        printf("Mova disco %d de %c para %c.\n", n, estacaInicio, estacaFim);
    else {
        hanoi(n-1, estacaInicio, estacaAux, estacaFim);
        printf("Mova disco %d de %c para %c.\n", n, estacaInicio, estacaFim);
        hanoi(n-1, estacaAux, estacaFim, estacaInicio);
    }
}
```

# Exercícios

# Exercício

O que será impresso pela chamada `imprimir(5)`?

```c
void imprimir(int i) {
    int j;
    if (i > 0) {
        imprimir(i - 1);
        for (j = 1; j <= i; j++)
            printf("*");
        printf("\n");
    }
}
```

## Exercício

Mostre o estado da pilha de memória durante a execução da função `fibonacci` com a chamada `fibonacci(5)`.

Qual versão é mais eficiente para se calcular o *n*-ésimo número de Fibonacci, a recursiva ou a iterativa? Justifique.

# Exercício

Escreva uma função recursiva que, dado um número inteiro positivo $n$, imprima a representação binária de $n$