

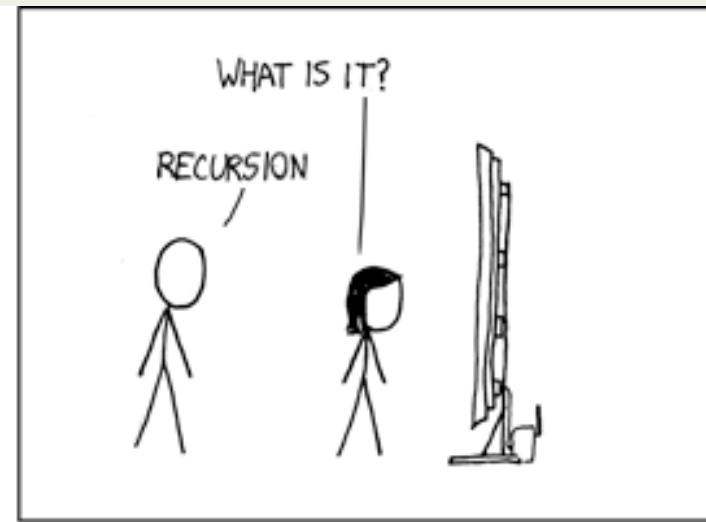
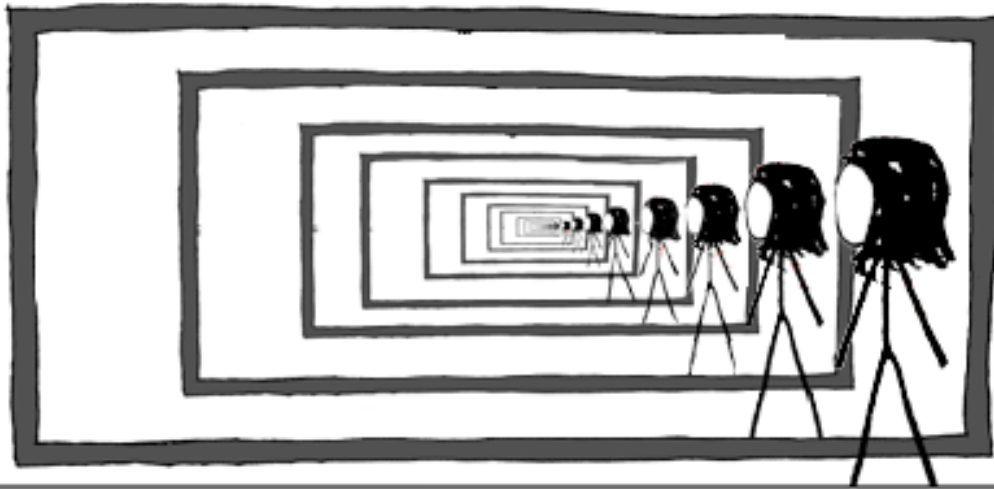
Estrutura de Dados Básicas I.

Revisão Sobre Recursão

Prof. Eiji Adachi M. Barbosa

REVISÃO SOBRE RECURSÃO

Recursão



Fonte: <http://xkcdsw.com/1105>

O Que é Recursão?

**Recursão é um
conceito
independente da
computação**

Natureza



Arte



Linguagem

- Frase:
 - Sujeito + Predicado
 - Sujeito + Verbo + Frase

Matemática

- Fatorial:

- $X! = X * (X-1)! = X * (X-1) * (X-2)! = \dots$

- Exponenciação:

- $X^Y = X * X^{Y-1} = X * X * X^{Y-2} = X * X * X * X^{Y-3} \dots$

Matemática

- Números Naturais: (def. Recursiva)
 - X é um número natural sse.:
 - $X = 0$
 - X é sucessor de 0

E na Computação?

Computação

- Recursão pode ser usada como uma estratégia para resolver problemas dividindo-os em subproblemas da mesma forma
- Em programação, é uma técnica em que uma função chama a si mesma, direta ou indiretamente
 - Poderoso recurso para descrevermos algoritmos concisamente

Recursão

- Podemos projetar uma solução recursiva para um determinado problema se for possível identificar:
 - Um caso simples, para o qual a solução do problema é trivial
 - Uma decomposição do problema que permita quebrá-lo numa versão menor com mesma forma

Formato de Solução Recursiva

- Caso base:
 - Solução trivial para um problema pequeno
 - Garante que a recursão não será infinita
- Caso recursivo:
 - Decomposição do problema maior em subproblemas de mesma forma
 - Deve garantir que os subproblemas “caminhem” em direção ao caso base

Formato de Solução Recursiva

```
foo( Problema de tamanho N )
{
    if( Caso simples identificado )
    {
        Resolva o problema trivialmente,
        sem chamada recursiva
    }
    else
    {
        Quebre o problema em um ou mais subproblemas
        de mesmo formato
        Resolva os subproblemas usando chamadas recursivas
        Combine a solução dos subproblemas para resolver
        o problema original
    }
}
```


Exemplos

- Soma recursiva
- Multiplicação recursiva
- Fatorial

Tipos de Recursão

- Direta:
 - A função chama a si mesma diretamente
- Indireta:
 - A função chama a si mesma através de outras funções
- Simples:
 - A função chama a si mesma uma vez
- Múltipla:
 - A função chama a si mesma múltiplas vezes

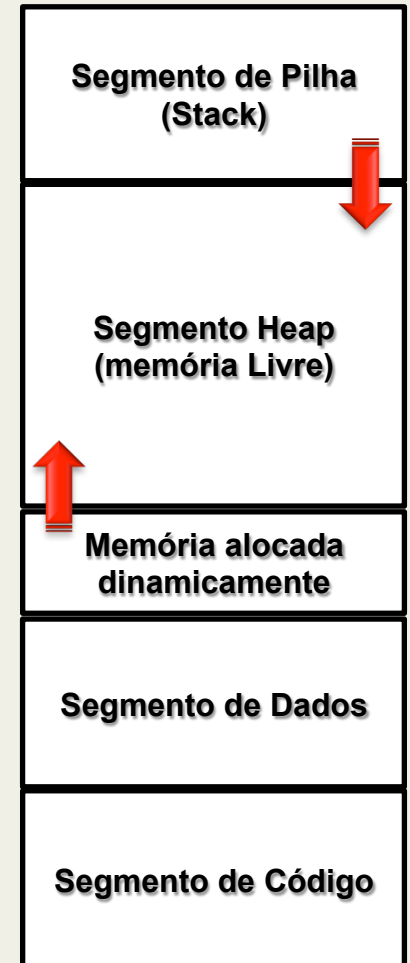
Como a recursão funciona internamente?

Funcionamento de Chamadas Recursivas

- Internamente, chamadas recursivas operam da mesma forma que chamadas não recursivas
 - A cada chamada de função, cria-se um registro independente na pilha de execução do programa
 - Ao término de uma função, o seu registro é retirado da pilha de execução

Memória de Trabalho

- Segmento de pilha (stack):
 - Onde funções alocam temporariamente suas variáveis locais
- Segmento heap:
 - Onde variáveis dinâmicas são alocadas (tempo de execução)
- Segmento de dados:
 - Onde variáveis globais e estáticas são alocadas (tempo de compilação)
- Segmento de código:
 - Onde instruções de máquina do programa são encontradas



Memória de Trabalho

- Quando funções são chamadas dentro de um programa, é criado um Registro de Ativação na Stack
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa que chamou a função
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao ponto de retorno no programa que chamou a função

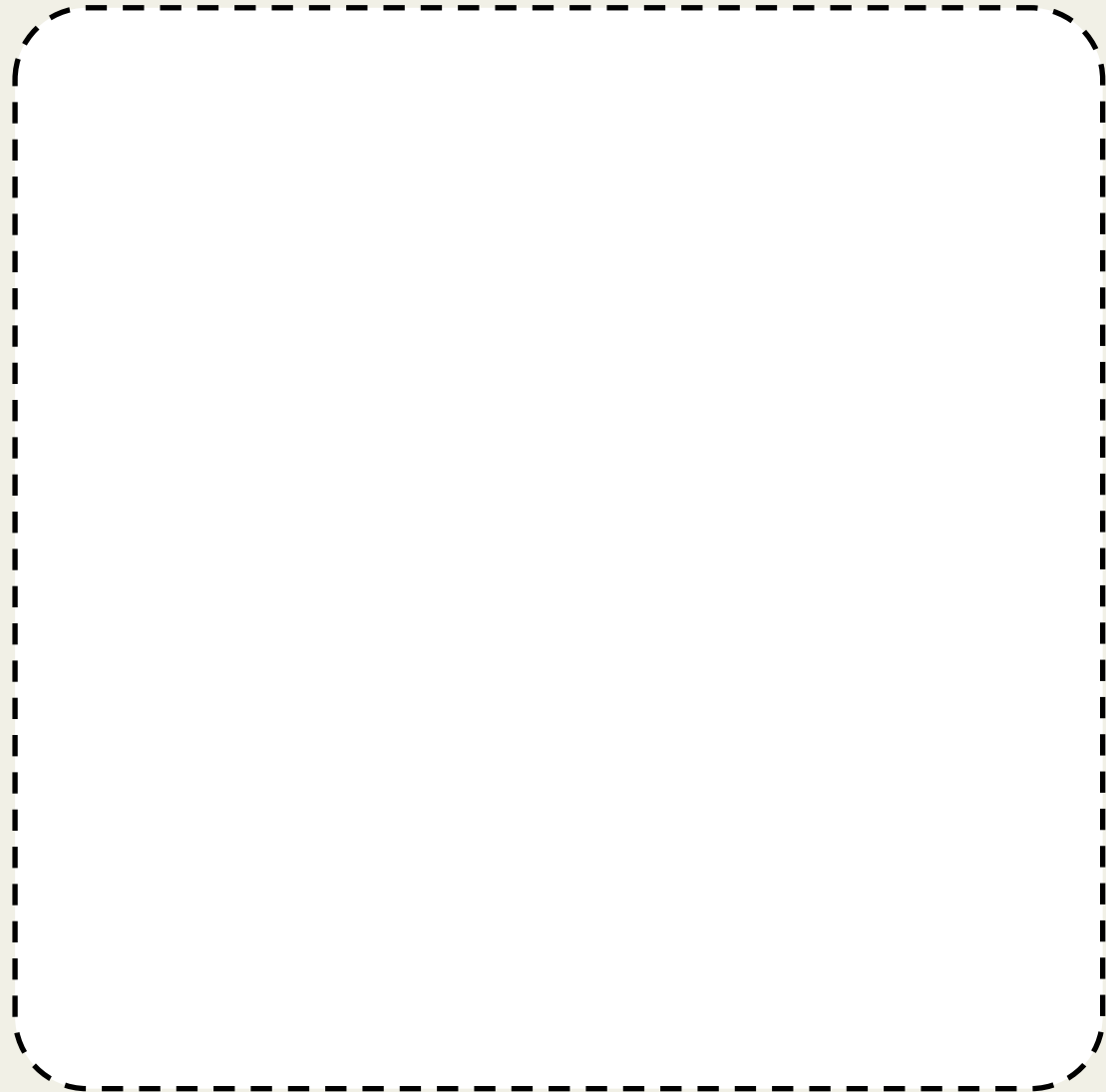
Registro de Ativação

- Guarda o estado de uma função, ou seja:
 - Variáveis locais
 - Valores dos parâmetros
 - Endereço de retorno
 - Valor de retorno

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```



Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

#main, [f = ?]

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = ?, ret = ?]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = ?, ret = ?]
#fact, [a = 0, ret = 1]
```


Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = ?, ret = ?]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = 1, ret = ?]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = ?, ret = ?]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = 1, ret = ?]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = ?, ret = ?]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```


Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = 2, ret = ?]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = ?, ret = ?]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = 6, ret = ?]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = 6, ret = 24]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = ?]
#fact, [a = 4, f = 6, ret = 24]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Stack

```
int main(void){
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int a)
{
    if( a == 0 )
    {
        return 1;
    }
    else
    {
        int f =
            fact( a-1 );
        return a*f;
    }
}
```

```
#main, [f = 24]
#fact, [a = 4, f = 6, ret = 24]
#fact, [a = 3, f = 2, ret = 6]
#fact, [a = 2, f = 1, ret = 2]
#fact, [a = 1, f = 1, ret = 1]
#fact, [a = 0, ret = 1]
```

Problemas Comuns em Soluções Recursivas

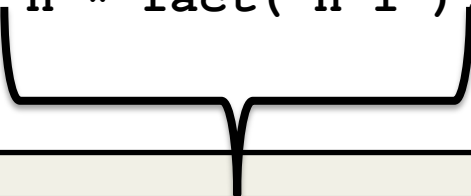
- Ausência de caso base
- Checagem incorreta do caso base
- Solução incorreta no caso base
- Decomposição incorreta do problema
 - Subproblemas não “caminham” em direção ao caso base
 - Subproblemas não tem a mesma forma do problema original
- Construção incorreta da solução a partir das soluções dos subproblemas
- Estouro de pilha (stack overflow)

Recursão de Cauda

- Quando a última instrução de uma função é uma chamada recursiva, esta recursão é chamada de “Recursão de Cauda”
- É útil implementarmos funções recursivas com recursão de cauda
 - Para funções que apresentam recursão de cauda, tipicamente o compilador consegue eliminar a recursão, gerando, em código de máquina, uma versão mais eficiente do que a recursiva

Recursão de Cauda

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



Depois do retorno da função,
ainda há uma multiplicação a ser feita

Recursão de Cauda

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

```
int fact(int n){
    return t_fact(n, 1);
}

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

Recursão de Cauda

```
int fact(int n){  
    return t_fact(n, 1);  
}  
  
int t_fact(int n, int a){  
    if( n == 0 ){  
        return a;  
    }  
    else{  
        return t_fact( n-1, a*n );  
    }  
}
```

Criamos um parâmetro extra para “guardarmos” o valor que antes era calculado após o retorno da função.

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

#main, [f=?]

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
```


Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=?]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=?]
#tail_fact, [n=0, a=24, ret=?]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=?]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=?]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=?]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```


Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=?]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=?]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=24]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=?]
#tail_fact, [n=4, a=1, ret=24]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=24]
#tail_fact, [n=4, a=1, ret=24]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=?]
#fact, [n=4, ret=24]
#tail_fact, [n=4, a=1, ret=24]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```

Stack

```
int main(void)
{
    int f = fact(4);
    printf("%d\n", f);
    return 0;
}

int fact(int n){
    return t_fact(n, 1);
}

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

```
#main, [f=24]
#fact, [n=4, ret=24]
#tail_fact, [n=4, a=1, ret=24]
#tail_fact, [n=3, a=4, ret=24]
#tail_fact, [n=2, a=12, ret=24]
#tail_fact, [n=1, a=24, ret=24]
#tail_fact, [n=0, a=24, ret=24]
```


Recursão e Iteração

- Algoritmos recursivos que apresentam recursão em cauda, possuem uma solução iterativa (não recursiva) equivalente mais eficiente
- Recursão e iteração possuem o mesmo poder de expressividade
- É possível traduzir a recursão para a forma iterativa
 - Eliminando a recursão de cauda
 - Manipulando com índices
 - Usando uma estrutura auxiliar para "simular" a stack (pilha)

De Recursão para Iteração

- Forma geral:

```
int funcao( param )
{
    if( condicao )
        return caso_base( param );
    operacao1;
    operacao2;
    operacao3;
    ajusta( parametros );
    return funcao( calc(param) );
}
```



```
int funcao( param )
{
    int result = caso_base( param );
    while( !condicao ) {
        operacao1;
        operacao2;
        operacao3;
        ajusta( param );
        result = calc( param );
    }
    return result;
}
```

De Recursão para Iteração

- Exemplo:

```
int fact(int n){
    return t_fact(n, 1);
}

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

Quando Usar Recursão?

Expressividade

X

Eficiência

Quando Usar Recursão?

- Vale a pena para algoritmos cujas versões iterativas são mais complexas e requerem o uso explícito de uma pilha
 - Dividir para Conquistar (Ex. MergeSort, Quicksort)
 - Caminhamento em Árvores (pesquisa, backtracking)
- É sempre importante avaliar as complexidades de tempo e espaço (devido a pilha de execução) do algoritmo recursivo
 - Com isso, nota-se que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

Exercícios

- Termine os exercícios da aula passada
- Para as funções add, multiply e factorial, produza:
 - Versões recursivas com recursão em cauda
 - Versões iterativas transformadas a partir da versão recursiva com recursão em cauda
- Implemente programas e respectivos testes para as questões na lista de exercícios

Estrutura de Dados Básicas I.

Revisão Sobre Recursão

Prof. Eiji Adachi M. Barbosa