DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

# Estrutura de Dados

## Análise de Algoritmos Recursivos

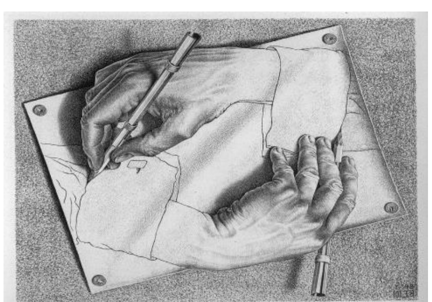
Professores: Luiz Chaimowicz e Raquel Prates

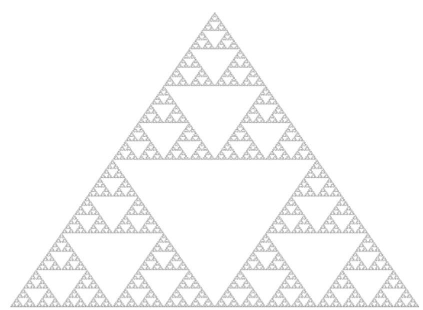
Revisão de

# ALGORITMOS RECURSIVOS

# Recursividade

- **Definição:** Um procedimento que chama a si mesmo é dito ser **recursivo**.
- **Vantagem:** Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.





# Recursividade

- Fatorial:

  n! = n*(n-1)!  para n>0

  0! = 1

- Em C

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

# Estrutura de uma função recursiva

- Normalmente, as funções recursivas são divididas em duas partes
  - Chamada Recursiva
  - Condição de Parada

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

# Estrutura de uma função recursiva

- A condição de parada é fundamental para evitar a execução de loops infinitos

- A chamada recursiva pode ser
  - Direta: função A chama ele mesma
  - Indireta: A chama B que chama A novamente

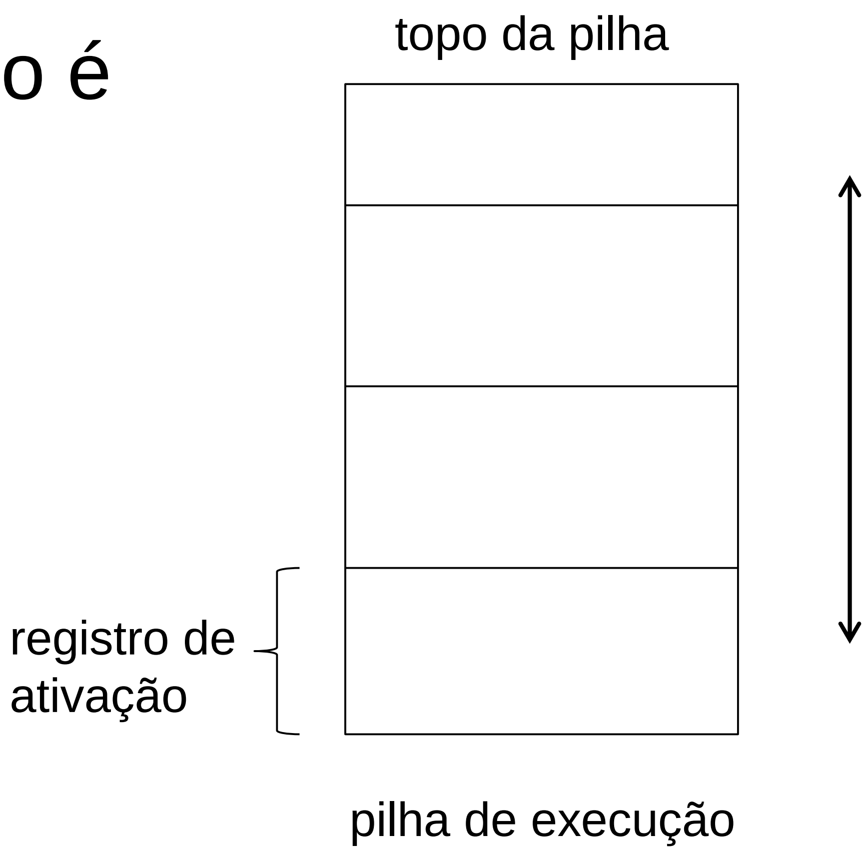
# Execução

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação** na **Pilha de Execução** do programa

- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função.

# Execução

- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função



# Exemplo de execução

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


main() {
   int f;
   f = fat(4);
   printf("%d",f);
}
```

| | |
|---|---|
| fat(0) | 1 |
| fat(1) | 1 |
| fat(2) | 2 |
| fat(3) | 6 |
| fat(4) | 24 |

pilha de execução

Voltando para

# ANÁLISE DE COMPLEXIDADE

# Função Fatorial Não Recursiva

- Qual a ordem de complexidade?

```
int fat (int n) {
   int f;
   f = 1;
   while(n > 0){
      f = f * n;
      n = n - 1;
   }
   return f;
}
```

# Análise de Complexidade

- E para a função recursiva?

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

# Análise de Algoritmos Recursivos

- Para cada procedimento recursivo é associada uma função de complexidade *f(n)* desconhecida, onde n mede o tamanho dos argumentos para o procedimento.
- Por se tratar de um algoritmo recursivo, *f(n)* vai ser obtida através de uma equação de recorrência.

# CONTINUAÇÃO

# Análise do Tempo de Execução

- **Equação de recorrência**: maneira de definir uma função por uma expressão envolvendo a mesma função (com entradas menores).
- Exemplo:

$$\begin{cases} T(n) = aT(n/b) + f(n), \text{para } n > c \\ T(n) = k, \text{para } n \leq c \end{cases}$$

# Análise de Algoritmos Recursivos

- Voltando à função fat

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

- Equação de recorrência para o Fatorial (número de multiplicações):

$$\boxed{\begin{array}{ll} T(n) = 1 + T(n-1), & para\ n > 0 \\ T(n) = 0 & para\ n \leq 0 \end{array}}$$

# Análise de Algoritmos Recursivos

- E para a função max? (Comparações de elementos)

```
int max(int *A, int e, int d){
    int u, v;
    int m = (e+d)/2;
    if (e == d) return A[e];
    u = max(A, e, m);
    v = max(A, m+1, d);

    if (u > v)
        return u;
    else
        return v;
}
```

$$T(n) = 2T(n/2) + 1, \quad \text{para } n > 1$$
$$T(n) = 0. \quad \text{para } n \leq 1$$

# Exercício

- Determine a equação de recorrência para a função abaixo (operação relevante: atribuição para vetor X).

```
void ex(int *X, int n){
    int i;

    if (n > 0) {
        for(i=0; i<n-1; i++)
            X[i] = 0;
        X[n] = n;
        ex(X,n-1);
    }
}
```

$$T(n) = n + T(n\text{-}1), \quad para\ n > 0$$
$$T(n) = 0. \quad para\ n \leq 0$$

# Exercício

- Considere a seguinte função (operação relevante “inspecione item”):

```
Pesquisa(n) {
  (1)   if (n <= 1)
  (2)       ‘ inspecione item’ e termine;
        else  {
  (3)       para cada um dos n elementos ‘ inspecione item’;
  (4)       Pesquisa(n/3) ;
        }
}
```

$$\begin{cases} T(n) = n + T(n/3), \text{ para } n > 1 \\ T(n) = 1, \text{ para } n \leq 1 \end{cases}$$

# Qual é a ordem de complexidade?

- Voltando à função fat

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

- Equação de recorrência para o Fatorial (Custo geral da função):

$$T(n) = c + T(n\text{-}1),\ \ para\ n > 0$$
$$T(n) = d \qquad\qquad\qquad para\ n \leq 0$$

# Análise de Algoritmos Recursivos

- Precisamos resolver a equação de recorrência
- Vários métodos de resolução:
  - **Expansão de termos / Árvore de Recursão**
  - **Teorema Mestre**
  - Método da Substituição

# Análise de Algoritmos Recursivos

- Expansão de termos / Árvore de Recursão
  - Expanda a árvore de recursão
  - Calcule o custo em cada nível da árvore
  - Some os custos de todos os níveis
  - Calcule a fórmula fechada do somatório

# Análise de Algoritmos Recursivos

- De volta ao fatorial
  - Se n <= 0, temos uma operação de custo constante
  - Se n > 0, temos uma operação de custo constante e a chama chamada recursiva da função

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

$$T(n) = c + T(n\text{-}1), \; para \; n > 0$$
$$T(n) = d \qquad\qquad para \; n \leq 0$$

# Análise de Algoritmos Recursivos

- Expandindo a equação e depois fazendo uma substituição dos termos:

$$T(n) = c + T(n\text{-}1)$$
$$T(n\text{-}1) = c + T(n\text{-}2)$$
$$T(n\text{-}2) = c + T(n\text{-}3)$$
$$\ldots$$
$$T(1) = c + T(0)$$
$$T(0) = d$$

$$T(n) = \underbrace{c + c + c + \ldots + c}_{n \text{ vezes}} + d$$

$$T(n) = n.c + d \quad \rightarrow \quad O(n)$$

# Análise de Algoritmos Recursivos

- Então a ordem complexidade do algoritmo para calcular o fatorial de maneira recursiva é O(n)
- E quanto à versão não recursiva?

```
int fat (int n) {
   int f;
   f = 1;
   while(n > 0){
     f = f * n;
     n = n - 1;
   }
   return f;
}
```

Também é O(n)

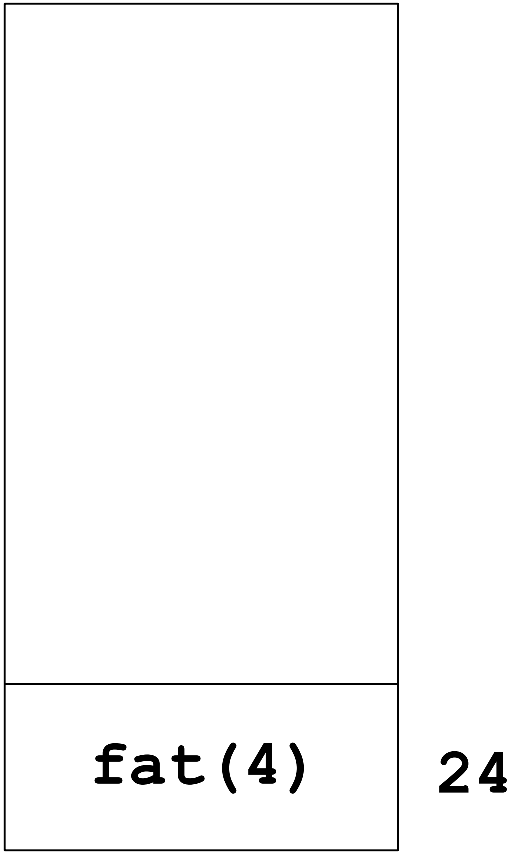
# Análise de Algoritmos Recursivos

- Qual é a melhor alternativa? O código recursivo ou o código iterativo?
- Vamos olhar o que acontece com a complexidade de espaço...

# Exemplo de execução

## Versão Iterativa:

```
void fat (int n) {
   int f = 1
   while(n > 0){
      f = f * n;
      n = n - 1;
    }
    return f;
}
main() {
   int f;
   f = fat(4);
   printf("%d",f);
}
```




pilha de execução

# Exemplo de execução

## Versão Recursiva:

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


main() {
  int f;
  f = fat(4);
  printf("%d",f);
}
```

| | |
|---|---|
| fat(0) | 1 |
| fat(1) | 1 |
| fat(2) | 2 |
| fat(3) | 6 |
| fat(4) | 24 |

pilha de execução

# Análise de Algoritmos Recursivos

- Para a abordagem recursiva complexidade de espaço é O(n), devido a pilha de execução
- Já na abordagem iterativa complexidade de espaço é O(1)
- Novamente, vemos que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

# Exercício

- Determine a equação de recorrência para a função abaixo (operação relevante: atribuição para vetor X). Qual sua complexidade?

```
void ex(int *X, int n){
    int i;

    if (n > 0) {
        for(i=0; i<n-1; i++)
            X[i] = 0;
        X[n] = n;
        ex(X,n-1);
    }
}
```

$$T(n) = n + T(n\text{-}1), \quad para\ n > 0$$
$$T(n) = 0. \quad para\ n \leq 0$$

# Análise da Complexidade

$$T(n) = n + T(n\text{-}1), \qquad para\ n > 0$$
$$T(n) = 0. \qquad para\ n \leq 0$$

Expandindo a equação e depois fazendo uma substituição dos termos:

$$T(n) = n + T(n\text{-}1)$$
$$T(n\text{-}1) = (n\text{-}1) + T(n\text{-}2)$$
$$T(n\text{-}2) = (n\text{-}2) + T(n\text{-}3)$$
$$\ldots$$
$$T(1) = 1 + T(0)$$
$$T(0) = 0$$

$$T(n) = n + \underbrace{(n\text{-}1) + (n\text{-}2) + \ldots + 2 + 1}$$

$$\frac{(1+n).n}{2}$$

$$O(n^2)$$

# Análise de Algoritmos Recursivos

- E para a função max? (Comparações de elementos)

```
int max(int *A, int e, int d){
    int u, v;
    int m = (e+d)/2;
    if (e == d) return A[e];
    u = max(A, e, m);
    v = max(A, m+1, d);

    if (u > v)
        return u;
    else
        return v;
}
```

$$T(n) = 2T(n/2) + 1, \quad para\ n > 1$$
$$T(n) = 0. \quad para\ n \leq 1$$

$$T(n) = 2T(n/2) + 1$$
$$T(1) = 0$$
$$\text{Expandindo a equação:}$$
$$T(n) = 2T(n/2) + 1$$
$$2T(n/2) = 4T(n/4) + 2$$
$$4T(n/4) = 8T(n/8) + 4$$
$$\ldots$$
$$2^{i-1}T(n/2^{i-1}) = 2^{i}T(n/2^{i}) + 2^{i-1}$$
$$\text{Substituindo os termos:}$$
$$T(n) = 2^{i}T(n/2^{i}) + 1 + 2 + 4 + \ldots + 2^{i-1}$$
$$\text{Para colocar a Condição de Parada:}$$
$$T(n/2^{i}) \rightarrow T(1)$$
$$n/2^{i} = 1 \rightarrow i = \log_2 n$$
$$\text{Logo:}$$
$$T(n) = 2^{i}T(n/2^{i}) + \sum_{k=0}^{\log_2 n - 1} 2^{k}$$
$$T(n) = 0 + \frac{1 - 2^{\log_2 n}}{1 - 2} = n - 1$$

Somatório de uma PG finita:

$$\sum_{i=0}^{n} r^{i} = \frac{1 - r^{n+1}}{1 - r}$$

# Exercício

- Seja a equação:

$$T(n) = 2T(n/2) + n;$$
$$T(1) = 1;$$

- Essa equação lembra que tipo de problema?
- Como resolver essa equação?

# Exercício

$$T(n) = 2T(n/2) + n$$

$$T(1) = 1$$

Expandindo a equação:

$$T(n) = 2T(n/2) + n$$

$$2T(n/2) = 4T(n/4) + n$$

$$4T(n/4) = 8T(n/8) + n$$

$$\vdots$$

$$2^{i-1}T(n/2^{i-1}) = 2^{i}T(n/2^{i}) + n$$

Substituindo os termos:

$$T(n) = 2^{i}T(n/2^{i}) + i.n$$

Para colocar a Condição de Parada:

$$T(n/2^{i}) \rightarrow T(1)$$

$$n/2^{i} = 1 \rightarrow i = \log_2 n$$

Logo:

$$T(n) = 2^{i}T(n/2^{i}) + i.n = 2^{\log_2 n}.1 + (\log_2 n).n = n + n.\log_2 n = O(n.\log_2 n)$$

# Teorema Mestre

- Método "receita de bolo" para resolver recorrências do tipo

$$T(n) = aT(n/b) + f(n)$$

$$\text{onde } a \geq 1, b > 1 \text{ e } f(n) \text{ positiva}$$

- Três casos possíveis.

# Teorema Mestre

$$T(n) = aT(n/b) + f(n)$$

$$\text{onde } a \geq 1, b > 1 \text{ e } f(n) \text{ positiva}$$

- Este tipo de recorrência é típico de algoritmos "dividir para conquistar"
  - Dividem um problema em $\boldsymbol{a}$ subproblemas
  - Cada subproblema tem tamanho $\boldsymbol{n/b}$
  - Custo para dividir e combinar os resultados é $\mathbf{f(n)}$

Exemplo – para o algoritmo de Max, temos a equação de recorrência

$$\begin{cases} T(n) = 2T(n/2) + 1, & para\ n > 1 \\ T(n) = 1. & para\ n \leq 1 \end{cases}$$

$a = 2$, $b = 2$ e $f(n) = 1$

# Teorema Mestre

**Teorema Mestre:** Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função assintoticamente positiva e $T(n)$ uma medida de complexidade definida sobre os inteiros. A solução da equação de recorrência:

$$T(n) = aT(n/b) + f(n),$$

para $b$ uma potência de $n$ é:

1. $T(n) = \Theta(n^{\log_b a})$, se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$,
2. $T(n) = \Theta(n^{\log_b a} \log n)$, se $f(n) = \Theta(n^{\log_b a})$,
3. $T(n) = \Theta(f(n))$, se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo $n$ a partir de um valor suficientemente grande.

# Teorema Mestre

**Intuição:** a função ***f(n)*** é comparada com $\boldsymbol{n^{\log_b a}}$ e a maior das duas funções é a solução da recorrência. No caso das duas funções serem equivalentes, a solução é $n^{\log_b a}$ multiplicada por um fator logarítmico

**Detalhes:**

- caso 1: a função *f(n)* deve ser polinomialmente menor do que $\mathbf{n^{\log_b a}}$ .
- caso 3: a função *f(n)* deve ser polinomialmente maior do que $\mathbf{n^{\log_b a}}$ .

Além disso, a função deve satisfazer uma condição de regularidade.

# Teorema Mestre - Exemplos

$$T(n) = 9T(n/3) + n$$

Onde, $a = 9$, $b = 3$, $f(n) = n$ e $n^{\log_b a} = n^{\log_3 9} = n^2$

Caso 1:

$T(n) = \Theta(n^{\log_b a})$, se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$,

O caso 1 se aplica porque $f(n) = O(n^{\log_b a - \epsilon}) = O(n)$, onde $\epsilon = 1$, e a solução é $T(n) = \Theta(n^2)$.



DCC

# Teorema Mestre - Exemplos

$$T(n) = 3T(n/4) + n\log n$$

Onde, $a = 3$ , $b = 4$ , $f(n) = n \log n$ e $n^{\log_b a} = n^{\log_3 4} = n^{0.793}$

Caso 3:

$T(n) = \Theta(f(n))$, se $f(n) = \Omega(n^{\log_b a+\epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo $n$ a partir de um valor suficientemente grande.

O caso 3 se aplica porque $f(n) = \Omega(n^{\log_3 4+\epsilon})$, onde $\epsilon \approx 0.207$ e $af(n/b) = 3(n/4)\log(n/4) \leq cf(n) = (3/4)n\log n$, para $c = 3/4$ e $n$ suficientemente grande. E a solução é $T(n) = \Theta(n \log n)$

# Teorema Mestre - Exemplos

$$T(n) = 2T(n/2) + n - 1$$

Onde, $a = 2$, $b = 2$, $f(n) = n - 1$ e $n^{\log_b a} = n^{\log_2 2} = n$

Caso 2:

$T(n) = \Theta(n^{\log_b a} \log n)$, se $f(n) = \Theta(n^{\log_b a})$

O caso 2 se aplica porque $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$, e a solução é $T(n) = \Theta(n \log n)$.

# Teorema Mestre - Exemplos

$$T(n) = 3T(n/3) + n \log n$$

onde $a = 3$, $b = 3$, $f(n) = n \log n$ e $n^{\log_b a} = n^{\log_3 3} = n$.

O caso 3 não se aplica porque, embora $f(n) = n \log n$ seja assintoticamente maior do que $n^{\log_b a} = n$, a função $f(n)$ não é polinomialmente maior: a razão $f(n)/n^{\log_b a} = (n \log n)/n = \log n$ é assintoticamente menor do que $n^\epsilon$ para qualquer constante $\epsilon$ positiva.

# O que faz a função X?
# Qual a sua complexidade?

```
int X(int *list, int lo, int hi, int k)
{
    int mid;

    if (lo > hi) return -1;
    mid = (lo + hi) / 2;
    if (list[mid] == k) return mid;
    else if (list[mid] > k)
        X(list, lo, mid - 1, k);
    else if (list[mid] < k)
        X(list, mid + 1, hi, k);

}
```