

Estruturas de Dados

Análise de Algoritmos Recursivos

Professores: Luiz Chaimowicz e Raquel Prates

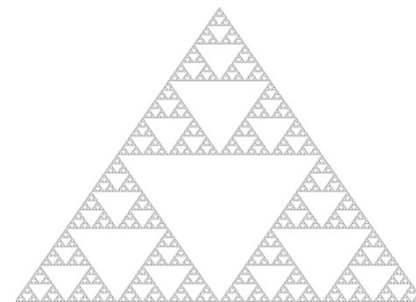
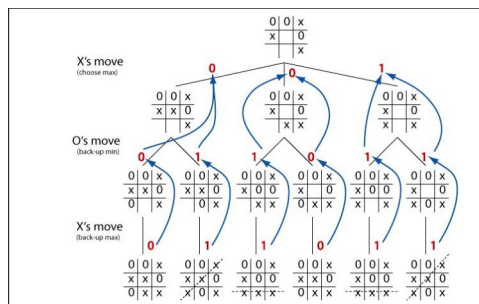
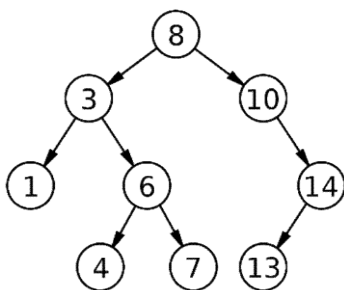
Análise de Algoritmos Recursivos

■ Sumário

- ❑ Revisão de Algoritmos Recursivos
 - Definição
 - Pilha de Execução
 - Exemplos
- ❑ Equações de Recorrência
 - Como representar o custo do algoritmo
- ❑ Resolução de Equações de Recorrência
 - Expansão de Termos (árvore de recursão)
 - Teorema Mestre

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:

$n! = n * (n-1)! \text{ para } n > 0$

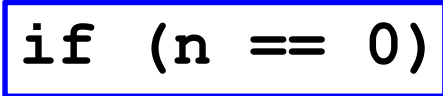

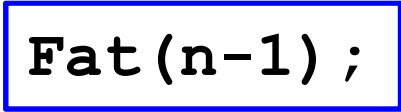

$0! = 1$

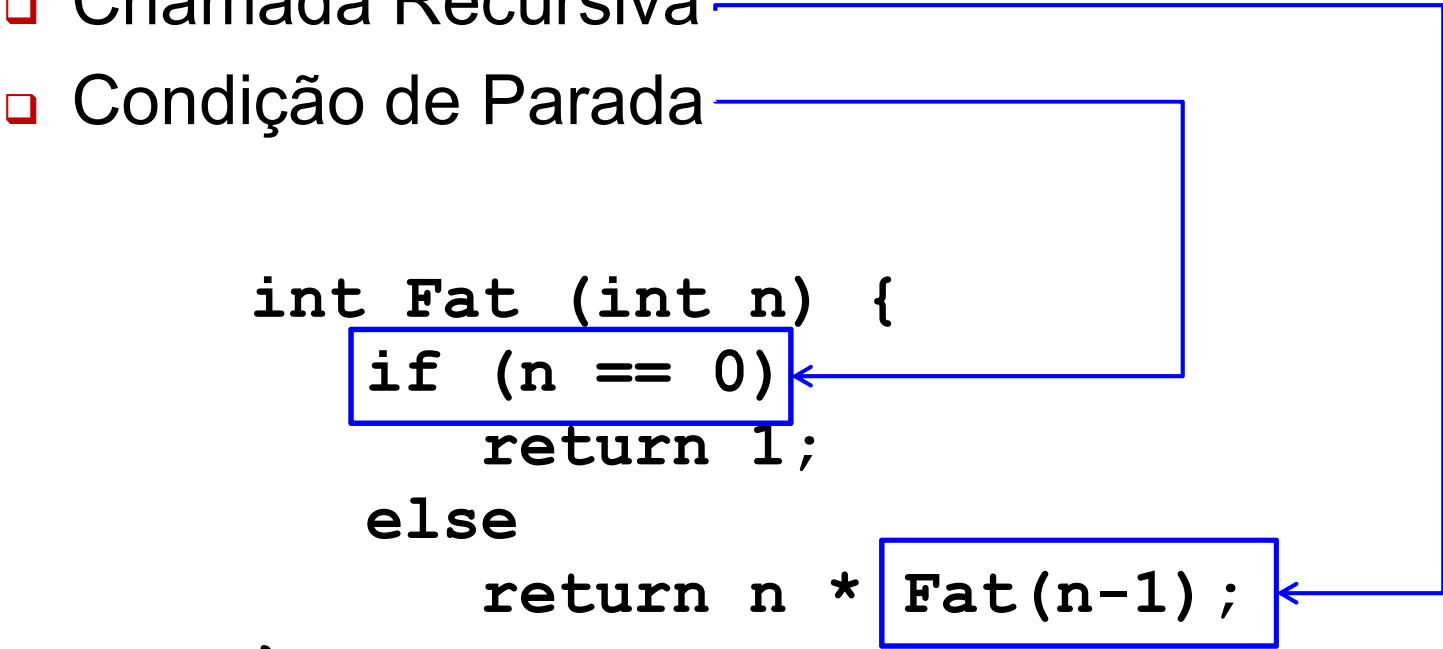
- Em C

```
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 *    
}
```

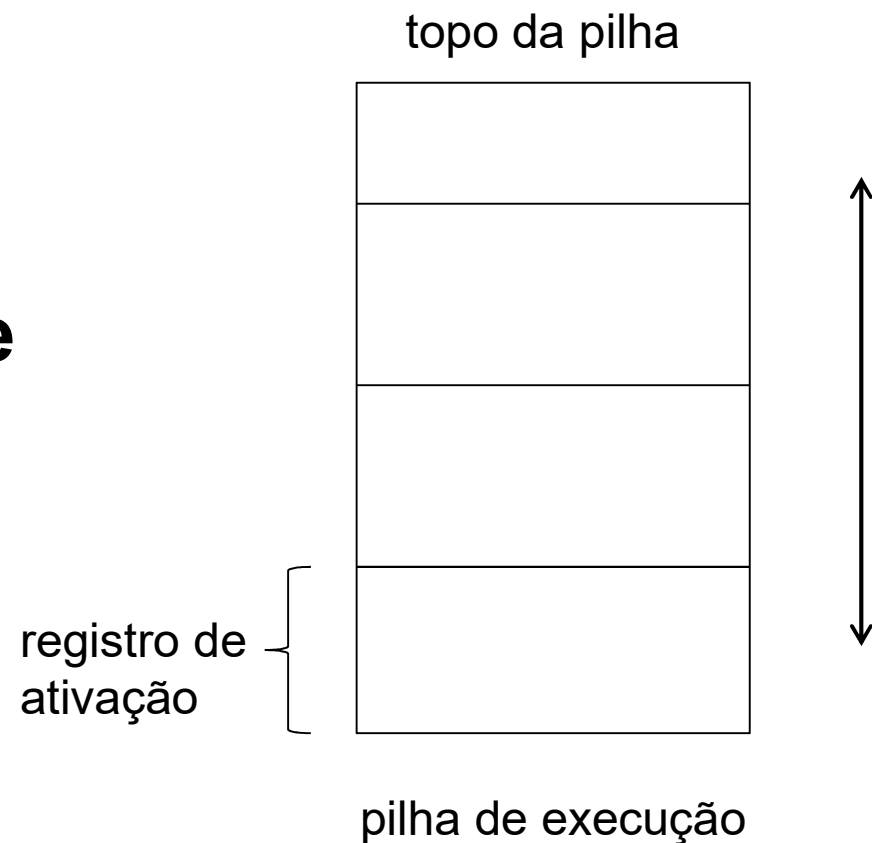


Estrutura de uma função recursiva

- 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
- A chamada recursiva pode ocorrer mais de uma vez dentro da função

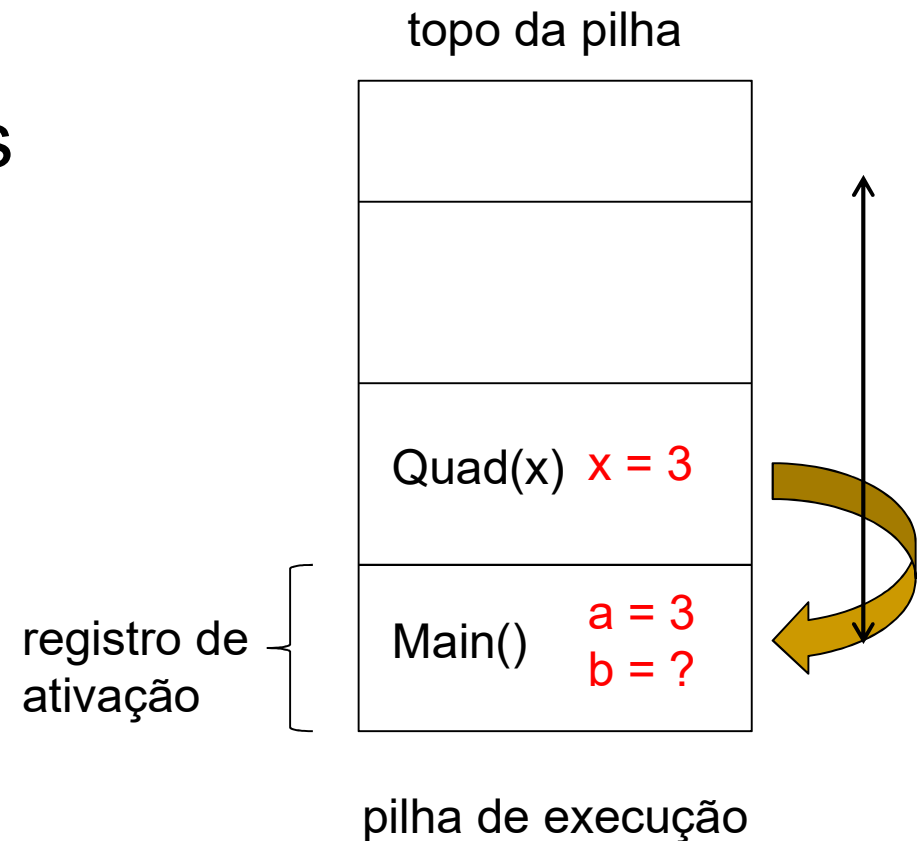
Execução de Algoritmos Recursivos

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



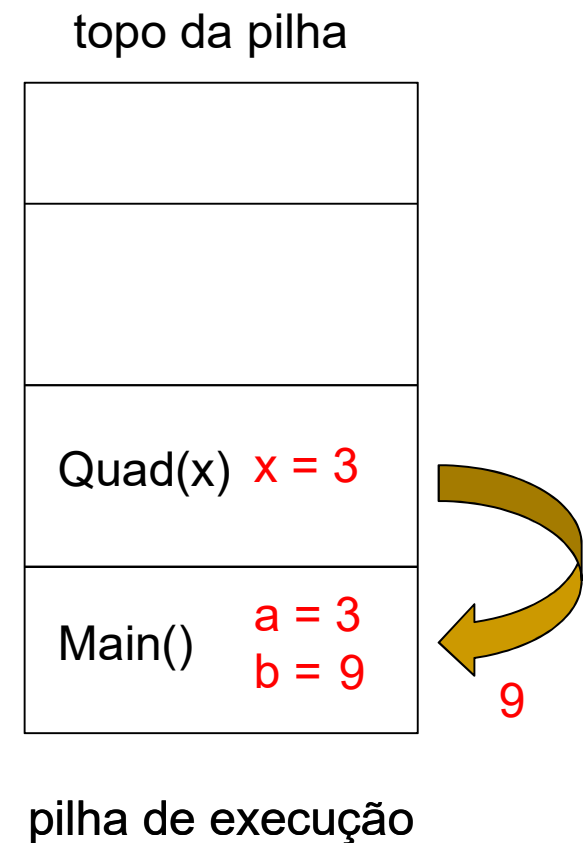
Execução de Algoritmos Recursivos

- 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 de Algoritmos Recursivos

- 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);  
}
```

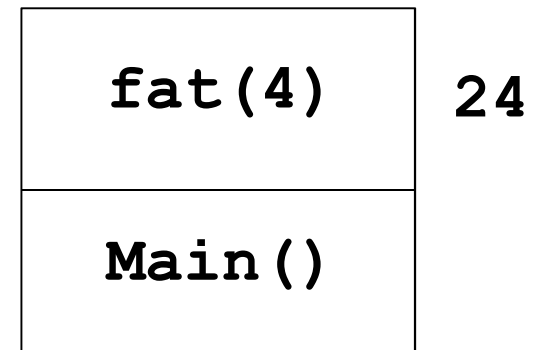
fat(0)	1
fat(1)	1
fat(2)	2
fat(3)	6
fat(4)	24
Main()	

pilha de execução

Função Fatorial Não Recursiva

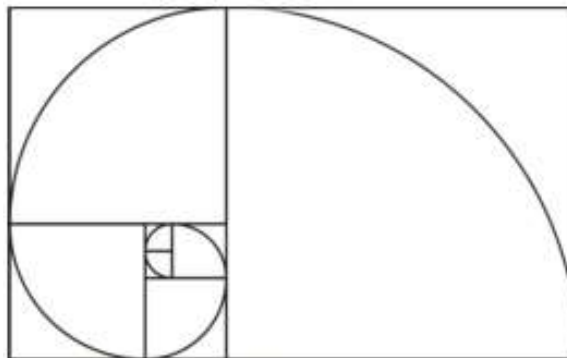
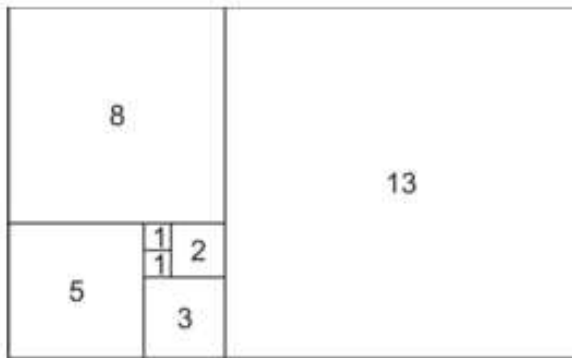
```
int fat (int n) {  
    int f;  
    f = 1;  
    while(n > 0) {  
        f = f * n;  
        n = n - 1;  
    }  
    return f;  
}
```

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



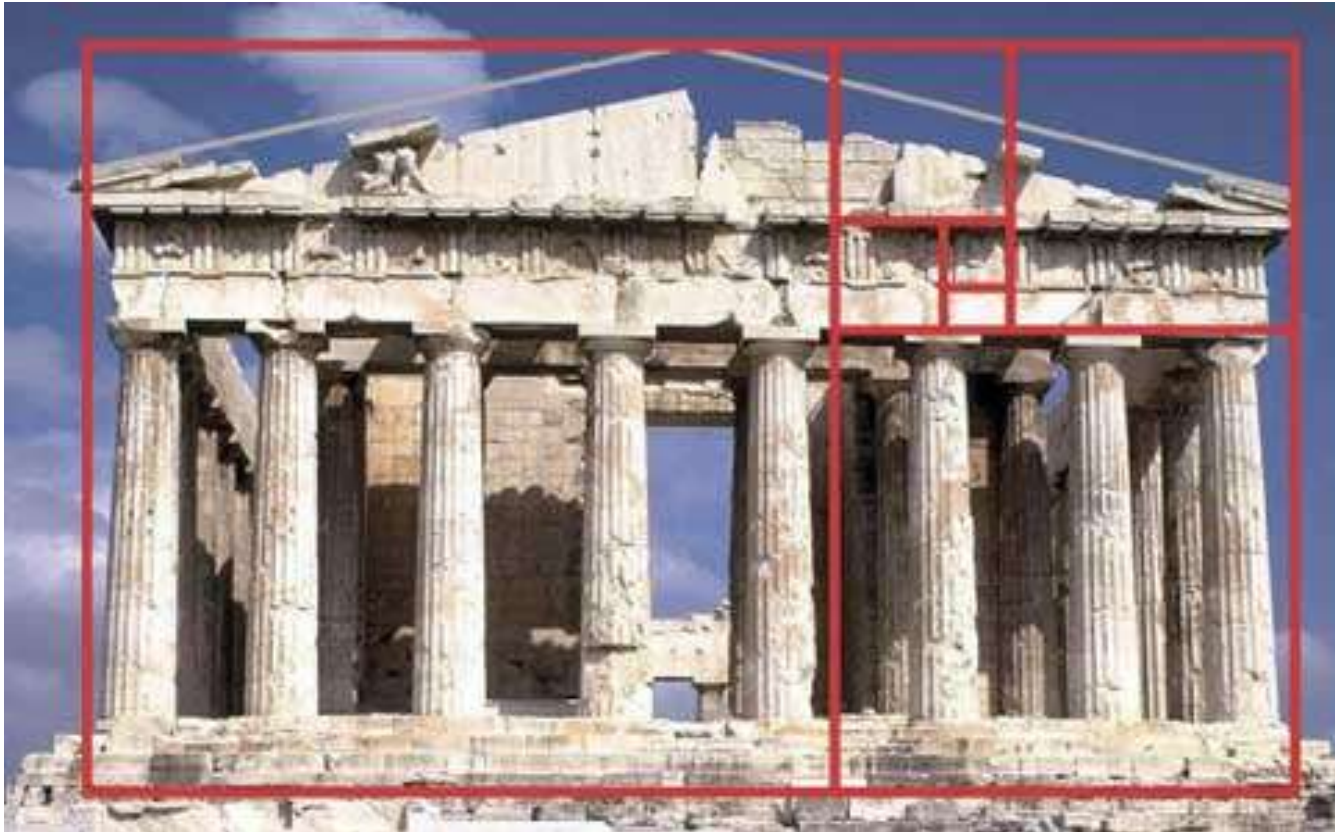
Série de Fibonacci

- $F(n) = F(n-1) + F(n-2) \quad n > 2,$
- $F(1) = F(2) = 1$
 - ❑ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...
 - ❑ A razão do item atual e o anterior é aproximadamente 1,618 (*golden ratio* ϕ)



Série de Fibonacci

- Partenon, largura/altura = $\phi = 1,6180339...$



Série de Fibonacci

■ Série de Fibonacci:

- ❑ $F(n) = F(n-1) + F(n-2) \quad n > 2,$
- ❑ $F(1) = F(2) = 1$

```
int Fib(int n) {  
    if (n < 3)  
        return 1;  
    else  
        return Fib(n-1) + Fib(n-2);  
}
```

Análise da função Fibonacci

- Ineficiência em Fibonacci
- Termos $F(n-1)$ e $F(n-2)$ são computados independentemente
 - Custo para cálculo de $F(n)$
 - $O(\phi^n)$ onde $\phi = (1 + \sqrt{5})/2 = 1,61803\dots$
 - *Golden ratio, Número de ouro, Phi*
- O custo é exponencial!

Fibonacci não recursivo

- Complexidade: $O(n)$
- **Conclusão: não usar recursividade cegamente!**

```
int FibIter(int n) {  
    int fn1 = 1, fn2 = 1;  
    int fn, i;  
  
    if (n < 3) return 1;  
  
    for (i = 3; i <= n; i++) {  
        fn = fn2 + fn1;  
        fn2 = fn1;  
        fn1 = fn;  
    }  
    return fn;  
}
```

<i>n</i>	20	30	50	100
<i>Recursiva</i>	1 seg	2 min	21 dias	10^9 anos
<i>Iterativa</i>	1/3 mseg	1/2 mseg	3/4 mseg	1,5 mseg

Quando vale a pena usar recursividade?

- Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
 - ❑ Dividir para Conquistar (Ex. Quicksort)
 - ❑ Caminhamento em Árvores (pesquisa)

Voltando para

ANÁLISE DE COMPLEXIDADE dos Algoritmos Recursivos

Função Fatorial Não Recursiva

■ Qual a ordem de complexidade?

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

Complexidade de Tempo: $O(1) + O(n) + O(1) = \mathbf{O(n)}$

Complexidade de Espaço: $O(1)$

Análise de Complexidade

- E para a função recursiva?

```
int fat (int n) {  
    if (n <= 0)  —————→ O(1)  
        return 1;      —————→ O(1)  
    else  
        return n * fat(n-1) ;→ O(1) + ?  
}
```

Problema: a análise de complexidade da função `fat` com o parâmetro n , depende da complexidade da própria função com o parâmetro $n-1$

Análise de Algoritmos Recursivos

- Na análise de complexidade, para cada procedimento é associada uma função de complexidade $f(n)$ desconhecida, onde n normalmente é relacionado com o tamanho da entrada do procedimento.
- Por se tratar de um algoritmo recursivo, $f(n)$ vai ser obtida através de uma equação de recorrência.

Análise de Algoritmos Recursivos

- **Equação de recorrência:** maneira de definir uma função por uma expressão envolvendo a mesma função com entradas menores.
- Uma equação de recorrência tem 2 partes
 - Caso base: no qual a equação tem uma solução para um determinado valor de entrada
 - Recorrência: no qual a solução da equação para uma entrada n é expressa em função da solução para valores menores

Análise de Algoritmos Recursivos

■ Exemplos:

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

$$T(1) = 1$$

$$T(2) = T(1) + 2 = 3$$

$$T(3) = T(2) + 3 = 6$$

$$T(4) = T(3) + 4 = 10$$

...

Análise de Algoritmos Recursivos

- Vamos utilizar a equação de recorrência para estimar o **custo** de se resolver um problema de tamanho n em função do custo de se resolver um problema menor

- Voltando à função fat:

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

- **Equação de recorrência:**
(considerando como custo o número de multiplicações):

$$\begin{aligned} T(n) &= 1 + T(n-1), \text{ para } n > 0 \\ T(n) &= 0 \quad \text{para } n \leq 0 \end{aligned}$$

Análise de Algoritmos Recursivos

■ E para Fibonacci?

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

$$T(n) = T(n-1) + T(n-2) + c, \text{ para } n > 2$$

$$T(n) = d \qquad \text{para } n \leq 2$$

onde c e d são constantes

Exemplo / Exercício

- Para a função Pesquisa mostrada abaixo, determine a sua equação de recorrência. (operação relevante “inspecione item”):

```
Pesquisa(n)
  Se (n <= 1)
    'inspecione item' e termine;
  Senão {
    para cada um dos n elementos 'inspecione item';
    Pesquisa(n/3) ;
  }
}
```

$$T(n) = 1, n \leq 1$$
$$T(n) = n + T(n/3)$$

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

Exemplo / Exercício

- O que faz a função abaixo?
- Qual a sua equação de recorrência? (considere como operação relevante o # de comparações de elementos)

```
void SRec(int *A, int n){ // A[0..n-1] é um vetor de int
    int i, imax, aux;
    if (n > 1) {
        imax = n-1;
        for(i=0; i<n-1; i++)
            if(A[i] > A[imax])
                imax = i;
        aux = A[imax];
        A[imax] = A[n-1];
        A[n-1] = aux;
        SRec(A, n-1);
    }
}
```

$$O(n) = (n-1)!$$

$$T(n) = 0, n \leq 1$$

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

Ordena o Vetor usando o algoritmo “Seleção” recursivo

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

Exemplo / Exercício

- O que faz a função abaixo?
- Qual a sua equação de recorrência? (considere como operação relevante a comparação “**if (u > v)**”)

```
int X(int *A, int e, int d) { // A[e..d] é um vetor de int
    int u, v, m;
    if (e == d) return A[e];
    m = (e+d)/2;
    u = X(A, e, m);
    v = X(A, m+1, d);
```

$$T(n) = 0, n \leq 1$$
$$T(n) = 2T(n/2) + n - 1$$

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

Encontra o maior elemento do vetor

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

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

Análise de Algoritmos Recursivos

- Como 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
 - ❑ A partir da recorrência, expanda os termos obtendo termos com entradas menores
 - ❑ Repita o processo até chegar no caso base
 - ❑ Substitua os valores com os termos de entradas menores já computados
 - ❑ Some os custos de todos os termos
 - ❑ Calcule a fórmula fechada do somatório

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):

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

Análise de Algoritmos Recursivos

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

$$T(n) = c + T(n-1)$$

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

$$T(n-2) = c + T(n-3)$$

...

$$T(1) = c + T(0)$$

$$T(0) = d$$

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

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

Análise de Algoritmos Recursivos

- 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 $O(n)$

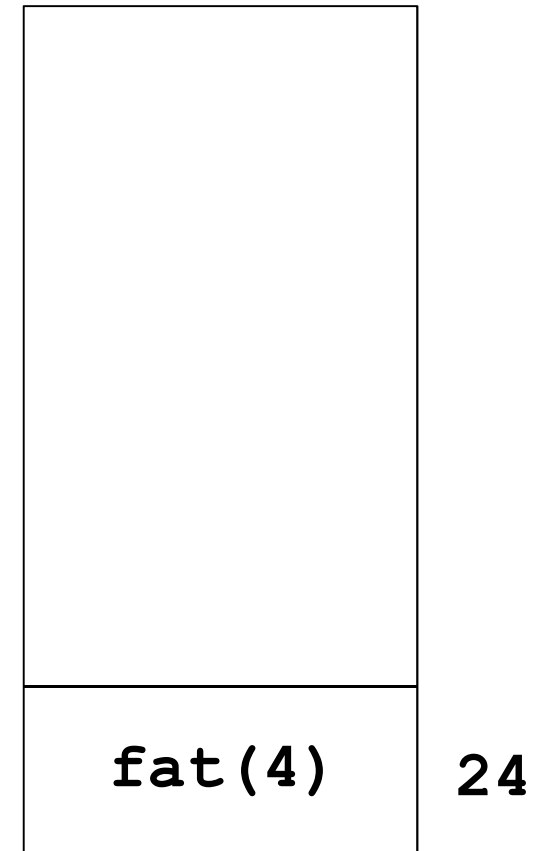
Análise de Algoritmos Recursivos

- 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

Análise de Algoritmos Recursivos

Resolvendo a equação de recorrência do
Seleção Recursivo (SRec):

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

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

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

$$T(n-2) = (n-3) + T(n-3)$$

...

$$T(2) = 1 + T(0)$$

$$T(1) = 0$$

$$T(n) = \underbrace{(n-1) + (n-2) + \dots + 2 + 1 + 0}$$

$$T(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

$$O(n^2)$$

Análise de Algoritmos Recursivos

■ E para a função max?

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

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

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

$$T(1) = 0$$

Expandindo a equação :

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

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

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

⋮

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

Substituindo os termos :

$$T(n) = 2^i T(n/2^i) + 1 + 2 + 4 + \dots + 2^{i-1}$$

Para colocar a Condição de Parada :

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

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

Logo :

$$T(n) = 2^i T(n/2^i) + \sum_{k=0}^{\log_2 n - 1} 2^k$$

$$T(n) = 0 + \frac{1 - 2^{\log_2 n}}{1 - 2} = n - 1$$

$$O(n)$$

Somatório de uma PG finita:

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

Teorema Mestre

Método “receita de bolo” para resolver recorrências do tipo

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

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

Teorema Mestre

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

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

- Este tipo de recorrência é típico de algoritmos “dividir para conquistar”
 - ❑ Dividem um problema em a subproblemas
 - ❑ Cada subproblema tem tamanho n/b
 - ❑ Cada chamada realiza um trabalho de custo $f(n)$
 - ❑ O caso base, normalmente omitido, tem um custo constante para um valor de n pequeno: $T(n)=c, n < k$

Teorema Mestre

■ Exemplo: Algoritmo Max

```
int Max(int *A, int e, int d) { // A[e..d] é um vetor de int
    int u, v, m;

    if (e == d) return A[e];
    m = (e+d)/2;
    u = Max(A, e, m);
    v = Max(A, m+1, d);

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


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

↳ $a = 2, b = 2 \text{ e } f(n) = 1$


Teorema Mestre

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

Compara-se a função $f(n)$ com o termo $n^{\log_b a}$

Caso 1: se $f(n) = O(n^{\log_b a - \varepsilon}) \rightarrow T(n) = \Theta(n^{\log_b a})$

 $f(n)$ é polinomialmente menor que $n^{\log_b a}$

Caso 2: se $f(n) = \Theta(n^{\log_b a}) \rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$

Caso 3: se $f(n) = \Omega(n^{\log_b a + \varepsilon}) \rightarrow T(n) = \Theta(f(n))$

 $f(n)$ é polinomialmente maior que $n^{\log_b a}$

Deve também satisfazer uma

Condição de Regularidade: $af(n/b) \leq cf(n), \quad c < 1, n \geq n_0$

Teorema Mestre

Intuição: a função $f(n)$ é comparada com $n^{\log_b a}$ 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: nos casos 1 e 3, a função $f(n)$ deve ser polinomialmente menor / maior do que $n^{\log_b a}$. Além disso, a função deve satisfazer uma condição de regularidade.

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.

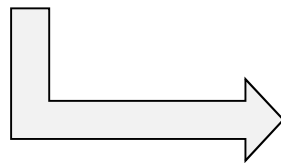
Teorema Mestre - Exemplos

$$\underline{T(n) = 9T(n/3) + n}$$

$$\left. \begin{array}{l} a = 9 \\ b = 3 \end{array} \right\} n^{\log_b a} = n^{\log_3 9} = n^2$$

$f(n) = n$ fazendo $\varepsilon = 1$ temos

$$f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{2-1}) = O(n)$$



Caso 1: $T(n) = \Theta(n^{\log_b a})$

$$\boxed{T(n) = \Theta(n^2)}$$

Teorema Mestre - Exemplos

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

$$\left. \begin{array}{l} a = 3 \\ b = 4 \end{array} \right\} n^{\log_b a} = n^{\log_4 3} = n^{0,793}$$

$$f(n) = n \log n \quad \text{fazendo } \varepsilon = 0,207 \text{ temos}$$

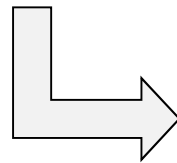
$$f(n) = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{0,793 + 0,207}) = \Omega(n)$$

Condição de Regularidade

$$af(n/b) \leq cf(n), \quad c < 1, n \geq n_0$$

$$3(n/4) \log(n/4) \leq (3/4)n \log n$$

$$\text{OK : } c = 3/4, n_0 = 1$$



Caso 3: $T(n) = \Theta(f(n))$

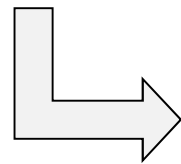
$$\boxed{T(n) = \Theta(n \log n)}$$

Teorema Mestre - Exemplos

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

$$\left. \begin{array}{l} a = 2 \\ b = 2 \end{array} \right\} n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = n - 1 \quad f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$



Caso 2: $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$$T(n) = \Theta(n \log n)$$

Teorema Mestre - Exemplos

$$\underline{T(n) = 3T(n/3) + n \log n}$$

$$\left. \begin{array}{l} a = 3 \\ b = 3 \end{array} \right\} n^{\log_b a} = n^{\log_3 3} = n$$

$$f(n) = n \log n \quad \text{n\~ao existe } \varepsilon \text{ tal que:}$$

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{1+\varepsilon})$$

Apesar de $f(n)$ ser maior que n , ela n\~ao \u00e9 polinomialmente maior...

$$\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n} = \log n$$

**Teorema mestre n\~ao
pode ser utilizado**