Verão 2019 - TÓPICOS DE PROGRAMAÇÃO

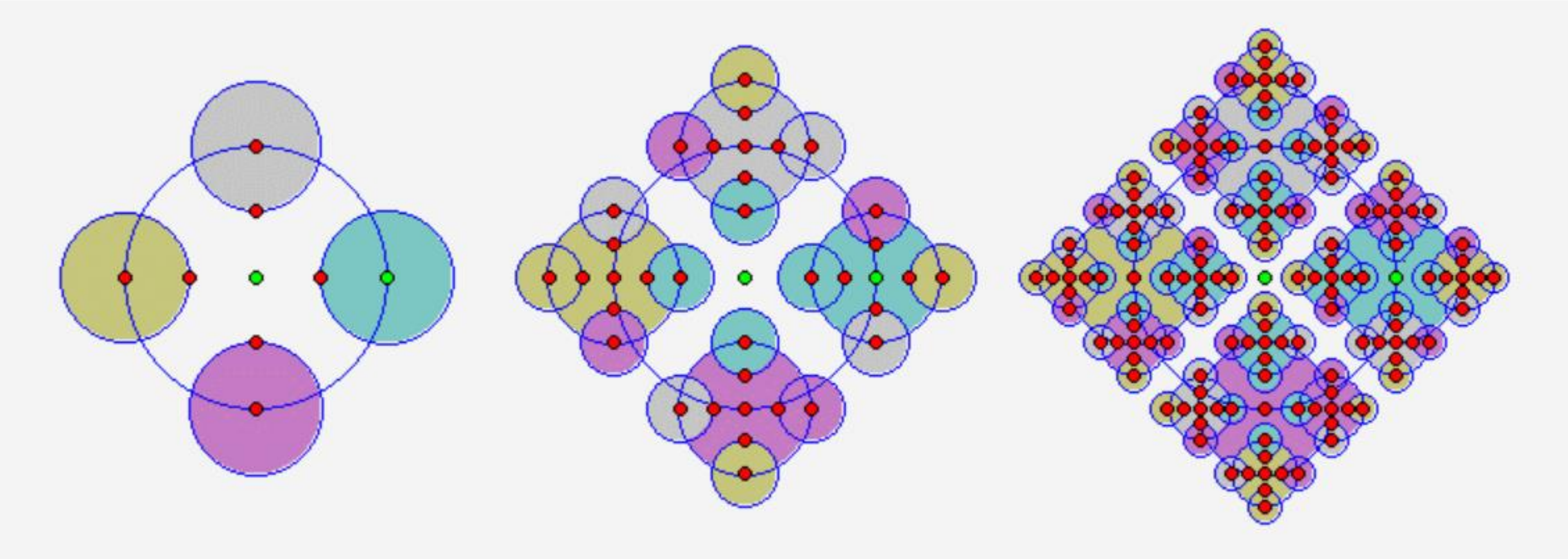
Prof. Dr. Leônidas O. Brandão (coordenador)

Profa. Dra. Patrícia Alves Pereira (ministrante)

Prof. Bernardo (Monitor)

Recursividade

Exemplos de imagens envolvendo recursividade são os *fractais* geométricos, na ilustração abaixa temos o **fractal *Tetra-Círculo*** (formado por circunferências com metade do raio original, construídas a partir dos pontos médios entre seus polos e seu centro)



Algoritmos recursivos (ou recorrentes)

Recursão é empregada em computação, tanto como estratégia de programação quanto na definição de expressões matemáticas e estruturas de dados.

Como exemplo do uso de recursão em definições:
o fatorial e os números de Fibonacci.

Algumas estruturas de dados também podem ser definidas de forma recursiva:
listas generalizadas e as árvores.

Algoritmos recursivos (ou recorrentes)

Como estratégia de programação, a recursividade é uma ferramenta poderosa que, quando bem empregada, resulta em algoritmos elegantes e legíveis.

Um algoritmo recursivo se caracteriza basicamente por conter chamadas a ele mesmo (recursividade direta),
ou chamadas para procedimentos que por sua vez invocam o algoritmo original (recursividade indireta).

Algoritmos recursivos (ou recorrentes)

É comum alguma resistência ao uso de técnicas recursivas na elaboração de algoritmos, motivada principalmente pelas dificuldades que surgem na depuração dos algoritmos e na análise de complexidade.

As dificuldades na depuração de algoritmos recursivos são consequências das estruturas hierárquicas (como pilhas e árvores) que estão associadas às chamadas recursivas.

Já a análise de algoritmos recursivos é mais complexa devido ao aparecimento de recorrências que surgem naturalmente da recursividade.

Algoritmos recursivos (ou recorrentes)

A ideia de qualquer algoritmo recursivo é simples:

- Se o problema é pequeno, resolva-o diretamente, como puder.
- Se o problema é grande, reduza-o a um problema menor do mesmo tipo .

Assim, você só precisa mostrar como obter uma solução da instância original a partir de uma solução da instância menor; o computador faz o resto.

A estrutura geral dos algoritmos recursivos seguem a mesma estrutura de funções matemáticas, e geralmente são constituídas de duas partes:
parte base e parte recursiva

Algoritmos recursivos (ou recorrentes)

Existem vantagens e desvantagens na utilização de recursividade em programação.

Algumas das vantagens do uso de recursão são:

- A clareza na interpretação do código
- Simplicidade e elegância na implementação.

Algumas das desvantagens são:

- Dificuldade para encontrar erros.
- Podem ser ineficientes.
- O espaço de memória que uma função recursiva consome para rascunho pode ser grande.

A principal preocupação na implementação de algoritmos recursivos é a questão de eficiência tanto de espaço quanto de tempo.

Algoritmos recursivos (ou recorrentes)

Função fatorial

Fat(0) = 1 na parte base, e
*Fat(n) = n * Fat(n-1)* na parte recursiva

Fat(5)
5!
5.4!
5.4.3!
5.4.3.2!
5.4.3.2.1!

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

Equação de recorrência T(n)

Uma recorrência é uma equação que descreve uma função em termos do seu valor em entradas menores

Útil para análise de complexidade de algoritmos recursivos ou do tipo “dividir para conquistar”

Fatorial: Equação de recorrência T(n)

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

Número de multiplicações quando $n = 1$ é zero

$$T(1) = 0$$

Número de comparações quando $n > 1$ é 1 mais o número de multiplicações para n-1

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

Fatorial: Equação de recorrência T(n)

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

Número de multiplicações quando $n = 1$ é zero

$$T(1) = 0$$

Número de comparações quando $n > 1$ é 1
mais o número de multiplicações para n-1

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

Outro exemplo: Sequência de Fibonacci

Sequência numérica proposta pelo matemático Leonardo Fibonacci no século XI.

Trata-se do exemplo clássico dos coelhos, em que Fibonacci descreve o crescimento de uma população desses animais.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n\text{-}1) + fibonacci(n\text{-}2),\ n>1$$

Outro exemplo: Sequência de Fibonacci

Sequência numérica proposta pelo matemático Leonardo Fibonacci no século XI.
Trata-se do exemplo clássico dos coelhos, em que Fibonacci descreve o crescimento de uma população desses animais.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

$Fib(0) = 1$ e $Fib(1) = 1$ na parte base, e

$Fib(n) = Fib(n-1) + Fib(n-2),\ n>1$ na parte recursiva

Exercício1: Sequência de Fibonacci

1) Escreva um algoritmo recursivo que calcule a sequencia de Fibonacci

2) Escreva a equação de recorrência T(n)

Exercício1: Sequência de Fibonacci

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

Número de chamadas recursivas

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

Algoritmo 1: função $fibo1(n)$

Seja a função $fibo1(n)$ que calcula o n-ésimo elemento da seqüência de Fibonacci.
Input: Valor de n
Output: O n-ésimo elemento da seqüência de Fibonacci
Function $fibo1(n)$

```
if n = 0 then
    return 0
else
    if n = 1 then
        return 1
    else
        return fibo1(n − 1) + fibo1(n − 2)
    end if
end if
```

Experimente rodar este algoritmo para $n = 100$:-)
A complexidade é $O(2^n)$.
(Mesmo se uma operação levasse um picosegundo, 2^{100} operações levariam 3×10^{13} anos = 30.000.000.000.000 anos.)

Algoritmo 2: função *fibo2*(n)

```
Function fibo2(n)
if n = 0 then
    return 0
else
    if n = 1 then
        return 1
    else
        penultimo ← 0
        ultimo ← 1
        for i ← 2 until n do
            atual ← penultimo + ultimo
            penultimo ← ultimo
            ultimo ← atual
        end for
        return atual
    end if
end if
```

A complexidade agora passou de $O(2^n)$ para $O(n)$.

Voce sabe que dá para fazer em $O(\log n)$?

Exercício 2: agora é com vc!

```
int valor (n,A) {
    if (n == 0) return 0;
    return valor (n−1, A) + A[n];
}
```

O que esse algoritmo faz?

Escreva a equação de recorrência T(n).

Exercício 2: soma recursiva

```
int soma (n,A){
    if  (n == 0)  return 0;
    return  soma (n−1, A) + A[n];
}
```

$T(1) = 1$

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

Exercício 3:

```
int valor (n,A) {
    if  (n == 1)  return A[1];
    int x = valor(n-1,A);
    if (x<A[n]) return A[n];
    return x;
}
```

O que esse algoritmo faz?
Escreva a equação de recorrência T(n).

Exercício 3: máximo

```
int max(n,A) {
    if  (n == 1)  return A[1];
    int x = max(n-1,A);
    if (x<A[n]) return A[n];
    return x;
}
```

$T(1) = 1$

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

Resolver recorrências

Resolver recorrências é uma arte, nem sempre fácil, e para tal utilizamos Métodos :

- da Substituição;
- da Árvore de Recursão ou;
- do Teorema Mestre.

Método da Árvore de Recursão

Exemplo – complexidade do Fatorial

Número de multiplicações quando $n = 1$ é zero

$$T(1) = 0$$

Número de comparações quando $n > 1$ é 1 mais o número de multiplicações para n-1

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

Método da Árvore de Recursão

Cada nó representa o custo de um único subproblema em algum lugar do conjunto de chamadas recursivas

Exercício 1 – complexidade do Fibonacci

Número de chamadas recursivas

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

Soma PG Finita

$$S = \frac{a_1\left(q^n - 1\right)}{q - 1}$$

Método da Árvore de Recursão

Cada nó representa o custo de um único subproblema em algum lugar do conjunto de chamadas recursivas

Exercício 2

$T(1) = 2$
$T(n) = T(n-1) + n \quad , \text{se } n \geq 2$

Método da Árvore de Recursão

Cada nó representa o custo de um único subproblema em algum lugar do conjunto de chamadas recursivas

Exercício 3

$$T(1) = 1$$

$$T(n) = T(n-1)+3n+2 \quad \text{para } n = 2,3,4,\ldots$$

Método da Árvore de Recursão

Exercício 4

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

Teorema Mestre

Método "receita de bolo" para resolver recorrências do tipo

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

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

O teorema mestre não se aplica a todas as recorrências!

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 $\mathbf{f}(\boldsymbol{n})$

Teorema Mestre (CORMEN, 2012): sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função assintoticamente positiva e seja $T(n)$ definida no domínio dos números inteiros não negativos pela recorrência

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

onde interpretamos que n/b significa $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então, $T(n)$ tem os seguintes limites assintóticos:

1. Se $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. Se $f(n) = \Theta\left(n^{\log_b a}\right)$, então $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. Se $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todos os n suficientemente grandes, então $T(n) = \Theta(f(n))$.

Teorema Mestre Simplificado:

Suponha

$$T(n) = a\,T(n/b) + c\,n^k$$

para algum $a \geq 1$ e $b > 1$ e onde n/b significa $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$.
Então, em geral,

se $a > b^k$ então $T(n) = \Theta(n^{\log_b a})$
se $a = b^k$ então $T(n) = \Theta(n^k \lg n)$
se $a < b^k$ então $T(n) = \Theta(n^k)$

Teorema Mestre – resumindo...

- O teorema mestre resolve recorrências que possuem a seguinte forma:

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

Cuidado!
Ao identificar a constante ***b***, observe que ela é o divisor na divisão ***n*/*b***.

Exemplo: para $T(2n/3)$ o valor de ***b*** é igual a ***3/2***, pois **2*n*/3 = *n* / 3/2**.

- ***n*** é o tamanho do problema
- ***a*** e ***b*** são constantes
- o valor de ***a*** é igual ao número de subproblemas no qual o problema original foi dividido
- ***n*/*b*** é o tamanho de cada um desses subproblemas
- a função ***f*(*n*)** representa o custo no tempo de cada chamada recursiva do algoritmo analisado.

Exercício 1 - resolva usando o teorema mestre

A recorrência a seguir é do algoritmo de ordenação Merge Sort

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

Exercício 1 - resolva usando o teorema mestre

A recorrência a seguir é do algoritmo de ordenação Merge Sort

T(1) = 1
T(n) = 2T(n/2) + n , se n≥2

se $a = b^k$ então $T(n) = \Theta(n^k \lg n)$

a = 2 **b = 2** **k = 1**

$\boldsymbol{como\ 2 = 2^1\ pelo\ Teorema\ Mestre\ temos\ que\ T(n) = \theta(n.lgn)}$

Exercício 2

Resolva a recorrência

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

usando o teorema mestre

Exercício 2 - solução

Resolva a recorrência

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

usando o teorema mestre

Caso 1 se aplica, temos

$$T(n) = \Theta(n^2)$$

Exercício 3

Resolva a recorrência

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

usando o teorema mestre

Exercício 3 - solução

Resolva a recorrência

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

usando o teorema mestre

Caso 2 se aplica, temos

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

Exercício 4 - resolva usando o teorema mestre

$$T(n) = 3T(n/4) + n\log_2 n$$

Suponha

$$T(n) = a\,T(n/b) + f(n)$$

para algum $a \geq 1$ e $b > 1$ e onde n/b significa $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$.
Então, em geral,

$$\begin{array}{llll}
\text{se} & f(n) = \mathrm{O}(n^{\log_b a - \epsilon}) & \text{então} & T(n) = \Theta(n^{\log_b a}) \\
\text{se} & f(n) = \Theta(n^{\log_b a}) & \text{então} & T(n) = \Theta(n^{\log_b a} \lg n) \\
\text{se} & f(n) = \Omega(n^{\log_b a + \epsilon}) & \text{então} & T(n) = \Theta(f(n))
\end{array}$$

para qualquer $\epsilon > 0$.

Relembrando....

Ordem de Complexidade	Nome
$O(1)$	Constante
$O(log\ n)$	Logarítmica
$O(n)$	Linear
$O(n\ log\ n)$	Linearitmica
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial
$O(n!)$	Exponencial

Exercício 4 - solução

1º passo: identificar as constantes a e b, e a f(n)

$$a = 3, \quad b = 4, \quad f(n) = n \log_2 n$$

2º passo: calcular $\log_b a$

$$\log_b a = \log_4 3 \approx 0{,}79$$

Exercício 4 - solução

3º passo: identificar se T(n) atende os casos do Teorema mestre

Veja que esse exemplo os casos 1 e 2 não são satisfeitos.

Exercício 4 - solução

4º passo: verificando as restrições do caso 3

Vamos escolher $\varepsilon \approx 0{,}21$, para que $\log_b a + \varepsilon = 1$, para facilitar os cálculos.

$$\Omega(n^{\log_b a+\epsilon}) = \Omega(n^{0{,}7924812504...+0{,}2075187496...}) = \Omega(n)$$

e $f(n) = n \log_2 n = \Omega(n)$, pois $f(n)$ é assintoticamente maior.

Exercício 4 - solução

5º passo: verificando a condição de regularidade $af(n/b) \leq cf(n)$

$$
\begin{aligned}
af(n/b) &\leq cf(n) \\
3 \times \frac{n}{4} \log_2 \frac{n}{4} &\leq cn \log_2 n \\
\frac{3}{4} n \left(\log_2 n - \log_2 4\right) &\leq cn \log_2 n \\
\frac{3}{4} n \left(\log_2 n - 2\right) &\leq cn \log_2 n \\
\frac{3}{4} n \log_2 n - 2 \times \frac{3}{4} n &\leq cn \log_2 n \\
\frac{3}{4} n \log_2 n - \frac{3}{2} n &\leq cn \log_2 n
\end{aligned}
$$

Exercício 4 - solução

Se escolhermos $c = 3/4 < 1$, então

$$\begin{aligned} \frac{3}{4}n\log_2 n - \frac{3}{2}n \leq \frac{3}{4}n\log_2 n \\ -\frac{3}{2}n \leq 0 \\ \frac{3}{2}n \geq 0 \\ n \geq 0 \end{aligned}$$

Ou seja, a condição é satisfeita para $c = 3/4$ e qualquer n maior do que zero. Concluímos que $T(n)$ se enquadra no terceiro caso do teorema mestre, portanto $T(n) = \Theta(f(n)) = \Theta(n\log_2 n)$.