Recorrências e Equações de recorrência

Jorge E. S. Souza

Recursividade

Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser recursivo

❑ Recursividade permite descrever algoritmos de forma mais clara e concisa.

❑ Especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.

Fatorial

- ❑ $0! = 1$
- ❑ $n! = n(n - 1)!$

$$
\begin{aligned}
6! &= 6 \times (5!) \\
6! &= 6 \times 5 \times (4!) \\
6! &= 6 \times 5 \times 4 \times (3!) \\
6! &= 6 \times 5 \times 4 \times 3 \times (2!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times (1!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times (0!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\
6! &= 720.
\end{aligned}
$$

Fatorial

- ❑ $0! = 1$
- ❑ $n! = n(n - 1)!$

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

Recursividade

Normalmente, as funções recursivas são divididas em duas partes:

- ❑ Condição de parada.

  Evitar loops infinitos

- ❑ Chamada recursiva.

  Pode ser direta (mais comum) ou indireta

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

Chamada recursiva

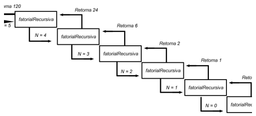
Quando uma chamada de função é feita, é criado um registro de ativação na pilha de execução do programa

❑ O registro de ativação guarda;

  ❑ Os parâmetros e variáveis locais da função;

  ❑ O ponto de retorno da função;

❑ Quando a função termina, o registro de ativação é desempilhado e a execução volta ao subprograma que chamou a função.



Custo de tempo e espaço

⇒ A complexidade de tempo do fatorial recursivo é O(n).

- ❑ Calculado via equações de recorrência;
- ❑ Complexidade de espaço também é O(n);

⇒ Na implementação não recursiva, a complexidade de espaço é O(1):

$$
\begin{aligned}
6! &= 6 \times (5!) \\
6! &= 6 \times 5 \times (4!) \\
6! &= 6 \times 5 \times 4 \times (3!) \\
6! &= 6 \times 5 \times 4 \times 3 \times (2!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times (1!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times (0!) \\
6! &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1
\end{aligned}
$$

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

Use com moderação

Recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos.

Exemplo:

A sequência: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Pode ser definida por:

- ❑ F(n) = F(n-1) + F(n-2), para n > 2 e
- ❑ F(1) = F(2) = 1, para n <= 2. sendo n positivo e não nulo.

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

Exponencial

fib (n-1) + fib (n-2)

fib (n-1) + fib (n-2) + fib (n-1) + fib (n-2)

fib (n-1) + fib (n-2) + fib (n-1) + fib (n-2) + fib (n-1) + fib (n-2) + fib (n-1) + fib (n-2)

.

.

.

Exponencial o tempo ? no espaço ? nos dois ?

Ineficiente

⇒ Termos F(n-1) e F(n-2) são computados independentemente e repetidas vezes;

⇒ Número de chamadas recursivas é igual ao número de fibonacci sendo calculado;

⇒ Custo para o cálculo de F(n) é de complexidade exponencial;

Então como calcular?

Implementação iterativa:

```
int fib (int n) {
        int  f_m2 = 1;  int  f_m1 = 1;
        int f;
        for (int i = 2. i <= n;  i++) {
                f = f_m1 + f_m2;
                f_m2 = f_m1;
                f_m1 = f;
        }
        return f;
}
```

- ❏ Complexidade de tempo: O(n)
- ❏ Complexidade de espaço: O(1)

Quando usar recursividade?

Problemas cuja implementação iterativa é complexa e requer uso explícito de uma pilha:

- ❏ Algoritmos tipo dividir para conquistar (quicksort)
- ❏ Caminhamento em árvores
- ❏ Busca exaustiva

Recorrência

- Quando um algoritmo contém uma chamada recursiva, seu tempo de execução pode frequentemente ser descrito por uma recorrência;

- Com o ferramental aprendido até o momento, não somos capazes de analisar a complexidade de algoritmos recursivos;

- Para os algoritmos recursivos, a ferramenta principal desta análise não é uma somatória, mas um tipo especial de equação chamada **relação de recorrência**.

- Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores;

Equação de recorrência

- ❏ Para cada procedimento recursivo é associada uma função de complexidade T(n) desconhecida, onde n mede o tamanho dos argumentos para o procedimento.

- ❏ Equação de recorrência: maneira de definir uma função por uma expressão envolvendo a mesma função.

Como montar a equação de recorrência?

Considere o algoritmo “pouco formal” ao lado:

❏ O algoritmo inspeciona **n** elementos de um conjunto qualquer;

❏ De alguma forma, isso permite descartar ⅔ dos elementos e fazer uma chamada recursiva sobre um terço do conjunto original.

```
Algoritmo Pesquisa(vetor)
   if vetor.size()<= 1 then
        inspecione elemento;
   else
       inspecione cada elemento recebido (vetor);
       Pesquisa(vetor.subLista(1,(vetor.size()/3));
   end if
end.
```

Montando a equação de recorrência

1| Algoritmo Pesquisa(vetor)	
2| if vetor.size()<= 1 then	$\Theta(1)$
3| inspecione elemento;	$\Theta(1)$
3| else	
5| inspecione cada elemento recebido (vetor);	
6| Pesquisa(vetor.subLista(1,(vetor.size()/3));	
7| end if	
8| end.	

<u>Caso base da recursão:</u>

- O custo da linha 2 é $\Theta(1)$;
- O custo da linha 3 é $\Theta(1)$;

Montando a equação de recorrência

```
1|  Algoritmo Pesquisa(vetor)
2|      if vetor.size()<= 1 then                              Θ(1)
3|          inspecione elemento;                              Θ(1)
3|      else
5|          inspecione cada elemento recebido (vetor);        Θ(n)
6|          Pesquisa(vetor.subLista(1,(vetor.size()/3));      T(n/3)
7|      end if
8|  end.
```

<u>Caso geral da recursão:</u>

- O custo da linha 5 é $\Theta(n)$;
- A linha 6 é onde a própria função Pesquisa é chamada em um conjunto reduzido.

Montando a equação de recorrência

Algoritmo	Custo
Algoritmo Pesquisa(vetor)	
if vetor.size()<= 1 then	$\Theta(1)$
inspecione elemento;	$\Theta(1)$
else	
inspecione cada elemento recebido (vetor);	$\Theta(n)$
Pesquisa(vetor.subLista(1,(vetor.size()/3));	$T(n/3)$
end if	
end.	

{ Base }

{ recursão }

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

Montando a equação de recorrência

Algoritmo Pesquisa(vetor)	
if vetor.size()<= 1 then	$\Theta(1)$
inspecione elemento;	$\Theta(1)$
else	
inspecione cada elemento recebido (vetor);	$\Theta(n)$
Pesquisa(vetor.subLista(1,(vetor.size()/3));	$T(n/3)$
end if	
end.	

{ Base }

{ recursão }

⇒ Resolva a equação de recorrência...

Métodos de resolução de recorrências

⇒ Expandir, Conjecturar e Verificar ou Método de substituição: mais geral, mas é necessário adivinhar a forma da solução.

⇒ Método de expansão da árvore de recorrência: pouco formal, mas bastante intuitivo.

⇒ Teorema Mestre: cobre muitos casos, mas não todos.

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

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

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

$$\begin{aligned} T(n) &= n + T(n/3) \\ T(n/3) &= n/3 + T(n/3/3) \end{aligned}$$

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

$$\begin{aligned}
T(n) &= n + T(n/3) \\
T(n/3) &= n/3 + T(n/3/3) \\
T(n/3/3) &= n/3/3 + T(n/3/3/3) \\
T(n/3/3/3) &= n/3/3/3 + T(n/3/3/3/3) \\
&\ldots \\
T(n/3/3\ldots/3) &= n/3/3/3\ldots/3 + T(n/3/3/3\ldots/3)
\end{aligned}$$

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

$$\begin{aligned}
T(n) &= n + T(n/3) \\
T(n/3) &= n/3 + T(n/3/3) \\
T(n/3/3) &= n/3/3 + T(n/3/3/3) \\
T(n/3/3/3) &= n/3/3/3 + T(n/3/3/3/3) \\
&\dots \\
T(n/3/3\dots/3) &= n/3/3/3\dots/3 + T(n/3/3/3\dots/3) \\
T(1) &= 1
\end{aligned}$$

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

$$\begin{aligned}
T(n) &= n + \cancel{T(n/3)} \\
\cancel{T(n/3)} &= n/3 + \cancel{T(n/3/3)} \\
\cancel{T(n/3/3)} &= n/3/3 + \cancel{T(n/3/3/3)} \\
\cancel{T(n/3/3/3)} &= n/3/3/3 + \cancel{T(n/3/3/3/3)} \\
&\dots \\
T(n/3/3\dots/3) &= n/3/3/3\dots/3 + \cancel{T(n/3/3/3\dots/3)} \\
\cancel{T(1)} &= 1
\end{aligned}$$

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

T(n) = n + n/3 + n/3/3 + … + n/3/3/3.../3 + 1

⇒ A fórmula representa a soma de uma série geométrica de razão ⅓, multiplicada por n, e adicionada de T(n/3/3/3/3.../3), que menor ou igual a 1.

$$T(n) = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i + 1$$

Expandir

$$T(n) = \begin{cases} 1, \text{ se } n \leq 1 \\ T(n/3) + n, \text{ caso contrário} \end{cases}$$

T(n) = n + n/3 + n/3/3 + … + n/3/3/3.../3 + 1

$$T(n) = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i + 1 \;\rightarrow\; usando: \;\; \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$T(n) = n\left(\frac{1}{1-1/3}\right) + 1 = \frac{3n}{2} + 1$$

Portanto T(n) ∈ Θ (n)

Algumas fórmulas úteis

Série Aritmética	Série Geométrica	Série Harmônica
$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \equiv O\left(n^2\right)$	$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x\text{-}1} \equiv O\left(x^n\right)$	$\sum_{i=1}^{n} \frac{1}{k} = ln\ n + O(1) \equiv O\left(ln\ n\right)$
$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \equiv O\left(\frac{n^3}{3}\right)$	$\sum_{i=0}^{n} \frac{i}{2^i} = 2$	$\sum_{i=1}^{n} \frac{1}{k} = ln\ n + O(1) \equiv O\left(ln\ n\right)$

Expandir, Conjecturar e Verificar

- **Expandir**: usar repetidamente a recorrência para expandir a expressão a partir do n-ésimo termo até o caso base.

- **Conjecturar**: a expansão nos ajuda a conjecturar a solução da recorrência.

- **Verificar**: a conjectura é finalmente verificada por indução.

Exemplo 1:

Resolva a seguinte equação de recorrência:

$$s(1) = 2$$

$$s(n) = 2s(n - 1) \text{ para } n \geq 2$$

Exemplo 1:

Resolva a seguinte equação de recorrência:

$$\left.\begin{array}{l} s(1) = 2 \\ s(n) = 2s(n - 1) \text{ para } n \geq 2 \end{array}\right.$$

$\left\{ \text{Base} \right\}$

$\left\{ \text{recursão} \right\}$

Exemplo 1:

Expandindo. Aplicando a definição para n, n - 1, n - 2, etc.:

s(1) = 2 Base

s(n) = 2s(n - 1) recursão

$$= 2[2s(n-2)] \quad = 2^2 s(n-2)$$
$$= 2^2[2s(n-3)] \quad = 2^3 s(n-3)$$
$$= 2^3[2s(n-4)] \quad = 2^4 s(n-4)$$

Exemplo 1:

Expandindo. Aplicando a definição para n, n - 1, n - 2, etc.:

$$\mathbf{s(1) = 2} \quad \text{Base}$$

$$\mathbf{s(n) = 2s(n - 1)} \quad \text{recursão}$$

$$= 2[2s(n - 2)] \quad = 2^2 s(n - 2)$$
$$= 2^2[2s(n - 3)] \quad = 2^3 s(n - 3)$$
$$= 2^3[2s(n - 4)] \quad = 2^4 s(n - 4)$$

...

Analisando o padrão, conjecturamos que, após k expansões, a equação tem a forma: $\mathbf{s(n) = 2^k s(n - k)}$

Exemplo 1:

❑ A expansão tem que parar quando $n - k = 1$, ou seja, quando $k = n - 1$. Nesse ponto temos:

Exemplo 1:

- A expansão tem que parar quando n - k = 1, ou seja, quando k = n - 1. Nesse ponto temos:

$$s(n) = 2^{n-1}s[n - (n - 1)]$$

Exemplo 1:

- A expansão tem que parar quando n - k = 1, ou seja, quando k = n - 1. Nesse ponto temos:

$$\begin{aligned} s(n) &= 2^{n-1}s[n - (n - 1)] \\ &= 2^{n-1}s(1) \\ &= 2^{n-1}(2) \\ &= 2^{n} \end{aligned}$$

Ainda falta provar que 2^n é a solução da recorrência.

Indução

- ❑ A base da indução é **$s(1) = 2^1$** que é verdade pelo caso base da definição recursiva.

Supondo que $s(k) = 2^k$, queremos provar que $s(k + 1) = 2^{k+1}$:

$$s(k + 1) = 2s(k)$$
$$s(k + 1) = 2(2^k)$$
$$s(k + 1) = 2^{k+1}$$

E portanto fica provada a relação de recorrência.

Métodos de resolução de recorrências

⇒ Expandir, Conjecturar e Verificar ou Método de substituição: mais geral, mas é necessário adivinhar a forma da solução.

⇒ Método de expansão da árvore de recorrência: pouco formal, mas bastante intuitivo.

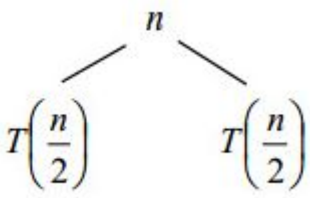
⇒ Teorema Mestre: cobre muitos casos, mas não todos.

Método de Expansão da Árvore de Recorrência

Exemplo

$$T(n) = \begin{cases} \Theta(1) & \text{, se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{, se } n > 1 \end{cases}$$

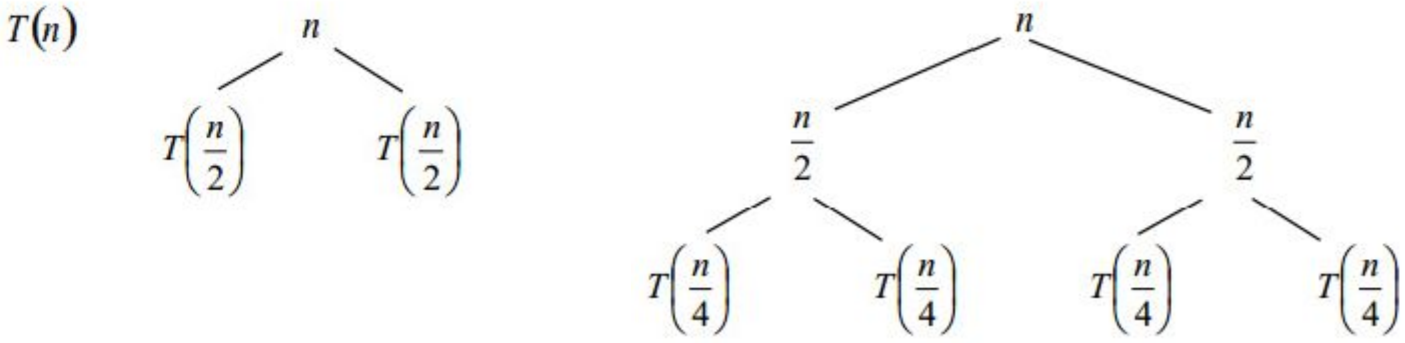
$T(n)$



Método de Expansão da Árvore de Recorrência

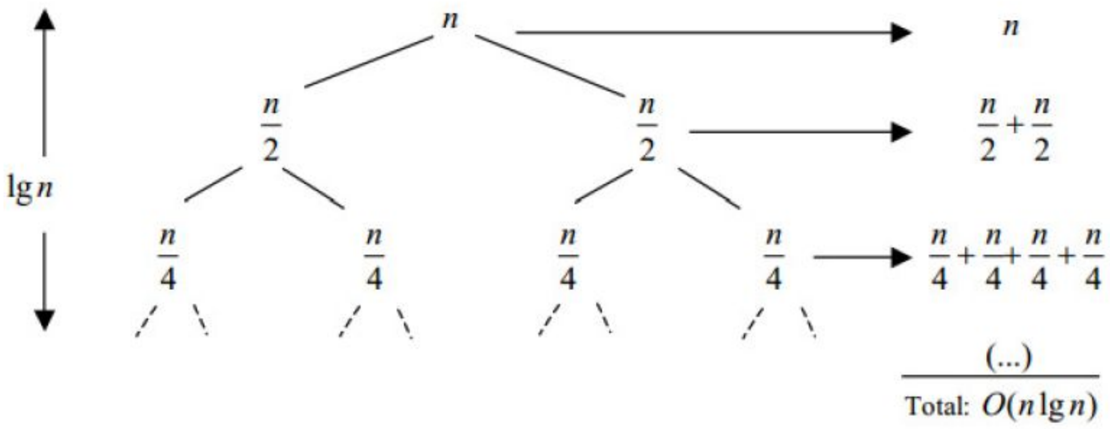
Exemplo

$$T(n) = \begin{cases} \Theta(1) & \text{, se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{, se } n > 1 \end{cases}$$



Exemplo

$$T(n) = \begin{cases} \Theta(1) & \text{, se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{, se } n > 1 \end{cases}$$



Árvore de Recorrência para a expressão $T(n) = 2T(\frac{n}{2}) + n$

Método de Expansão da Árvore de Recorrência

⇒ Talvez o método mais intuitivo.

⇒ Consiste em desenhar uma árvore cujos nós representam os tamanhos dos problemas.

⇒ Cada nível **i** contém todos os subproblemas de profundidade **i**.

⇒ Dois aspectos importantes:

- A altura da árvore.
- O número de passos executados de cada nível.

⇒ A solução da recorrência (tempo de execução do algoritmo) é a soma de todos os passo de todos os níveis.

Recorrências Lineares de Primeira Ordem

- Uma recorrência da forma

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

é chamada de recorrência linear de primeira ordem.

- Quando f(n) é uma constante r, tem-se

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

- Quando **g(n) = 0** para todo **n** dizemos que a recorrência é homogênea.

Recorrências Lineares de Primeira Ordem

Teorema 1

Para as constantes positivas a e r e qualquer função g definida nos naturais, a solução para a recorrência linear de primeira ordem

$$T(n) := \begin{cases} rT(n-1) + g(n), & \text{se } n > 1, \\ a, & \text{se } n = 1. \end{cases}$$

é dada por

$$T(n) = r^{n-1}a + \sum_{i=0}^{n-2} r^i g(i)$$

Método mestre

O método mestre fornece uma receita para a solução de recorrências da forma:

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

onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva.

Método mestre

Considere a recorrência:

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

onde **$a \geq 1$** e **$b > 1$** são constantes, f(n) é uma função assintoticamente positiva e **n/b** pode ser **$\lfloor n/b \rfloor$** ou **$\lceil n/b \rceil$**.

Então:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$ então $T(n) = \Theta(n^{\log_b a})$

Método mestre

Considere a recorrência:

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

onde **a ≥ 1** e **b > 1** são constantes, f(n) é uma função assintoticamente positiva e **n/b** pode ser **⌊n/b⌋** ou **⌈n/b⌉**.

Então:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$ então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$ então $T(n) = \Theta(n^{\log_b a} \log n)$

Método mestre

Considere a recorrência:

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

onde **a ≥ 1** e **b > 1** são constantes, f(n) é uma função assintoticamente positiva e **n/b** pode ser **⌊n/b⌋** ou **⌈n/b⌉**.

Então:

1. Se $f(n) = O(n^{\log_b a-\epsilon})$ para alguma constante $\epsilon > 0$ então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$ então $T(n) = \Theta(n^{\log_b a} \log n)$
3. 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 n suficientemente grande, então $T(n) = \Theta(f(n))$

wtf

Resultados

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

a ≥ 1 e **b > 1**

Exercícios 1

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

Exercícios 1

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

a = ?

b = ?

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a}\log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = O(n^{\log_b a-\epsilon})$

Exercícios 1

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

a = 9

b = 3

o b sempre vai ser inverso, então 1/3 fica 3/1

ϵ = 1

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = O(n^{\log_b a - \epsilon})$

$n \leq c.\, n^{\log_3 9 - \epsilon}$

n é equivalente a f(n) ali em baixo

$n \leq c.\, n^{2 - \epsilon}$

$n \leq c.\, n$ ∴ verdade

Pelo Caso 1, T(n) = Θ(n²).

Exercícios 2

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

a = ? 1

b = ? 3

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a}\log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = O(n^{\log_b a-\epsilon})$

1 é equivalente ao f(n)

Exercícios 2

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

a = 1

b = 3/2

o b sempre vai ser inverso, então 2/3 fica 3/2

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \epsilon})$ e $af(n/b) \leq cf(n),\ c < 1$

Prova: $f(n) = O(n^{\log_b a - \epsilon})$

pq tá usando 1?

$$1 \leq c.\ n^{\log_{1,5} 1 - \epsilon}$$

$$1 \leq c.\ n^{0 - \epsilon} \quad \therefore \text{ absurdo}$$

Pelo Caso 1, falhou.

Exercícios 2

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

a = 1

b = 3/2

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a}\log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \le cf(n)$, $c < 1$

Prova: $f(n) = \Theta(n^{\log_b a})$

aqui seria aquele esquema de estar no meio, mas usamos a igualdade.

$1 = c. n^{\log_{1,5} 1}$

$1 = c. n^0$

$1 = c. 1$ ∴ verdade

Pelo Caso 2, T(n) = Θ(log(n)).

Exercícios 3

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

a = ?

b = ?

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = O(n^{\log_b a-\epsilon})$

Exercícios 3

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

$a = 3$

$b = 4$

$\epsilon = 0,2924$

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = O(n^{\log_b a-\epsilon})$

$n \log n \leq c.\ n^{\log_4 3 - \epsilon}$

$n \log n \leq c.\ n^{\log_4 2}$

$n \log n \leq c.\ n^{0,5}$ $\quad \therefore$ absurdo

Pelo Caso 1, falhou.

Exercícios 3

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

a = 3

b = 4

ϵ = ?

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a}\log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a+\epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = \Theta(n^{\log_b a})$

$n \log n = c.\ n^{\log_4 3}$

$n \log n = c.\ n^{0,79}$ $\therefore$ absurdo

Pelo Caso 2, falhou.

Exercícios 3

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

a = 3

b = 4

ϵ = 0,21

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

$T(n)$	Se
$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \epsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \epsilon})$ e $af(n/b) \leq cf(n)$, $c < 1$

Prova: $f(n) = \Omega(n^{\log_b a + \epsilon})$

$n \log n \geq c . n^{\log_4 3 + \epsilon}$

$n \log n \geq c . n^1$ ∴ verdade

af(n/b) ⇒ 3(n/4) log (n/4) ≤ (3/4)n log n

Pelo Caso 3, T(n) = Θ(n log n).

Obrigado

jorge@imd.ufrn.br