Governo Federal

Ministério da Educação

departamento de Informática

Universidade Federal do Maranhão

A Universidade que Cresce com Inovação e Inclusão Social

Recorrências

Estrutura de Dados II

Recorrência

- Quando um algoritmo contem uma chamada recursiva a ele próprio, seu tempo de execução pode frequentemente ser descrito por **recorrência**.

- Recursividade permite descrever algoritmos de forma mais clara e concisa:
  - Especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas

Recorrência

S(inteiro positivo *n*)
//função que calcula iterativamente o valor *S*(*n*)
//para a sequência *S* do Exemplo 1
Variáveis locais:
inteiro *i*   //índice do laço
ValorAtual   //valor atual da função *S*

  se *n* = 1 **então**
    retorne 2
  senão
    i = 2
    ValorAtual = 2
    enquanto *i* < = *n* **faça**
    ValorAtual = 2 * *ValorAtual*
    i = *i* + 1
  fim do enquanto

  //agora *ValorAtual* tem o valor *S(n)*
  retorne *ValorAtual*
fim do se
fim da função *S*

S(inteiro positivo *n*)
//função que calcula o valor *S*(*n*) de forma recorrente
//para a sequência *S* do Exemplo 1

  se *n* = 1 **então**
    retorne 2
  senão
    retorne 2 * *S*(*n* – 1)
  fim do se
fim da função *S*

FONTE: GERSTING, Judith L. **Fundamentos matemáticos para a ciência da computação**. LTC, 2001.

Procedimento Recursivo

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

- Obtemos uma equação de recorrência para f(n).

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

Definição

- A definição e a utilização de relações de recorrência extrapolam o trato de algoritmos.
- Na verdade ela é uma ferramenta matemática para explicar funções em série. Vamos considerar uma exemplo.
- A série matemática conhecida como *Série de Fibonacci* ou *Números de Fibonacci*. Ela é assim definida:

Definição

- Série de Fibonacci:
  - o primeiro número da série é 1;
  - o segundo número é 1;
  - a partir do terceiro, cada número é resultado da soma dos dois anteriores.
  - Segundo esta definição, a sequência é assim disposta:
  - 1 1 2 3 5 8 13 21 34 55 89 ...

Definição

- Esta série pode ser expressa, de forma mais precisa e matemática, com a seguinte expressão:

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-1) + F(n-2) \end{cases}$$

- Perceba que esta definição é idêntica aquela dada acima e aplica-se da mesma forma na busca dos valores:

$$\begin{aligned} &F(1) = 1 \\ &F(2) = 1 \\ &F(3) = F(2) + F(1) = 1 + 1 = 2 \\ &F(4) = F(3) + F(2) = 2 + 1 = 3 \\ &F(5) = F(4) + F(3) = 3 + 2 = 5 \\ &\qquad\quad \vdots \qquad\qquad\quad \vdots \end{aligned}$$

Definição

- a Série de Fibonacci é uma relação de recorrência. Sabendo disto, vejamos a sua definição formal:

- *“Relação de Recorrência é uma equação ou inequação que descreve uma função ou série numérica utilizando-se dela própria na definição.”*

Exercício

Apresente os cinco primeiros valores da sequência T expressa pela relação de recorrência abaixo:

1. **$T(1) = 1$**
2. **$T(n) = T(n - 1) + 3$ para $n \geq 2$**

Definição

- Um outro exemplo bastante comum é o cálculo do fatorial de um número inteiro. Ele é definido matematicamente como:

$$\begin{cases} 1! = 1 \\ n! = n \cdot (n-1)! \end{cases} \quad \text{ou} \quad \begin{cases} F(1) = 1 \\ F(n) = n \cdot F(n-1) \end{cases}$$

Definição

- Não é muito complicado perceber uma conexão entre esta forma de expressar uma relação de recorrência e a sua solução via algoritmos recursivos.

- Se este raciocínio é válido neste sentido, da relação de recorrência ao algoritmo, o inverso também é.

- Ou seja, dado um algoritmo, é possível encontrar a relação de recorrência capaz de descrevê-lo.

Exemplo: Busca Binária

```
procedimento busca_binária( v: vetor[1..N] de inteiros;
                                x,min,max: inteiro ): inteiro
variáveis
    meio: inteiro;
início
    se max < min então
        retorna –1;         /* Não encontrou o elemento */
    meio := ( min + max ) / 2;
    se v[meio] = x então
        retorna meio;       /* Encontrou o elemento na posição meio */
    se x < v[meio] então
        retorna busca_binária( v, x, min, meio-1 );
    senão
        retorna busca_binária( v, x, meio+1, max );
    fim
fim
```

Exemplo: Busca Binária

- Neste algoritmo é buscado um valor em um vetor de números inteiros ordenados crescentemente através do método de busca binária

- Encontra-se o meio do vetor;
  – se este meio é o elemento procurado, a busca termina;
  – caso contrário, se o elemento buscado é menor do que aquele presente no meio do vetor, ele somente poderá estar na parte inferior do mesmo;
  – senão, na parte superior.
  – A busca irá terminar quando não tivermos mais intervalos de valores a pesquisar (max < min).

Exemplo: Busca Binária

- Para simplificar, vamos contar somente o número de passagens em cada uma das linhas válidas.

- Em cada chamada ao procedimento, as linhas serão executadas apenas uma vez, com exceção daquelas dentro dos testes, que podem não ser executadas.

- Supondo que não seja encontrado o elemento, será realizado o primeiro teste, o cálculo da variável meio, o segundo e o terceiro testes.

- Desta forma, serão quatro linhas executadas ao todo.

Exemplo: Busca Binária

- Na última chamada recursiva, quando o valor é determinado como não presente no vetor, pois o intervalo de trabalho não tem valores válidos, somente um teste será realizado.

- Assim, podemos descrever este algoritmo na seguinte forma recorrente:

$$\begin{cases} T(1) = 1 \\ T(n) = T\left(\frac{n}{2}\right) + 4 \end{cases}$$

Exemplo: Busca Binária

- Ou seja, o trabalho para solucionar o problema com um elemento na entrada (análogo ao caso onde não há mais elementos a pesquisar), será de 1 operação - teste de verificação do final.

- Se o problema tiver uma entrada maior, serão necessárias 4 operações (testes e cálculo do meio) e uma chamada recursiva para o mesmo problema, só que com uma entrada equivalente a metade da original (n/2).

Exemplo: Busca Binária

- Assim, é sempre possível extrairmos uma relação de recorrência de um algoritmo recursivo.
- O ponto de término da recursão será o caso base da recorrência.
- O número de chamadas recursivas aparece na definição recorrente e o trabalho adicional, medido por operações se necessário, no acréscimo à esta definição.
- Veja mais um exemplo:

$$\begin{cases} T(1) = 2 \\ T(n) = 2T\left(\frac{n}{2}\right) + n \end{cases}$$

Exemplo: Busca Binária

- Se esta relação de recorrência representa um algoritmo, ele necessita de duas operações para concluir o caso base (não é possível dizer que tipo de operação).

- Não sendo este caso, ele divide o problema nas metades (T(n/2)) e executa recursivamente o algoritmo para estas duas metades (**2**T(n/2)).

- Ele ainda faz um trabalho adicional de ***n*** operações, que pode ocorrer antes, entre ou depois das duas chamadas recursivas (para a quantificação é indiferente o momento no qual isso acontece).

Métodos de Solução

- 3 formas de se buscar esta solução: por substituição ou prova inteligente; por árvore de recursão; e, pelo método mestre.
  - Vamos analisar os métodos de resolução:
    - Por árvore de recursão
    - Método mestre

Árvore de Recursão

- Uma relação de recorrência, assim como um algoritmo recursivo, apresenta uma dificuldade adicional na busca da solução
  - pois não conhecemos o seu ponto de parada, ou seja, quando acontece a última chamada recursiva
  - se soubermos o número de vezes que uma recorrência é executada, teremos apenas que somá-la para encontrar o resultado final
  - A solução de uma recorrência por história completa trabalha justamente neste horizonte

Árvore de Recursão

- O objetivo é “abrir” a recorrência tentando deduzir coisas sobre ela de forma a chegarmos ao total.
- Ou seja, tenta-se mostrar a sua história. Vamos utilizar como primeiro exemplo a seguinte relação:

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n/2) + n \end{cases}$$

Árvore de Recursão

- Supondo uma entrada ou um dado de tamanho *n*, no nível mais alto a recursão trabalhará o equivalente a este *n* acrescida de duas chamadas recursivas a ela mesma com uma entrada de metade do original.
- Perceba que cada uma destas chamadas recursivas, no primeiro nível, terá a forma:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Árvore de Recursão

- Seguindo o mesmo raciocínio:

$$T(n/4) = 2T(n/8) + n/4$$

$$T(n/8) = 2T(n/16) + n/8$$

- Então, da relação pode ser feita a seguinte leitura :
  - ao receber uma dada entrada (independente do tamanho), ela trabalha duas vezes, de forma análoga, com uma entrada com tamanho de metade da original; depois, realiza um trabalho adicional equivalente (da mesma ordem) ao que entrou.

Árvore de Recursão

- Segundo esta idéia, esta recorrência poderia ser escrita assim:

$$T(n) = 2\left(2T\left(n/4\right) + n/2\right) + n$$

- E assim, este desmembramento pode continuar indefinidamente:

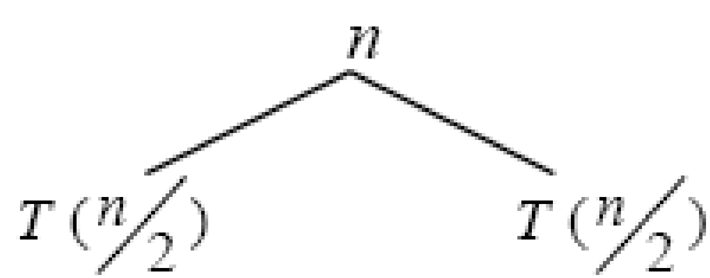
$$T(n) = 2 \cdot \left(2 \cdot \left(2T\left(n/8\right) + n/4\right) + n/2\right) + n$$
$$T(n) = 2 \cdot \left(2 \cdot \left(2 \cdot \left(2T\left(n/16\right) + n/8\right) + n/4\right) + n/2\right) + n$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

Árvore de Recursão

- Obviamente esta forma de desenho da recorrência ou da sua história é muito confusa, pelo menos a primeira vista. Desta forma, este método de solução faz a descrição da relação na forma de uma árvore. Veja:

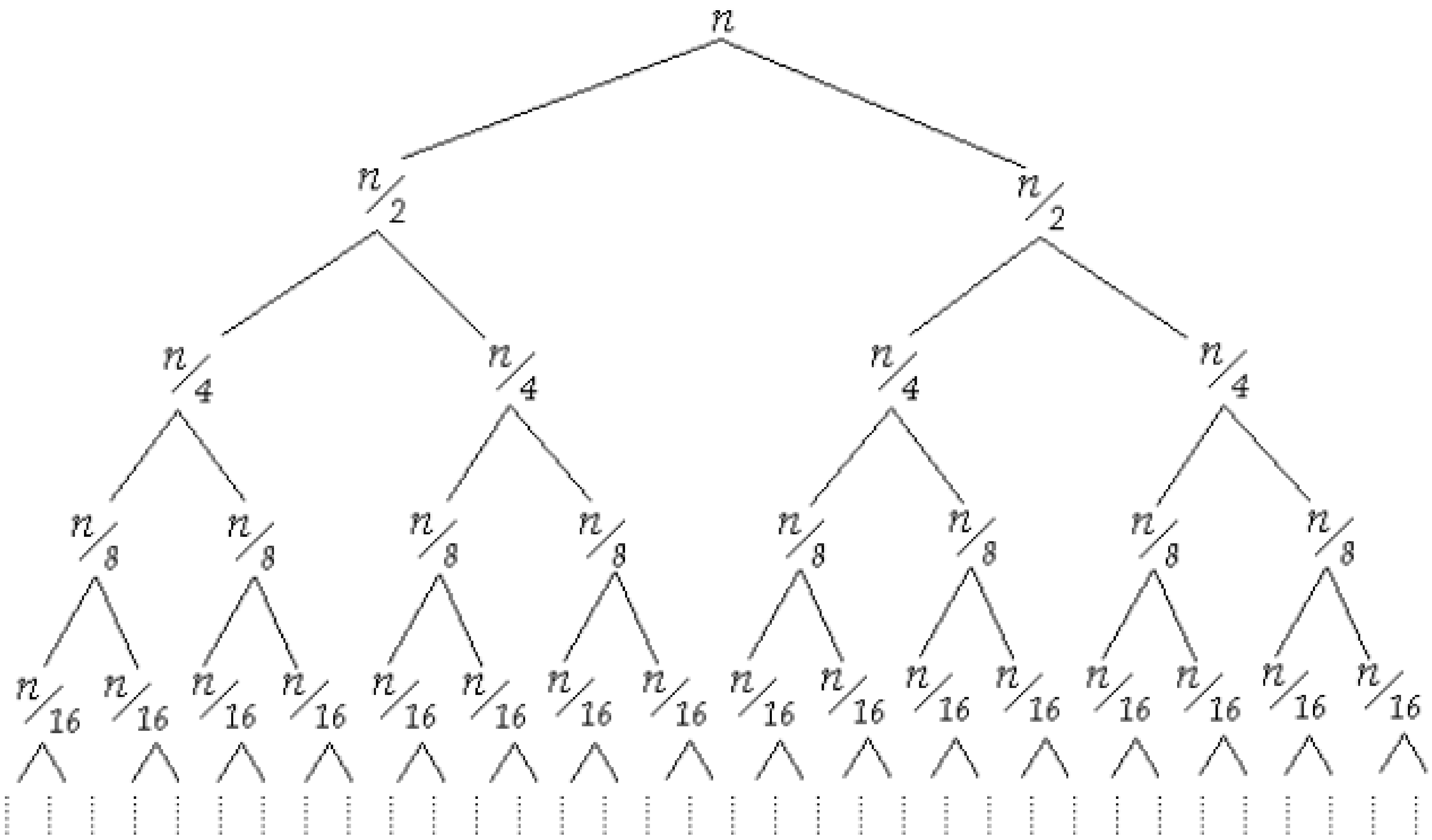
$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \rightarrow$$



Árvore de Recursão

- Nesta árvore nós temos o trabalho adicional como elemento centralizador.
- A partir deste elemento ocorrem tantas ramificações quantas as chamadas recursivas da recorrência.
- Neste caso serão duas delas (**2**T(n/2)), sendo cada uma delas de T(n/2).
- Agora, se ampliarmos sempre este raciocínio encontrará, para este caso, a árvore apresentada na figura a seguir.
- Esta forma de desenho da recorrência ainda não apresenta a solução da mesma, mas fornece importantes detalhes sobre o seu comportamento.

Árvore de Recursão



Árvore de Recursão

- Em primeiro lugar, perceba que necessitamos somar todo o conteúdo desta árvore para sabermos a solução da recorrência.

- Neste ponto esbarramos no mesmo problema inicial com a relação original: o final da execução.

- Aqui, entretanto, está mais fácil de se chegar a uma dedução sobre o tema.

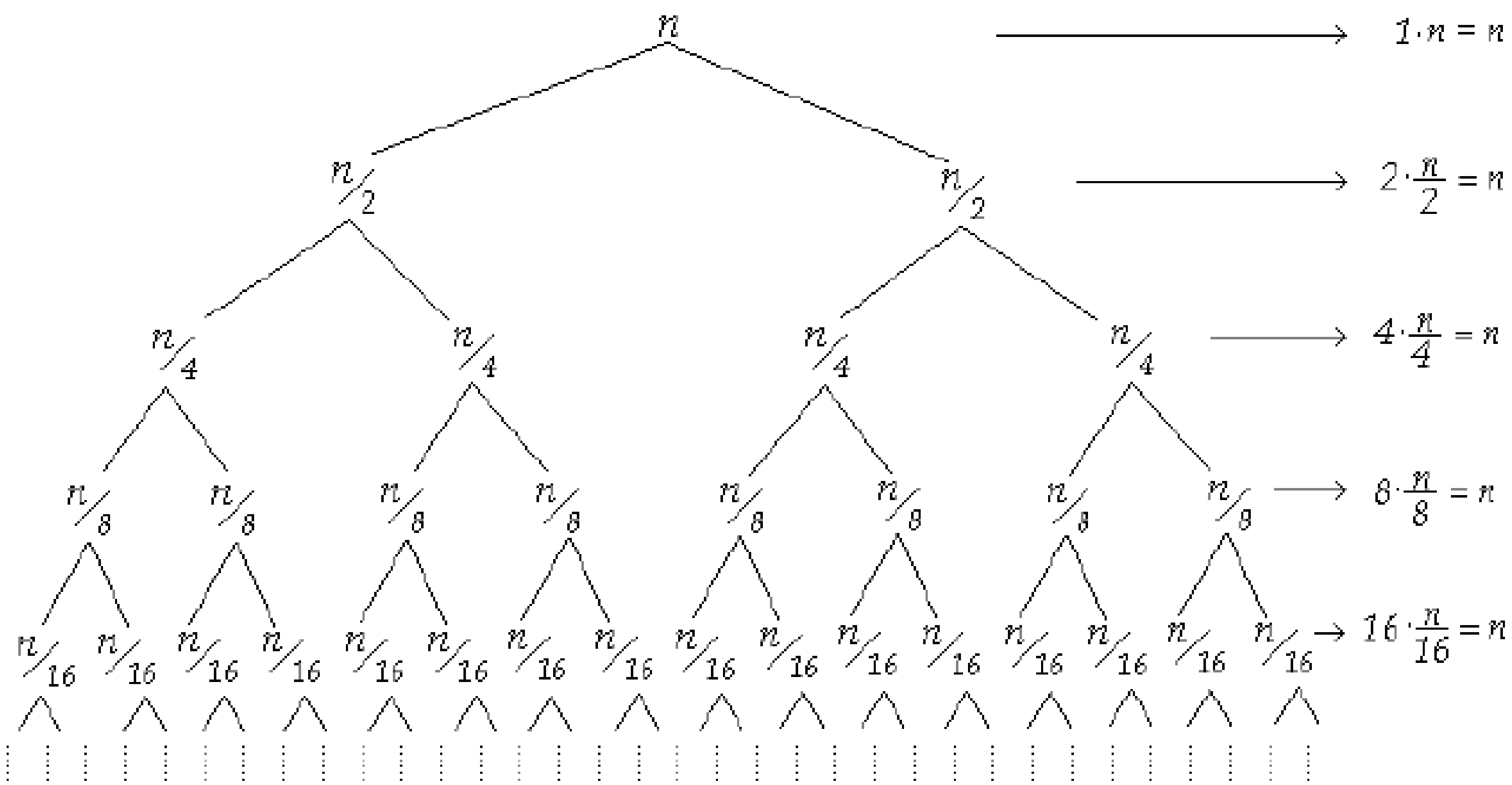
- Antes disso, vamos somar cada um dos níveis desta árvore.
  - O primeiro nível, por ter somente n, tem este como soma.
  - O segundo, por sua vez, também soma n.
  - O mesmo acontece com o terceiro nível.

Árvore de Recursão

- A árvore completa (próx. slide) permite deduzir que todos os níveis da árvore somam um trabalho equivalente a *n*.
- Sabemos agora que a mesma recorrência tomada como exemplo pode ser escrita como:

$$T(n) = n + n + n + n + \cdots + n = \sum_{i=1}^{X} n$$

Árvore de Recursão



Árvore de Recursão

- Onde X é a quantidade de níveis da árvore.
- Note que, em cada um dos níveis da relação estamos dividindo o elemento do nível sempre por dois.
- Segundo a definição desta relação, a divisão deve parar quando o tamanho da entrada chegar até 1 - que é o caso óbvio
- Note que, se o final for um nível acima, com T(2) = x, basta diminuir um do valor encontrado e assim por diante.

Árvore de Recursão

- Assim, a nossa pergunta fica resumida a seguinte:
  - quantas divisões serão necessárias em *n*, pelo valor 2, até que ele chega a 1?
  - Para encontrar a resposta, vamos supor, sem perda de correção, que *n* é potência de 2.
  - Assim, esta divisão é exata e pode ser representada de outra forma: quantas potências de 2 serão necessárias para chegar até n? Ou seja,

$$2^x = n?$$

Árvore de Recursão

- Segundo o estudo de logaritmos sabemos que a resposta desta questão é $x = \log_2 n$.

- Este mesmo raciocínio funciona ao expandirmos o valor de *n* para qualquer número que não seja, necessariamente, uma potência de 2.

- A única diferença nestes casos está no fato do número de divisões não ser inteiro, o que não apresenta nenhum problema

Árvore de Recursão

- De forma mais simplista possível, podemos formular uma pequena regra para este tipo de caso: sempre que dividirmos sucessivamente um valor qualquer por 2, a divisão chegará até 1 após $\log_2 n$ execuções.

- Para ser mais genérico ainda, se a divisão for por outro valor qualquer, além de 2, podemos utilizar o mesmo procedimento, apenas a base do logaritmo irá mudar.

- Finalmente, se ocorreram $\log_2 n$ divisões, deduzimos que a árvore que estamos analisando tem $\log_2 n$ níveis, somando *n* cada um deles.

Árvore de Recursão

- Assim, podemos reformular a totalização como:

$$T(n) = n + n + n + n + \cdots + n = \sum_{i=1}^{\log_2 n} n = n \log_2 n$$

Árvore de Recursão

- Para encerrarmos este método, já é possível caracterizar a forma de procedimento adotada:
  a) Em primeiro lugar, desenha-se a recorrência em forma de árvore. Isto faz com que a visualização de todo o comportamento da mesma fique facilitada;
  b) Após, somam-se os níveis desta árvore. Com isto buscamos deduzir o comportamento de todos os níveis existentes. Obviamente o desenho feito deve conter níveis na árvore o suficiente para permitir uma dedução correta;
  c) Uma vez conhecido o comportamento de cada nível, basta saber quantos eles são para conseguir totalizar a estrutura;
  d) Lembre-se que a resposta somente será válida se o somatório dos níveis for eliminado. Deve-se apresentar um valor total e final. Para isso é importante lembrar e recorrer a conceitos básicos da matemática.

Método Mestre

- Este é o método mais fácil e direto de todos.
- Em geral, recomenda-se que sempre se busque a solução com ele. Se isto não for possível, tenta-se então utilizar a árvore de recursão.
- O método mestre depende do teorema a seguir.

Método Mestre

- *"Sejam a* ***>=*** *1 e b > 1 constantes, seja f(n) uma função e seja T(n) definida sobre os inteiros não negativos pela recorrência"*

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

Método Mestre

- A recorrência acima descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas, cada um do tamanho n/b, onde a e b são constantes positivas.
- Então, T(n) pode ser limitado assintoticamente como a seguir.

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 $T(n) = \Theta(n^{\log_b a} \lg n)$.

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 para todo n suficientemente grande, então $T(n) = \Theta(f(n))$. ■

FONTE: Thomas Cormen, Charles Leiserson and Ronald Rivest. *Introduction to Algorithms*. Second Edition. McGraw-Hill, 2007.

Restrições

- É importante perceber que os três casos não abrangem todas as possibilidades para f(n).

- Existe uma lacuna entre os casos 1 e 2 quando f(n) é menor que $n^{\log_b a}$, mas não polinomialmente menor.

- De modo semelhante, há uma lacuna entre os casos 2 e 3 quando f(n) é maior que $n^{\log_b a}$, mas não polinomialmente maior.

- Se a função f(n) recair em uma dessas lacunas, ou se a condição de regularidade no caso 3 deixar de ser válida, o método mestre não poderá ser usado para resolver a recorrência.

Método Mestre

- **Outra forma:**
- Se f (n) ∈ Θ (n^d), onde d >= 0, então:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Fonte: [LEVITIN, 2012]

Exemplo 1

- **$T(n) = 4.T(n/2) + n$**

Analisando a recorrência, pode-se identificar que $a = 4$, $b = 2$ e $f(n) = n$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_b a} = n^{\log_2 4} = n^2$.

Como $f(n) = n = O\left(n^{\log_b a-\varepsilon}\right) = O(n^{2-\varepsilon})$ para $\varepsilon = 1$, pode-se aplicar o caso 1 do teorema master. Conclui-se então que a solução da recorrência é $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^2)$.

Exemplo 2

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

Analisando a recorrência, pode-se identificar que $a = 1$, $b = \frac{3}{2}$ e $f(n) = 1$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$.

Como $f(n) = 1 = \Theta\left(n^{\log_b a}\right) = \Theta(1)$, pode-se aplicar o caso 2 do teorema master. Conclui-se então que a solução da recorrência é $T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right) = \Theta(1 \cdot log\, n) = \Theta(log\, n)$.

Exemplo 3

- **T(n) = 9.T(N/3) + n³**

Analisando a recorrência, pode-se identificar que $a = 9$, $b = 3$ e $f(n) = n^3$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_3 9} = n^2$.

Como $f(n) = n^3 = \Omega\left(n^{\log_b a+\varepsilon}\right) = \Omega(n^{2+\varepsilon})$, para $\varepsilon = 1$, pode-se aplicar o caso 3 do teorema master desde que se prove também a condição $a \cdot f(n/b) \leq c \cdot f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande. A prova é mostrada a seguir.

$$a \cdot f(n/b) \leq c \cdot f(n)$$
$$9 \cdot f(n/3) \leq c \cdot f(n), \rightarrow f(n) = n^3$$
$$9 \cdot \frac{n^3}{27} \leq c \cdot n^3$$
$$\frac{1}{3} n^3 \leq c \cdot n^3, c = \frac{1}{3}, n \geq 1.$$

Portanto, de acordo com o caso 3, a solução da recorrência é $T(n) = \Theta(f(n)) = \Theta(n^3)$.

Questão Poscomp 2019

QUESTÃO 24 – Um procedimento recursivo é aquele que contém em sua descrição:

A) Uma prova de indução matemática.
B) Duas ou mais chamadas a procedimentos externos.
C) Uma ou mais chamadas a si mesmo.
D) Somente chamadas externas.
E) Uma ou mais chamadas a procedimentos internos.

[PosComp2019]

Considere os seguintes algoritmos recursivos que resolvem o mesmo problema em uma entrada de tamanho n

Algoritmo 1: Divide o problema em 3 partes de tamanho n/4 cada e gasta um tempo adicional O(1) por chamada.
Algoritmo 2: Divide o problema em 3 partes de tamanho n/2 cada e gasta um tempo adicional O(n^2) por chamada.
Algoritmo 3: Divide o problema em 3 partes de tamanho n/3 cada e gasta um tempo adicional de O(n) por chamada.

A complexidade dos algoritmos 1, 2 e 3 é, respectivamente:

A) $\Theta(n^{log_4 3}), \Theta(n^2), \Theta(n \log n)$

B) $\Theta\left(\frac{n}{4}\right), \Theta\left(\frac{n}{2}\right), \Theta\left(\frac{n}{3}\right)$

C) $\Theta(1), \Theta(n^2), \Theta(n)$

D) $\Theta(n^4), \Theta(n^2), \Theta(n^3)$

E) $\Theta(n^{log_4 3}), \Theta(n^{log_2 3}), \Theta(n^{log_3 3})$

$a < b^d \rightarrow \Theta(n^d)$

$a = b^d \rightarrow \Theta(n^d \log n)$

$a > b^d \rightarrow \Theta(n^{\log_b a})$

Bibliografia

- Paulo Azeredo. *Métodos de Classificação de Dados e Análise de suas Complexidades*. Campus, 1996.
- Thomas Cormen, Charles Leiserson and Ronald Rivest. *Introduction to Algorithms*. Second Edition. McGraw-Hill, 2007.
- Alfred Aho and Jeffrey Ullman. *Foundations of Computer Science*. Computer Science Press, 1992.
- Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*, 2nd edition. Addison-Wesley, 1988.
- Donald Knuth. *The Art of Computer Programming: Fundamentals Algorithms*. Addison-Wesley, 1973.
- Udi Mamber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.
- Nivio Zizanni. *Projeto de Algoritmos com implementações em Java e C++*. Tomson, 2006.