SCC-201 - Capítulo 3
Análise de Algoritmos - Parte 2

João Luís Garcia Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2016

Sumário

Sumário

Exercício

- Analise a sub-rotina recursiva abaixo:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   se n = 1
5     então aux ← 1
6     senão aux ← n * fatorial(n − 1);
7   fatorial ← aux;
8 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Se a recursão é um "disfarce" da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal,
  - O exemplo anterior é obviamente $\mathcal{O}(n)$.
- Eliminando a recursão:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   aux ← 1
5   enquanto n > 1 faça
6     aux ← aux * n;
7     n ← n − 1;
8   fatorial ← aux;
9 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Em muitos casos (incluindo casos em que a recursividade é bem empregada), é difícil transformá-la em repetição:
    - Nesses casos, para fazer a análise do algoritmo, pode ser necessário usar a análise de recorrência.
    - **Recorrência**: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.
    - Caso típico: algoritmos de divisão-e-conquista, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

Regras para o cálculo

- Exemplo de uso de recorrência:
  - Números de Fibonacci:
    - 0,1,1,2,3,5,8,13...
    - $fib(0) = 0$, $fib(1) = 1$, $fib(i) = fib(i-1) + fib(i-2)$.
  - A rotina:

```
1 sub-rotina fib(n: numérico)
2 início
3 declare aux numérico;
4 se n <= 1
5   então aux ← n
6   senão aux ← fib(n − 1) + fib(n − 2);
7 fib ← aux;
8 fim
```

Regras para o cálculo

- Seja $T(n)$ o tempo de execução da função:
  - **Caso 1**: Se $n = 0$ ou $n = 1$, o tempo de execução é constante, que é o tempo de testar o valor de n no comando *se*, mais atribuir o valor 1 à variável *aux*, mais atribuir o valor de *aux* ao nome da função; ou seja, $T(0) = T(1) = 3$.
  - **Caso 2**: Se $n > 2$, o tempo consiste em testar o valor de n no comando *se*, mais o trabalho a ser executado no *senão* (que é uma soma, uma atribuição e 2 chamadas recursivas), mais a atribuição de *aux* ao nome da função; ou seja, a recorrência $T(n) = T(n-1) + T(n-2) + 4$, para $n > 2$.

Regras para o cálculo

- Muitas vezes, a recorrência pode ser resolvida com base na prática e experiência do analista,
- Alguns métodos para resolver recorrências:
  - Método da substituição,
  - Método mestre,
  - Método da árvore de recursão.

Sumário

Resolução de recorrências

- Método da substituição:
  - Supõe-se (aleatoriamente ou com base na experiência) um limite superior para a função e verifica-se se ela não extrapola este limite:
    - Uso de indução matemática.
  - O nome do método vem da “substituição” da resposta adequada pelo palpite,
  - Pode-se “apertar” o palpite para achar funções mais exatas.

Resolução de recorrências

- Método mestre:
  - Fornece limites para recorrências da forma $T(n) = aT(\frac{n}{b}) + f(n)$, em que $a \geq 1$, $b > 1$ e $f(n)$ é uma função dada,
  - Envolve a memorização de alguns casos básicos que podem ser aplicados para muitas recorrências simples.

Resolução de recorrências

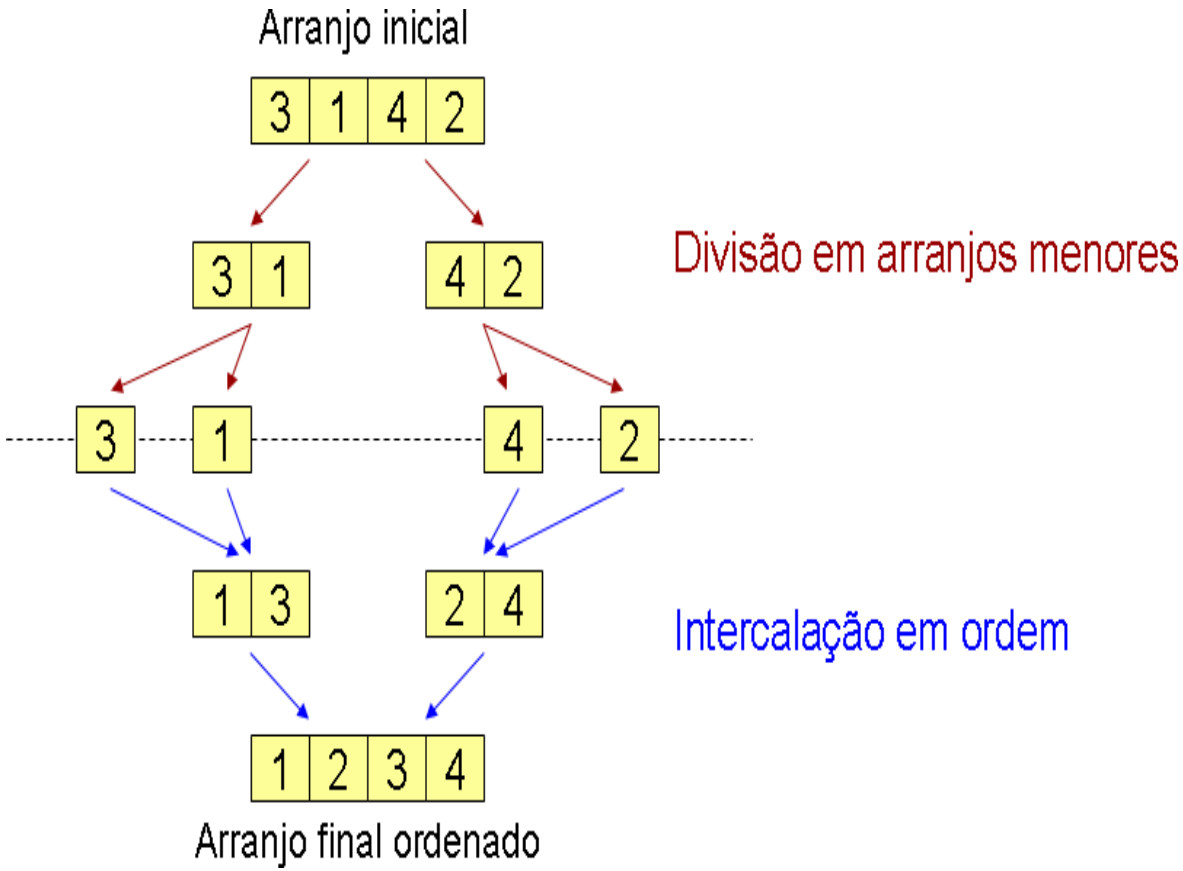
- Método da árvore de recursão:
  - Traça-se uma árvore que, nível a nível, representa as recursões sendo chamadas,
  - Em seguida, em cada nível/nó da árvore, são acumulados os tempos necessários para o processamento:
    - No final, tem-se a estimativa de tempo do problema.
  - Este método pode ser utilizado para se fazer uma suposição mais informada no método da substituição.

Resolução de recorrências

- Método da árvore de recursão:
  - **Exemplo**: algoritmo de ordenação de arranjos por intercalação:
    - Passo 1: divide-se um arranjo não ordenado em dois subarranjos,
    - Passo 2: se os subarranjos não são unitários, cada subarranjo é submetido ao passo 1 anterior; caso contrário, eles são ordenados por intercalação dos elementos e isso é propagado para os subarranjos anteriores.

Ordenação por intercalação

- Exemplo com arranjo de 4 elementos:



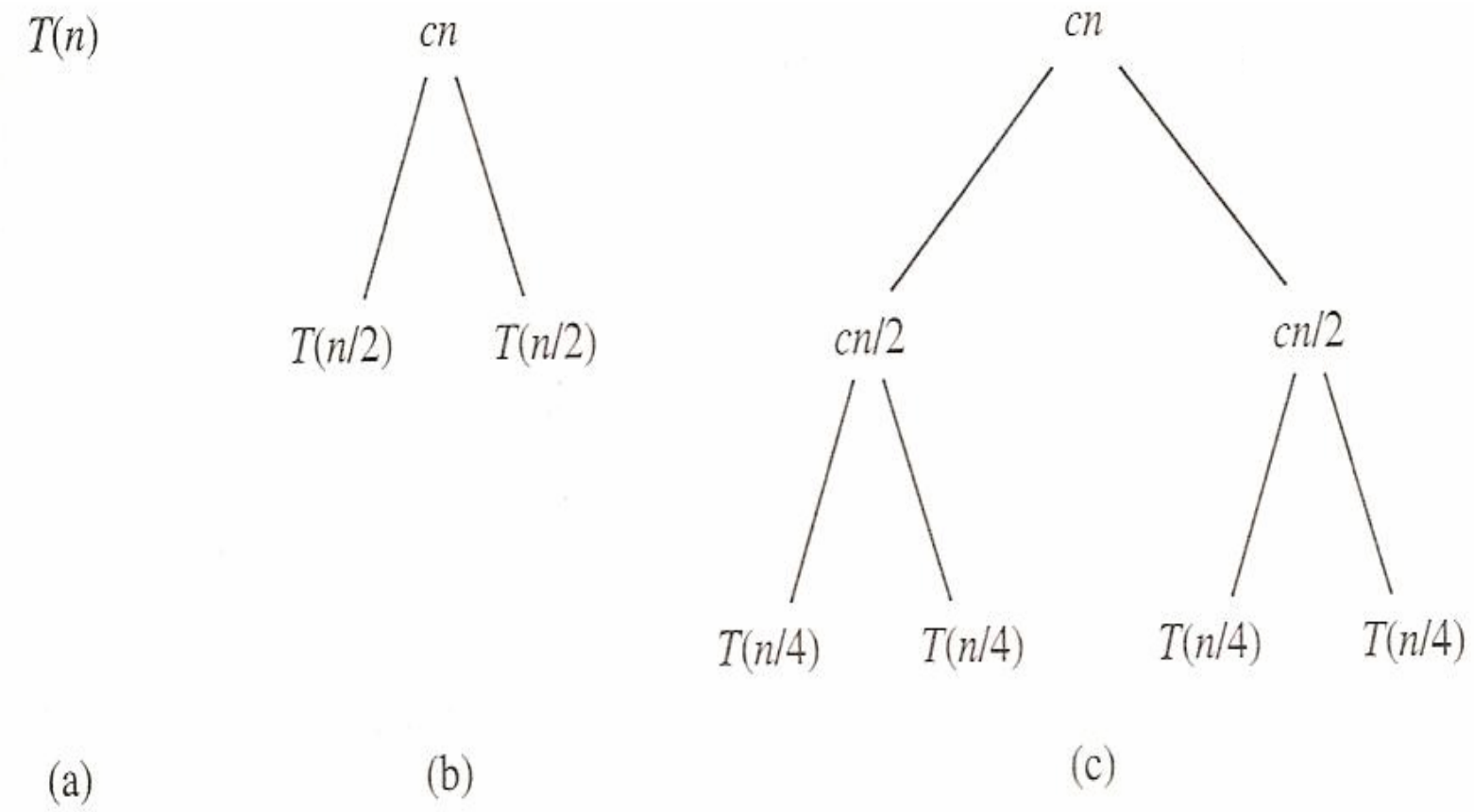
- Implemente a(s) sub-rotina(s) e calcule sua complexidade.

Resolução de recorrências

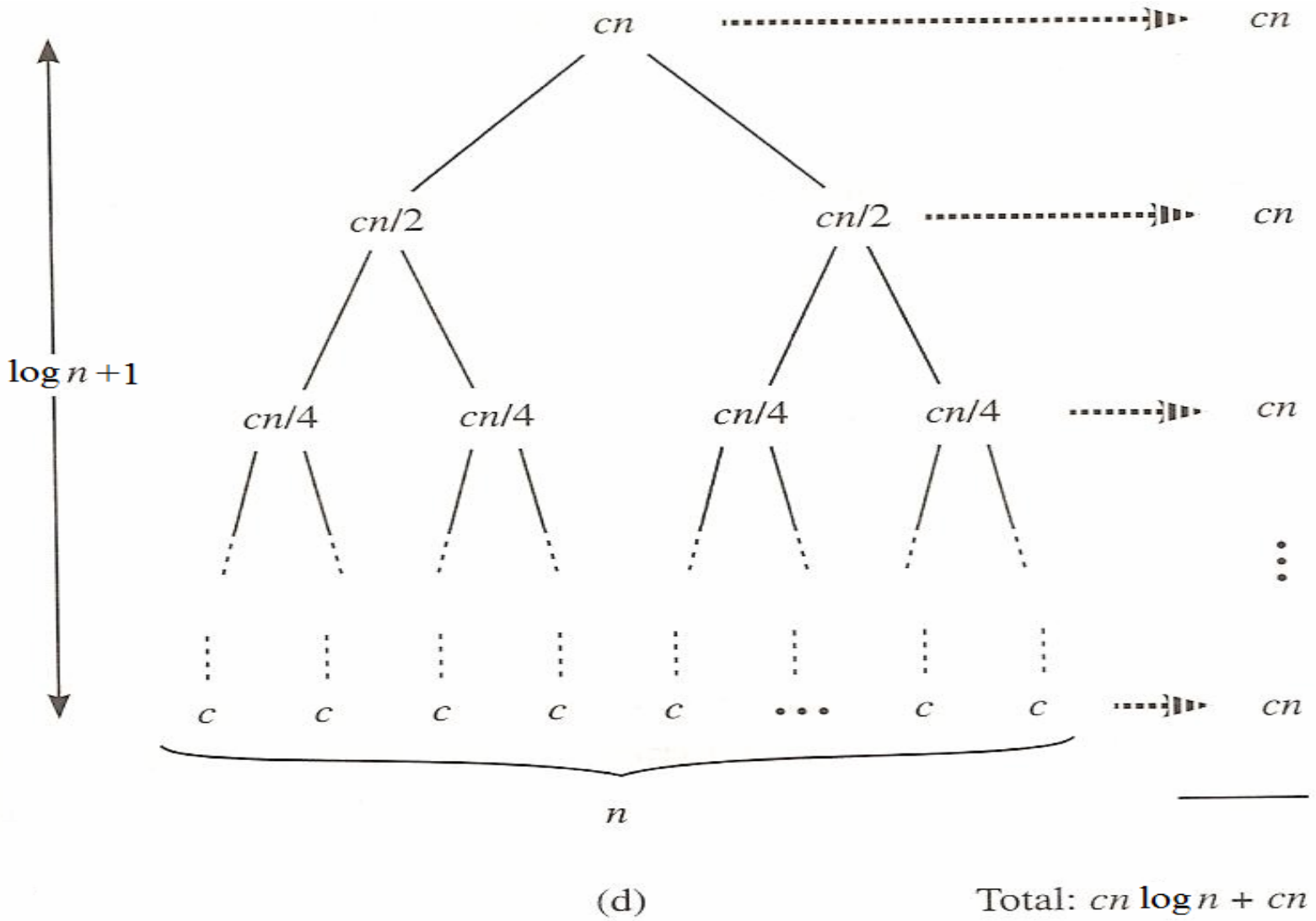
- Método da árvore de recursão:
  - Considere o tempo do algoritmo (que envolve recorrência):

$$T(n) = c, \text{ se } n = 1$$
$$T(n) = 2T(\tfrac{n}{2}) + cn, \text{ se } n > 1$$

Resolução de recorrências



Resolução de recorrências



Resolução de recorrências

- Tem-se que:
  - Na parte (a), há $T(n)$ ainda não expandido,
  - Na parte (b), $T(n)$ foi dividido em árvores equivalentes representando a recorrência com custos divididos ($T(\frac{n}{2})$ cada uma), sendo cn o custo no nível superior da recursão (fora da recursão e, portanto, associado ao nó raiz),
  - ...
  - No fim, nota-se que o tamanho da árvore corresponde a $(\log\ n) + 1$, o qual multiplica os valores obtidos em cada nível da árvore, os quais, nesse caso, são iguais:
    - Como resultado, tem-se $cn\ \log\ n + cn$, ou seja, $\mathcal{O}(n\ \log\ n)$.

Resolução de recorrências

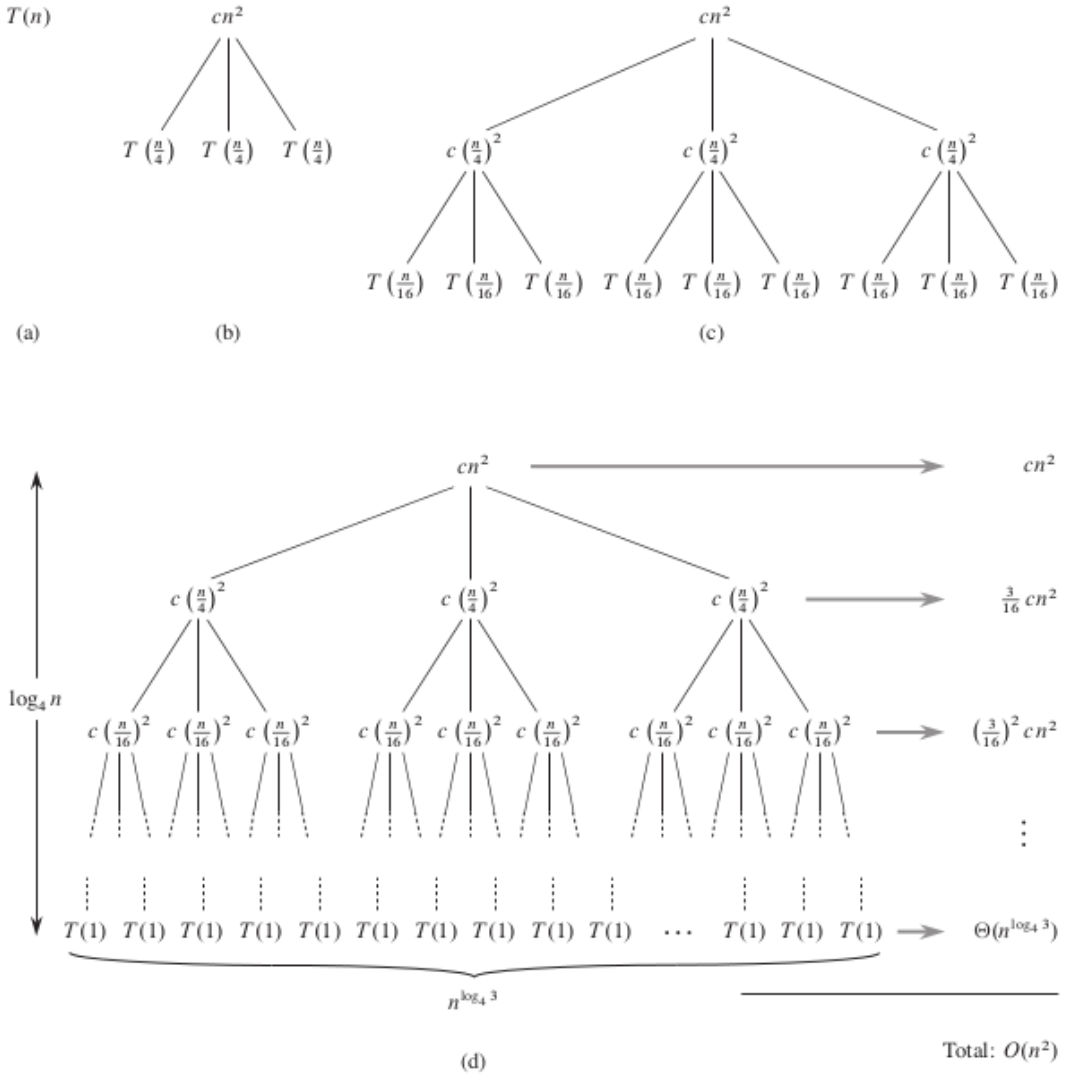
- Alguns dizem que a expressão correta é "$f(n)$ é $\mathcal{O}(g(n))$":
  - Seria considerado redundante e inadequado dizer "$f(n) \leq \mathcal{O}(g(n))$" ou (ainda pior) "$f(n) = \mathcal{O}(g(n))$",
  - Não é incorreto (embora não seja usual) dizer "$f(n) \in \mathcal{O}(g(n))$", já que o operador *Big-oh* representa todo um conjunto de funções.

Sumário

Exemplo 1

- O próximo slide [2] mostra a árvore de recursão para $T(n) = 3T(n/4) + cn^2$:
  - Assume-se que n é uma potência de 4 para que os tamanhos dos sub-problemas sejam inteiros,
  - Parte (a) mostra $T(n)$, que é expandido na parte (b) em uma árvore equivalente representando a recorrência,
  - O termo cn^2 na raiz representa o custo no nível superior da recursão e as três sub-árvores da raiz representam os custos dos sub-problemas de tamanho $n/4$,
  - A parte (c) mostra esse processo um passo a frente expandindo cada nó com custo $T(n/4)$ a partir da parte (b),
  - O custo de cada um dos três sucessores da raiz é $c(n/4)^2$,
  - Continua-se expandindo cada nó da árvore, quebrando-o em suas partes constituintes, como determinado pela recorrência.

Exemplo 1


T(n)
cn²
T(n/4) T(n/4) T(n/4)
c(n/4)²
T(n/16)
(a)
(b)
(c)
log₄ n
cn²
3/16 cn²
(3/16)² cn²
T(1)
Θ(n^log₄ 3)
n^log₄ 3
(d)
Total: O(n²)

Exemplo 1

- Como os tamanhos dos sub-problemas descrescem por um fator de 4 cada vez que descemos um nível, deve-se alcançar uma condição limite.
- Quão longe da raiz alcançaremos?
- O tamanho do sub-problema para um nó na profundidade i é $n/4^i$.
- Portanto, o tamanho do sub-problema alcança $n = 1$ quando $n/4^i = 1$, ou equivalentemente, quando $i = \log_4 n$.
- Ou seja, a árvore tem $\log_4 n + 1$ níveis (em profundidades 0, 1, 2, ..., $\log_4 n$).

Exemplo 1

- Depois, determina-se o custo em cada nível da árvore.
- Cada nível tem três vezes mais nós que o nível acima, e portanto o número de nós na profundidade i é 3^i.
- Como os tamanhos dos sub-problemas reduzem por um fator de 4 para cada nível a partir da raiz, cada nó na profundidade i, para $i = 0, 1, 2, ..., \log_4 n - 1$, tem um custo de $c(n/4^i)^2$.
- Multiplicando, o custo total de todos os nós na profundidade i é $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- O nível do fundo, na profundidade $\log_4 n$, tem $3^{\log_4 n} = n^{\log_4 3}$ nós, cada um com custo $T(1)$, para um custo total de $n^{\log_4 3} T(1)$, que é $\Theta(n^{\log_4 3})$, assumindo $T(1)$ como uma constante.

Exemplo 1

- Custo da árvore inteira:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

pois

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

Exemplo 1

Quando a soma é infinita e $|x| < 1$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Logo

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

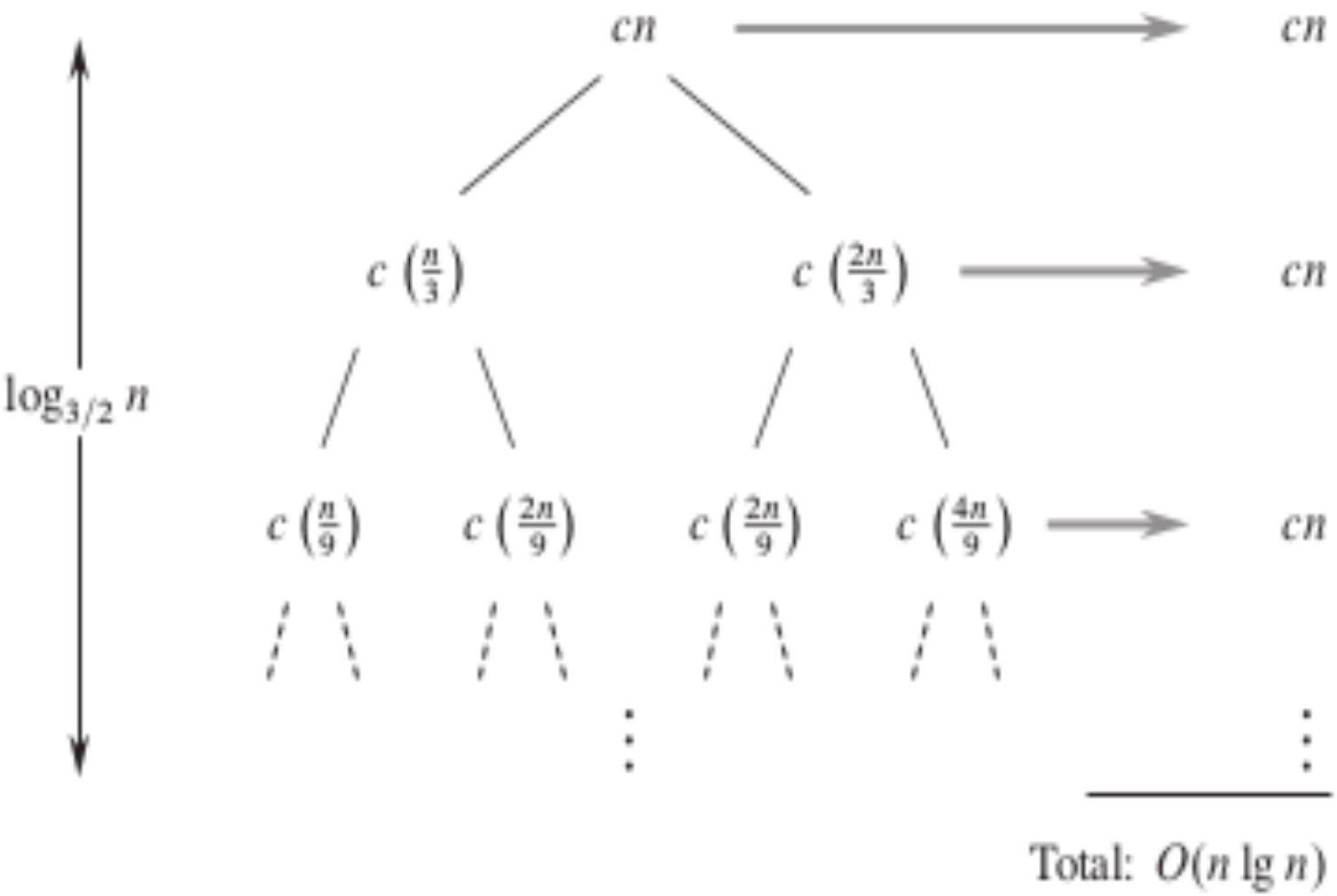
$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = \mathcal{O}(n^2).$$

pois $\log_4 3 = 0.79$.

Exemplo 2

- Outro exemplo [2]: árvore de recursão para $T(n) = T(n/3) + T(2n/3) + \mathcal{O}(n)$.
- c representa o fator constante no termo $\mathcal{O}(n)$.
- Quando se adiciona os valores ao longo dos níveis da árvore de recursão, chega-se ao valor de cn para cada nível.
- O caminho mais longo da raiz a uma folha é $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$.
- Como $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, a altura da árvore é $\log_{3/2} n$.

Exemplo 2



Exemplo 2

- Intuitivamente, espera-se que a solução para a recorrência seja no máximo o número de níveis vezes o custo de cada nível, ou $\mathcal{O}(cn \log_{3/2} n) = \mathcal{O}(n \log n)$.
- A figura mostra apenas os níveis superiores da árvore de recursão e no entanto, nem todo nível contribui com um custo de cn.
- Considere o custo das folhas.
- Se a árvore for uma árvore binária completa de altura $\log_{3/2} n$, então haveria $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas.
- Como o custo de cada folha é uma constante, o custo total de todas as folhas seria $\Theta(n^{\log_{3/2} 2}) = \omega(n \log n)$[1], já que $\log_{3/2} 2$ é uma constante maior que 1.

[1] Diz-se que $f(n)$ é **little ômega** $g(n)$ ou que $f(n) = \omega(g(n))$, se e somente se $f(n) = \Omega(g(n))$ e $f(n) \neq \Theta(g(n))$. A taxa de crescimento de $f(n)$ é maior do que a taxa de $g(n)$.

Exemplo 2

- Esta árvore de recursão não é uma árvore binária completa, portanto ela tem menos de $n^{\log_{3/2} 2}$ folhas.
- Consequentemente, níveis próximos do fundo contribuem menos que cn para o custo total.
- Como estamos preocupados apenas em adivinhar (método da substituição), não precisamos nos preocupar com o custo exato.
- Portanto, nosso "chute" para o limite superior será $\mathcal{O}(n \log n)$.
- Usaremos então o método da substituição para verificar que $\mathcal{O}(n \log n)$ é um limite superior para a solução da recorrência.
- Mostramos que $T(n) \leq d\ n\ \log n$, onde d é uma constante positiva.

Exemplo 2

- Tem-se

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn \\
&= (d(n/3)\log n - d(n/3)\log 3) \\
&\quad +(d(2n/3)\log n - d(2n/3)\log(3/2)) + cn \\
&= d\ n\ \log n - d((n/3)\log 3 + (2n/3)\log(3/2)) + cn \\
&= d\ n\ \log n - d((n/3)\log 3 + (2n/3)\log 3 - (2n/3)\log 2) + cn \\
&= d\ n\ \log n - d\ n(\log 3 - 2/3) + cn \\
&\leq d\ n\ \log n
\end{aligned}
$$

- desde que $d \geq c/(\log 3 - (2/3))$.

Problema das 8 Rainhas

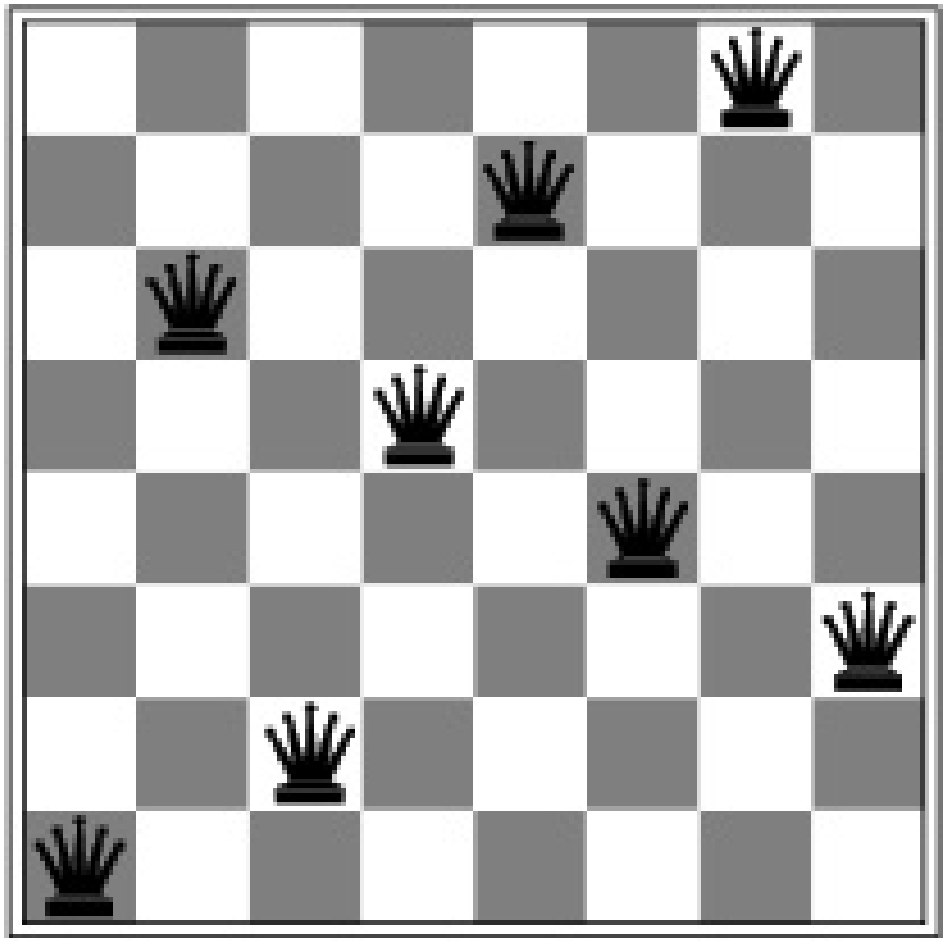
Exercício proposto

Resolver, através da recursão, o **problema das 8 rainhas**. Fazer a análise do algoritmo em termos de *big-Oh* (equação de recorrência). O problema das 8 rainhas consiste em colocar num tabuleiro de xadrez (matriz 8×8), 8 rainhas, uma em cada linha, de tal forma que uma rainha não "coma" outra. Lembre-se de que a rainha é a peça do xadrez que se movimenta qualquer número de casas na vertical, na horizontal e nas diagonais.

Obs.: O problema tem 92 soluções distintas.

Problema das 8 Rainhas

- Veja uma “quase” solução:



Problema das 8 Rainhas

92 Soluções:

1 5 8 6 3 7 2 4	1 6 8 3 7 4 2 5	1 7 4 6 8 2 5 3	1 7 5 8 2 4 6 3	2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4	2 5 7 4 1 8 6 3	2 6 1 7 4 8 3 5	2 6 8 3 1 4 7 5	2 7 3 6 8 5 1 4
2 7 5 8 1 4 6 3	2 8 6 1 3 5 7 4	3 1 7 5 8 2 4 6	3 5 2 8 1 7 4 6	3 5 2 8 6 4 7 1
3 5 7 1 4 2 8 6	3 5 8 4 1 7 2 6	3 6 2 5 8 1 7 4	3 6 2 7 1 4 8 5	3 6 2 7 5 1 8 4
3 6 4 1 8 5 7 2	3 6 4 2 8 5 7 1	3 6 8 1 4 7 5 2	3 6 8 1 5 7 2 4	3 6 8 2 4 1 7 5
3 7 2 8 5 1 4 6	3 7 2 8 6 4 1 5	3 8 4 7 1 6 2 5	4 1 5 8 2 7 3 6	4 1 5 8 6 3 7 2
4 2 5 8 6 1 3 7	4 2 7 3 6 8 1 5	4 2 7 3 6 8 5 1	4 2 7 5 1 8 6 3	4 2 8 5 7 1 3 6
4 2 8 6 1 3 5 7	4 6 1 5 2 8 3 7	4 6 8 2 7 1 3 5	4 6 8 3 1 7 5 2	4 7 1 8 5 2 6 3
4 7 3 8 2 5 1 6	4 7 5 2 6 1 3 8	4 7 5 3 1 6 8 2	4 8 1 3 6 2 7 5	4 8 1 5 7 2 6 3
4 8 5 3 1 7 2 6	5 1 4 6 8 2 7 3	5 1 8 4 2 7 3 6	5 1 8 6 3 7 2 4	5 2 4 6 8 3 1 7
5 2 4 7 3 8 6 1	5 2 6 1 7 4 8 3	5 2 8 1 4 7 3 6	5 3 1 6 8 2 4 7	5 3 1 7 2 8 6 4
5 3 8 4 7 1 6 2	5 7 1 3 8 6 4 2	5 7 1 4 2 8 6 3	5 7 2 4 8 1 3 6	5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4	5 7 4 1 3 8 6 2	5 8 4 1 3 6 2 7	5 8 4 1 7 2 6 3	6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4	6 2 7 1 4 8 5 3	6 3 1 7 5 8 2 4	6 3 1 8 4 2 7 5	6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8	6 3 5 8 1 4 2 7	6 3 7 2 4 8 1 5	6 3 7 2 8 5 1 4	6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3	6 4 2 8 5 7 1 3	6 4 7 1 3 5 2 8	6 4 7 1 8 2 5 3	6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5	7 2 4 1 8 5 3 6	7 2 6 3 1 4 8 5	7 3 1 6 8 5 2 4	7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6	7 4 2 8 6 1 3 5	7 5 3 1 6 8 2 4	8 2 4 1 7 5 3 6	8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4	8 4 1 3 6 2 7 5			

Sumário

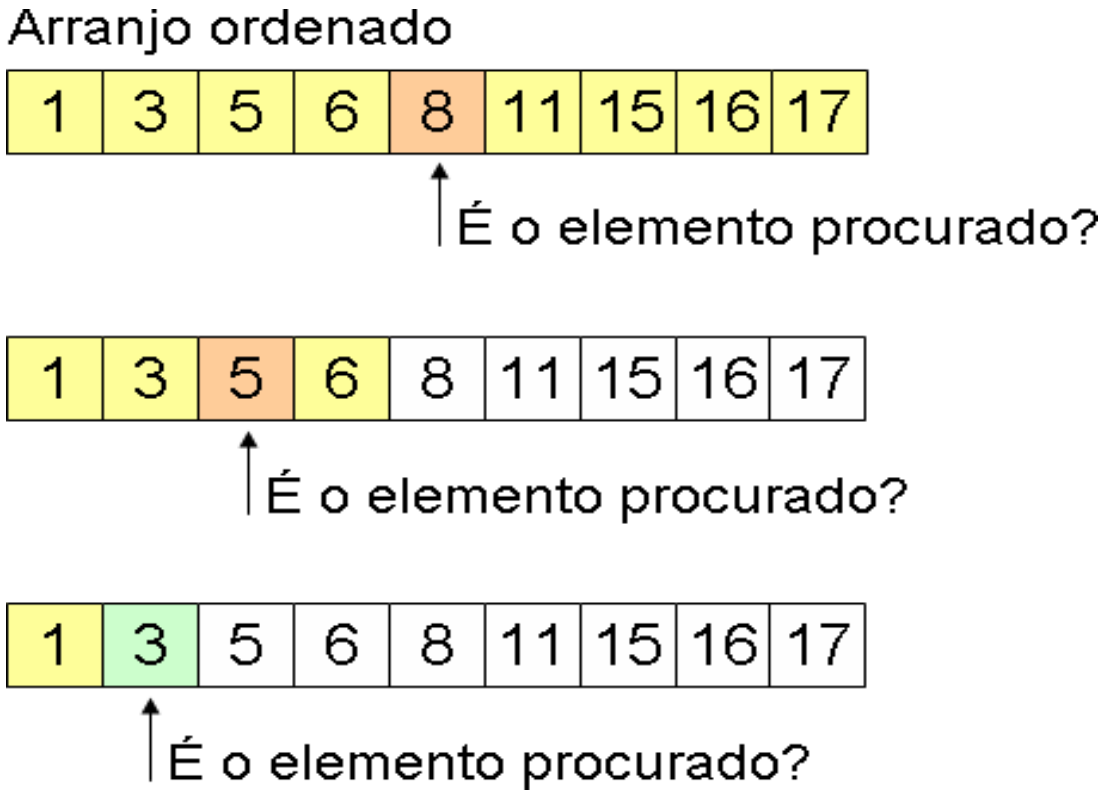
Cuidado!

- A análise assintótica é uma ferramenta fundamental ao projeto, análise ou escolha de um algoritmo específico para uma dada aplicação,
- No entanto, deve-se ter sempre em mente que essa análise "esconde" fatores assintoticamente irrelevantes, mas que em alguns casos podem ser relevantes na prática, particularmente se o problema de interesse se limitar a entradas (relativamente) pequenas:
  - Por exemplo, um algoritmo com tempo de execução da ordem de $10^{100}n$ é $\mathcal{O}(n)$, assintoticamente melhor do que outro com tempo $10\ n\ \log\ n$, o que nos faria, em princípio, preferir o primeiro,
  - No entanto, 10^{100} é o número estimado por alguns astrônomos como um limite superior para a quantidade de átomos existente no universo observável!

Sumário

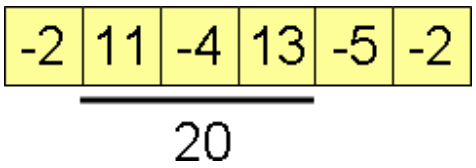
Análise de algoritmos recursivos

- Muitas vezes, temos que resolver recorrências:
  - **Exemplo**: pesquisa binária pelo número 3:



Exercícios propostos

1. Implemente o algoritmo da busca binária em um arranjo ordenado, teste e analise o algoritmo,
2. Faça um algoritmo para resolver o problema da maior soma de subseqüência em um arranjo e analise-o:



3. Implemente o algoritmo de Euclides para calcular o máximo divisor comum para 2 números, e faça a análise de recorrência do mesmo,
4. Escreva e analise um algoritmo recursivo para calcular x^n.

Sumário

Estratégia

- Dada uma função para computar n entradas, a estratégia *divisão-e-conquista* separa as entradas em k subconjuntos distintos, $1 < k \leq n$, levando a k subproblemas,
- Estes subproblemas devem ser resolvidos, e então um método deve ser encontrado para combinar subsoluções em uma solução do todo,
- Se os subproblemas ainda forem muito grandes, então a estratégia divisão-e-conquista pode ser reaplicada,
- Os subproblemas resultantes são do *mesmo* tipo do problema original.

Estratégia

- A reaplicação do princípio é naturalmente expressa por um algoritmo recursivo,
- Subproblemas do mesmo tipo cada vez menores são gerados até que finalmente subproblemas que são pequenos o suficiente para serem resolvidos sem a separação são produzidos.

Estratégia

- Considere a estratégia divisão-e-conquista com a separação da entrada em dois subproblemas do mesmo tipo,
- Pode-se escrever uma abstração de controle que espelha a forma como um algoritmo baseado na estratégia parecerá,
- Por *abstração de controle* entende-se um procedimento cujo fluxo de controle é claro mas cujas operações primárias são especificadas por outros procedimentos cujos significados precisos são indefinidos,
- `DEC` é inicialmente chamado como `DEC(P)`, onde `P` é o problema a ser resolvido.

DEC(P)

- PEQUENO(P) é uma função booleana que determina se o tamanho da entrada é pequeno o suficiente para que a resposta possa ser computada sem dividir a entrada,
- Se isso for verdade, a função S é chamada,
- De outra forma, o problema P é dividido em subproblemas menores,
- Estes subproblemas p_1,p_2,...p_k, são resolvidos por aplicações recursivas de DEC,
- COMBINE é uma função que determina a solução para P usando as soluções para os k subproblemas,

DEC(P)

```
Algoritmo DEC(P)
{
 Se PEQUENO(P) então retorne S(P);
 senão
 {
   Divida P em instâncias menores p1, p2, ...pk,  k ≥ 1;
   Aplique DEC a cada um destes subproblemas;
   retorne COMBINE(DEC(p1), DEC(p2), ..., DEC(pk));
 }
}
```

DEC(P)

- Se o tamanho de P é n e os tamanhos dos k subproblemas são n_1, n_2, ..., n_k, respectivamente, então o tempo de computação de DEC(P) é descrito pela **relação (equação) de recorrência**:

$$T(n) = \begin{cases} g(n) & n \text{ pequeno} \\ T(n_1) + T(n_2) + ... + T(n_k) + f(n) & n \text{ grande} \end{cases}$$

- onde $T(n)$ é o tempo para DEC em qualquer entrada de tamanho n e $g(n)$ é o tempo para computar a resposta para entradas pequenas,
- A função $f(n)$ é o tempo para dividir P e combinar as soluções para os subproblemas.

Recorrências

- Para algoritmos baseados em divisão-e-conquista que produzem subproblemas do mesmo tipo do problema original, é muito natural descrever estes algoritmos usando **recursão**.
- A complexidade de muitos algoritmos divisão-e-conquista é dada por recorrências da forma:

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- onde a e b são constantes conhecidas. Assume-se que $T(1)$ é conhecido e n é uma potência de b (isto é, $n = b^k$).

Método da substituição

- Um dos métodos para resolver tal relação de recorrência é chamado de *método da substituição*,
- Este método repetidamente faz substituições para cada ocorrência da função T do lado direito até que todas as ocorrências desapareçam,
- **Exemplo**: Considere o caso no qual $a = 2$ e $b = 2$. Seja $T(1) = 2$ e $f(n) = n$. Tem-se:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\dots \end{aligned}$$

Método da substituição

- Em geral, nota-se que $T(n) = 2^i T(n/2^i) + in$, para qualquer $\log\ n \geq i \geq 1$,
- Em particular, então, $T(n) = 2^{\log\ n} T(n/2^{\log\ n}) + n\ \log\ n$, correspondente a escolha de $i = \log\ n$. Portanto, $T(n) = nT(1) + n\ \log\ n = n\ \log\ n + 2n$.

Sumário

Estratégia

- Outro Exemplo: Seja $T(n) = T(n-1) + t_2$. Pode-se escrever $T(n-1) = T(n-1-1) + t_2$, desde que $n > 1$.
- Como $T(n-1)$ aparece no lado direito da primeira equação, pode-se substituir o lado direito inteiro da última equação,
- Repetindo o processo, chega-se a:

$$\begin{aligned} T(n) &= T(n-1) + t_2 \\ &= (T(n-2) + t_2) + t_2 \\ &= T(n-2) + 2t_2 \\ &= (T(n-3) + t_2) + 2t_2 \\ &= T(n-3) + 3t_2 \\ &\dots \end{aligned}$$

Estratégia

- O próximo passo requer certa intuição. Pode-se tentar obter o padrão emergente. Neste caso, é óbvio: $T(n) = T(n-k) + kt_2$, onde $1 \leq k \leq n$.
- Se houver dúvidas sobre nossa intuição, sempre pode-se provar por indução:
  - **Caso Base**: Para $k = 1$, a fórmula é correta: $T(n) = T(n-1) + t_2$.
  - **Hipótese Indutiva**: Assuma que $T(n) = T(n-k) + kt_2$ para $k = 1, 2, \ldots$. Assim, $T(n) = T(n-l) + lt_2$.
- Note que usando a relação de recorrência original pode-se escrever $T(n-l) = T(n-l-1) + t_2$, para $l \leq n$.

Estratégia

- Logo:

$$\begin{aligned} T(n) &= T(n-l-1)+t_2+lt_2 \\ &= T(n-(l+1))+(l+1)t_2 \end{aligned}$$

- Portanto, por indução em l, a fórmula está correta para todo $0 \leq k \leq n$.
- Portanto, mostrou-se que $T(n) = T(n-k)+kt_2$, para $1 \leq k \leq n$. Agora, se n é conhecido, pode-se repetir o processo até que se tenha $T(0)$ do lado direito.
- O fato de que n é desconhecido não deve ser impeditivo: consegue-se $T(0)$ do lado direito quando $n-k=0$. Isto é, $k=n$. Fazendo $k=n$ tem-se

$$\begin{aligned} T(n) &= T(n-k)+kt_2 \\ &= T(0)+nt_2 \\ &= t_1+nt_2 \end{aligned}$$

Sumário

Big-Oh

- Como as relações de recorrência podem ser usadas para ajudar a determinar o tempo de execução (big-Oh) de funções recursivas,
- Uma função com, características similares: qual é a complexidade assintótica da função FACACOISA mostrada abaixo? Por que?
- Assuma que a função COMBINE roda no tempo $\mathcal{O}(n)$ quando $|left - right| = n$, i.e., quando COMBINE é usada para combinar n elementos no vetor a.

Big-Oh

```
void FacaCoisa(int a[], int left, int right)
// póscondição:  a[left] <= ...  <= a[right]
{
 int mid = (left+right)/2;
 if (left < right)
 {
   FacaCoisa(a,left,mid);
   FacaCoisa(a,mid+1,right);
   Combine(a,left,mid,right);
 }
}
```

- Esta função é uma implementação do algoritmo de ordenação *merge sort*.
- A complexidade do *merge sort* é $\mathcal{O}(n \log n)$ para um vetor de n elementos.

A Relação de Recorrência

- Seja $T(n)$ o tempo para FACACOISA executar em um vetor de n elementos, i.e., quando $|left - right| = n$.
- Veja que o tempo para executar um vetor de um elemento é $\mathcal{O}(1)$, tempo constante.
- Tem-se então o seguinte relacionamento:

$$T(n) = \begin{cases} 2T(n/2) + \mathcal{O}(n) & \text{o } \mathcal{O}(n) \text{ é para COMBINE} \\ \mathcal{O}(1) \end{cases}$$

- Este relacionamento é chamado de *relação de recorrência* porque a função $T(...)$ ocorre em ambos os lados de "=".
- Esta relação de recorrência descreve completamente descreve a função FACACOISA, tal que se se resolver a relação de recorrência pode-se saber a complexidade de FACACOISA já que $T(n)$ é o tempo para executar FACACOISA.

Caso base

- Quando se escreve uma relação de recorrência, deve-se escrever duas equações: uma para o caso geral e uma para o caso base.
- As equações referem-se à função recursiva a qual a recorrência se aplica.
- O caso base é normalmente uma operação $\mathcal{O}(1)$, apesar de poder ser diferente.
- Em algumas relações de recorrência o caso base envolve entrada de tamanho um, tal que escreve-se $T(1) = \mathcal{O}(1)$.
- Entretanto há casos em que o caso base tem tamanho zero.
- Em tais casos, a base poderia ser $T(0) = \mathcal{O}(1)$.

Resolvendo relações de recorrência

- Pode-se realmente resolver a relação de recorrência dada no slide anterior:
  - Escreve-se n em vez de $\mathcal{O}(n)$ na primeira linha para simplificar.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\dots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

Resolvendo relações de recorrência

- Note que a última linha é derivada observando um padrão – esta é a "sacada" – generalização de padrões matemáticos como parte do problema.
- Sabe-se que $T(1) = 1$ e esta é uma forma de terminar a derivação. Na verdade, deseja-se que $T(1)$ apareça do lado direito do sinal de "=".
- Isto significa que se quer $n/2^k = 1$ ou $n = 2^k$ ou $\log\ n = k$.
- Continuando com a derivação anterior, tem-se o seguinte já que $k = \log\ n$

$$\begin{aligned} &= 2^k T(n/2^k) + kn \\ &= 2^{\log\ n} T(1) + (\log\ n)n \\ &= n + n\ \log\ n \quad \text{[lembre-se que } T(1) = 1\text{]} \\ &= \mathcal{O}(n\ \log\ n) \end{aligned}$$

Resolvendo relações de recorrência

- Resolveu-se a relação de recorrência e sua solução é o que se esperava.
- Para tornar isto uma prova formal, seria necessário usar indução para mostrar que $\mathcal{O}(n \log n)$ é a solução para a dada relação de recorrência,
- Mas o método "rápido" mostrado acima mostra como derivar a solução,
- A verificação subsequente que esta é a solução é deixado para algoritmos mais avançados.

Referências I

[1] Astrachan, O. L.
Big-Oh for Recursive Functions: Recurrence Relations.
http://www.cs.duke.edu/~ola/ap/recurrence.html

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.

[3] Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

Referências II

[4] Pardo, T. A. S.
Análise de Algoritmos. SCE-181 Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2008.

[5] Preiss, B. R.
Data Structures and Algorithms with Object-Oriented Design Patterns in C++. 1999.
http://www.brpreiss.com/books/opus4/html/page41.html\#SECTION003151000000000000000

Referências III

[6] Rosa, J. L. G.
Análise de Algoritmos - parte 2 e Divisão e Conquista. SCC-201 Introdução à Ciência da Computação II (capítulo 3).
Slides. Ciência de Computação. ICMC/USP, 2009.