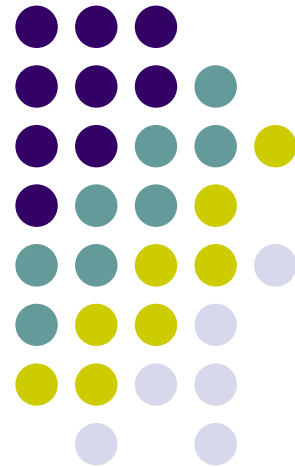


Análise de algoritmos

Parte II





Análise de algoritmos

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores
 - **Empírica** ou **teoricamente**
- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los
 - Função da análise de algoritmos

Calculando o tempo de execução



- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^n i^3$

Início

declare soma_parcial numérico;

soma_parcial \leftarrow 0;

para $i \leftarrow 1$ até n faça

 soma_parcial \leftarrow soma_parcial + $i * i * i$;

escreva(soma_parcial);

Fim

Calculando o tempo de execução



- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^n i^3$

Início

declare soma_parcial numérico;

soma_parcial \leftarrow 0;

para $i \leftarrow 1$ até n faça

 soma_parcial \leftarrow soma_parcial + $i * i * i$;

 escreva(soma_parcial);

Fim

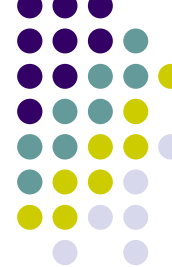
1 unidade de tempo

1 unidade para inicialização de i ,
 $n+1$ unidades para testar se $i \leq n$ e n
unidades para incrementar $i = 2n+2$

4 unidades (1 da soma, 2
das multiplicações e 1 da
atribuição) executada n
vezes (pelo comando
“para”) = $4n$ unidades

1 unidade para escrita

Calculando o tempo de execução



- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^n i^3$

Início

declare soma_parcial numérico;

soma_parcial \leftarrow 0;

para $i \leftarrow 1$ até n faça

soma_parcial \leftarrow soma_parcial + $i * i * i$;

1 unidade de tempo

1 unidade para inicialização de i ,
 $n+1$ unidades para testar se $i \leq n$ e n
unidades para incrementar $i = 2n+2$

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando “para”) = $4n$ unidades

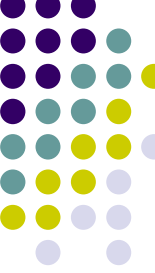
Custo total: somando tudo, tem-se $6n+4$ unidades de tempo, ou seja, a função é **$O(n)$**

1 unidade para escrita

Calculando o tempo de execução

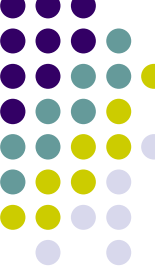


- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa **cansativa**
- Em geral, como se dá a resposta em termos do *big-oh*, **costuma-se desconsiderar as constantes e elementos menores dos cálculos**
 - No exemplo anterior
 - A linha $\text{soma_parcial} \leftarrow 0$ é insignificante em termos de tempo
 - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha $\text{soma_parcial} \leftarrow \text{soma_parcial} + i * i$
 - O que realmente dá a grandeza de tempo desejada é a repetição na linha para $i \leftarrow 1$ até n faça



Regras para o cálculo

- Repetições
 - O tempo de execução de uma repetição é pelo menos o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada



Regras para o cálculo

- Repetições aninhadas
 - A análise é feita de dentro para fora
 - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
 - O exemplo abaixo é $O(n^2)$

para $i \leftarrow 0$ até n faça
 para $j \leftarrow 0$ até n faça
 faça $k \leftarrow k+1$;



Regras para o cálculo

- Comandos consecutivos
 - É a soma dos tempos de cada um, o que pode significar o máximo entre eles
 - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$

para $i \leftarrow 0$ até n faça

$k \leftarrow 0$;

para $i \leftarrow 0$ até n faça

 para $j \leftarrow 0$ até n faça

 faça $k \leftarrow k+1$;



Regras para o cálculo

- Se... então... senão
 - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao então e os comandos relativos ao senão
 - O exemplo abaixo é $O(n)$

se $i < j$
então $i \leftarrow i+1$
senão para $k \leftarrow 1$ até n faça
 $i \leftarrow i*k$;



Regras para o cálculo

- Chamadas a sub-rotinas
 - Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

Exercício

- Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo

Início

declare i e j numéricos;

declare A vetor numérico de n posições;

$i \leftarrow 1$;

enquanto $i \leq n$ faça

$A[i] \leftarrow 0$;

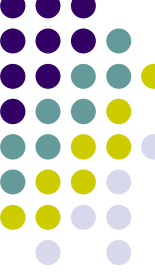
$i \leftarrow i + 1$;

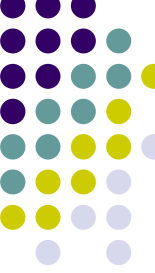
para $i \leftarrow 1$ até n faça

 para $j \leftarrow 1$ até n faça

$A[i] \leftarrow A[i] + i + j$;

Fim





Exercício em duplas

- Analise a sub-rotina recursiva abaixo

sub-rotina fatorial(n: numérico)

início

declare aux numérico;

se $n \leq 1$

então $\text{aux} \leftarrow 1$

senão $\text{aux} \leftarrow n * \text{fatorial}(n-1)$;

fatorial \leftarrow aux;

fim



Regras para o cálculo

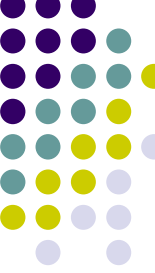
- Sub-rotinas recursivas
 - Se a recursão é um “disfarce” da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal
 - O exemplo anterior é obviamente $O(n)$

```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
se  $n \leq 1$ 
    então  $aux \leftarrow 1$ 
    senão  $aux \leftarrow n * \text{fatorial}(n-1)$ ;
fatorial  $\leftarrow aux$ ;
fim
```

Eliminando
a recursão

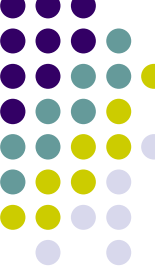


```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
 $aux \leftarrow 1$ ;
enquanto  $n > 1$  faça
     $aux \leftarrow aux * n$ ;
     $n \leftarrow n - 1$ ;
fatorial  $\leftarrow aux$ ;
fim
```



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 fazer uma **análise de recorrência**
 - *Recorrência: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores*
 - Caso típico: algoritmos de **dividir-e-conquistar**, 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
 - Exemplos?



Regras para o cálculo

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

sub-rotina fib(n: numérico)

início

declare aux numérico;

se $n \leq 1$

então $aux \leftarrow 1$

senão $aux \leftarrow fib(n-1)+fib(n-2)$;

$fib \leftarrow aux$;

fim



Regras para o cálculo

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

sub-rotina fib(n: numérico)

início

declare aux numérico;

se $n \leq 1$

então $\text{aux} \leftarrow 1$

senão $\text{aux} \leftarrow \text{fib}(n-1) + \text{fib}(n-2)$;

$\text{fib} \leftarrow \text{aux}$;

fim

Seja $T(n)$ o tempo de execução da função.

Caso 1:

Se $n=0$ ou 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, **$T(0)=T(1)=3$** .



Regras para o cálculo

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

sub-rotina fib(n: numérico)

início

declare aux numérico;

se $n \leq 1$

então $\text{aux} \leftarrow 1$

senão $\text{aux} \leftarrow \text{fib}(n-1) + \text{fib}(n-2)$;

$\text{fib} \leftarrow \text{aux}$;

fim

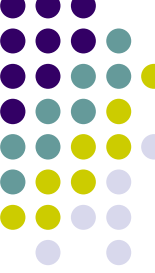
Caso 2:

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 $n > 2$.



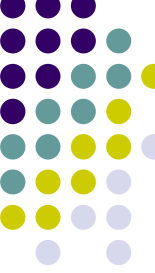
Regras para o cálculo

- 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



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



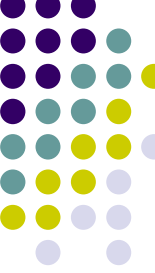
Resolução de recorrências

- Método mestre
 - Fornece limites para recorrências da forma $T(n)=aT(n/b)+f(n)$, 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



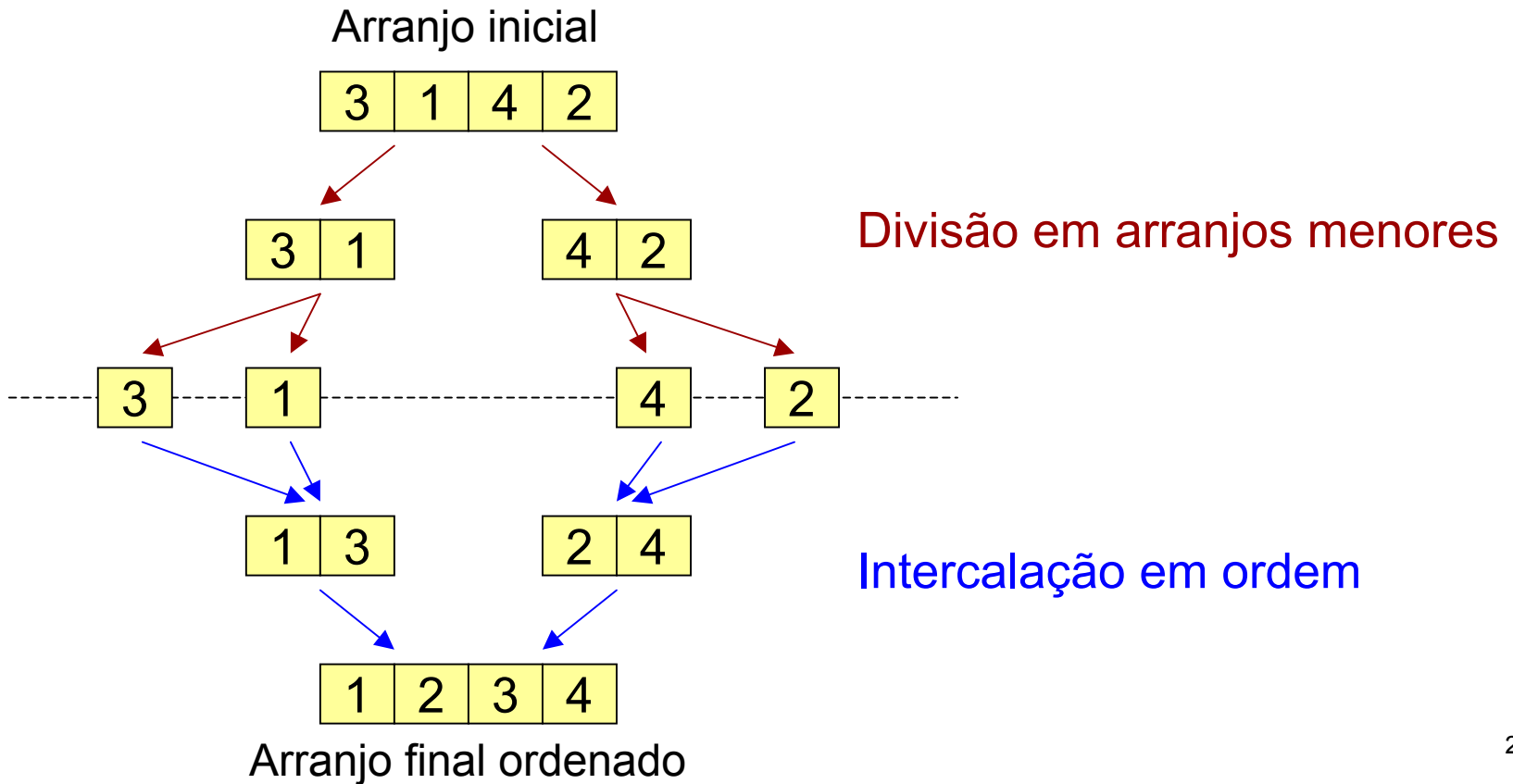
Resolução de recorrências

- 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





Resolução de recorrências

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

$T(n)=c$, se $n=1$ n operações para a intercalação do
 $T(n)=2T(n/2)+cn$, se $n>1$ passo final

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

...

$$= 2^i T(n/2^i) + i(cn)$$

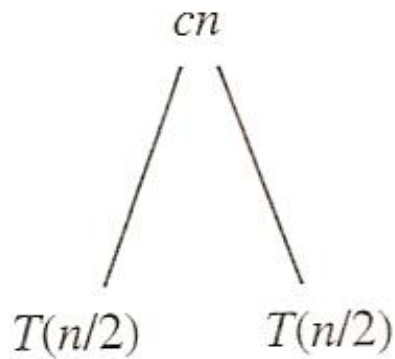
Considerando o limite $n/2^i = 1$, logo $i = \log_2 n$, e

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(1) + \log_2 n (cn) \\ &= cn + cn \log_2 n \\ &= cn \log_2 n + cn \end{aligned}$$

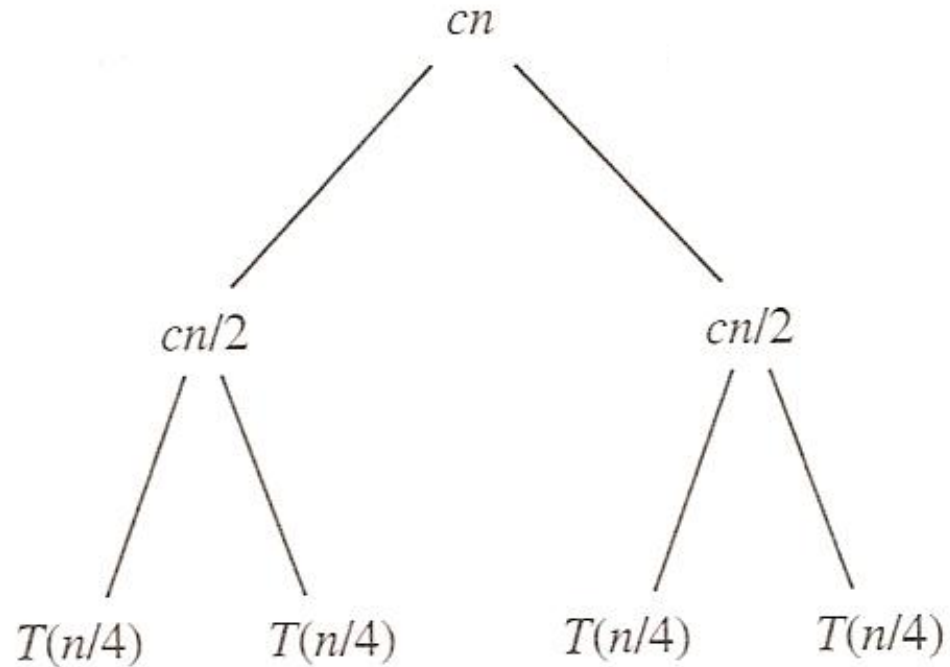
Resolução de recorrências



$T(n)$

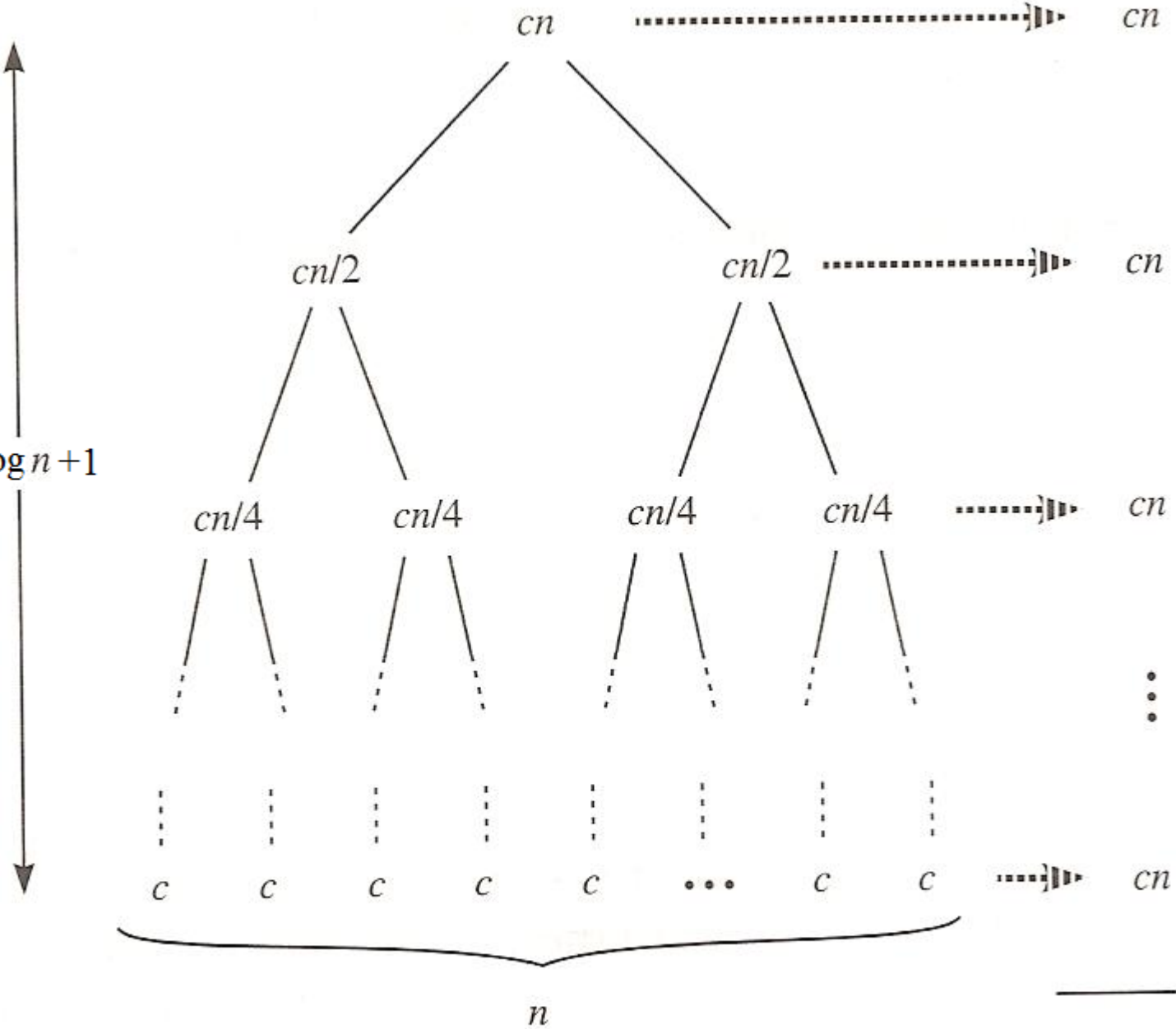


(a)



(b)

(c)



(d)

Total: $cn \log n + cn$



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 ($T(n/2)$ 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 $(\log n)+1$, que é o número de vezes do custo dos níveis (iguais a n)
 - Como resultado, tem-se $cn \log n + cn$, ou seja, $O(n \log n)$

Precauções



- 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 $O(n)$, assintoticamente melhor do que outro com tempo $10n \log n$, 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!

Análise de algoritmos recursivos



- Exemplo: pesquisa binária pelo número 3

Arranjo ordenado

1	3	5	6	8	11	15	16	17
---	---	---	---	---	----	----	----	----

↑ É o elemento procurado?

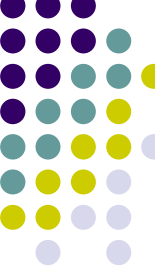
1	3	5	6	8	11	15	16	17
---	---	---	---	---	----	----	----	----

↑ É o elemento procurado?

1	3	5	6	8	11	15	16	17
---	---	---	---	---	----	----	----	----

↑ É o elemento procurado?

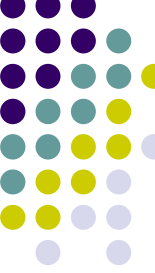
Análise de algoritmos recursivos



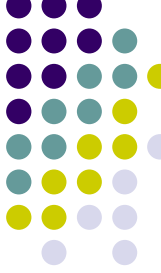
- Implemente o algoritmo da busca binária em um arranjo ordenado

Análise de algoritmos recursivos

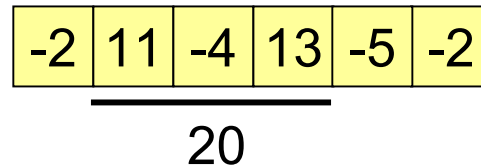
- Teste e analise o algoritmo



Análise de algoritmos recursivos



- Problema da maior soma de subsequência em um arranjo



Análise de algoritmos recursivos



- Faça um algoritmo para resolver o problema e analise-o

Análise de algoritmos recursivos



- Idem para o Algoritmo de Euclides para calcular o máximo divisor comum para 2 números

Análise de algoritmos recursivos



- Idem para o Algoritmo para calcular x^n