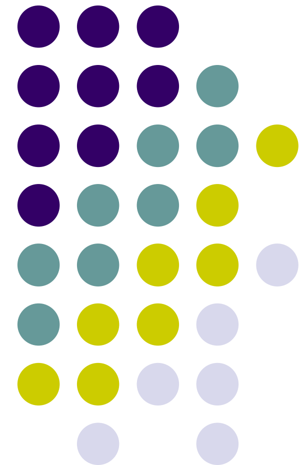
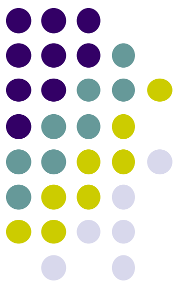
Análise de algoritmos

Introdução à Ciência de Computação II

Baseados nos Slides do Prof. Dr. Thiago A. S. Pardo



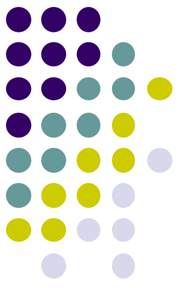
Análise de algoritmos



- 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

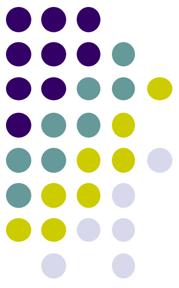
Calculando o tempo de execução



- 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 ← 0;
para i←1 até n faça
    soma_parcial←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 ← 0;
para i←1 até n faça
    soma_parcial←soma_parcial+i*i*i;
escreva(soma_parcial);
Fim
```

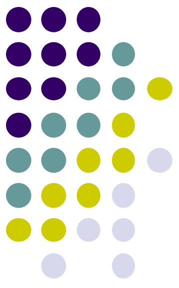
soma_parcial ← 0; → 1 unidade de tempo

para i←1 até n faça → 1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

soma_parcial←soma_parcial+i*i*i; → 4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando "para") = 4n unidades

escreva(soma_parcial); → 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 ← 0;

para i←1 até n faça

...parcial+i*i*i;

1 unidade de tempo

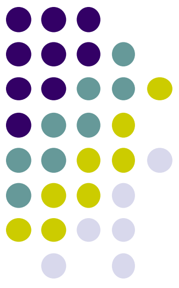
1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

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

1 unidade para escrita

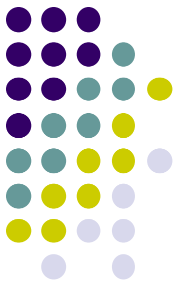
Custo total: somando tudo, tem-se 6n+4 unidades de tempo, ou seja, a função é **O(n)**

Calculando o tempo de execução



- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa

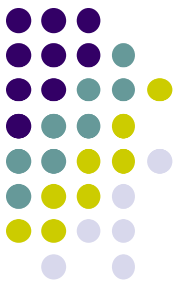
- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
  - No exemplo anterior
    - A linha soma_parcial←0 é insignificante em termos de tempo
    - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha soma_parcial←soma_parcial+i*i*i
    - O que realmente dá a grandeza de tempo desejada é a repetição na linha **para i←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

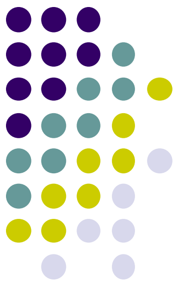
Regras para o cálculo



- 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←0 até n faça
    para j←0 até n faça
            faça k←k+1;
```

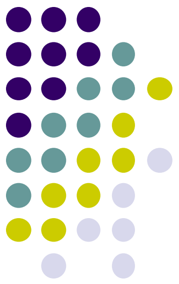
Regras para o cálculo



- 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←0 até n faça
    k←0;
para i←0 até n faça
    para j←0 até n faça
            faça k←k+1;
```

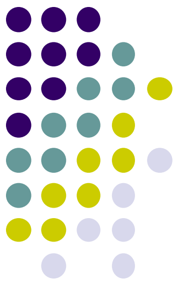
Regras para o cálculo



- 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)

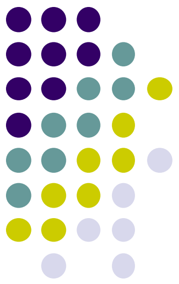
```
se i<j
   então i←i+1
   senão para k←1 até n faça
                        i←i*k;
```

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

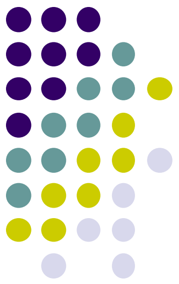
Exercício



- 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←1;
enquanto i≤n faça
    A[i]←0;
    i←i+1;
para i←1 até n faça
    para j←1 até n faça
       A[i]←A[i]+i+j;
Fim
```

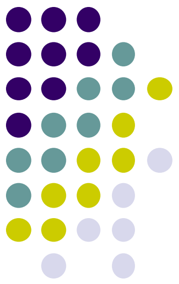
Exercício



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

```
Início
declare i, j e x numéricos;
declare m uma matriz de N linhas por N colunas;
para i←1 até N faça
    j←1;
    enquanto j<N faça
        x←m[i,j];
        imprima(x);
        j←j+2;
imprima(i);
Fim
```

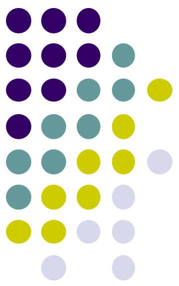
Exercício



- Analise a sub-rotina recursiva abaixo

```
sub-rotina fatorial(n: numérico)
início
    declare aux numérico;
    se n≤1 então
       aux←1
    senão
       aux←n*fatorial(n-1);
    retorne aux;
fim
```

Regras para o cálculo



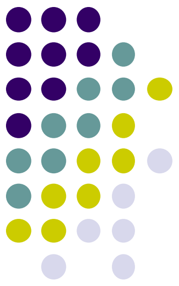
- Sub-rotinas recursivas
  - Se a recursão é um “disfarce” da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal
  - O exemplo anterior é obviamente O(n)

```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
se n≤1
        então aux←1
        senão aux←n*fatorial(n-1);
retorne aux;
fim
```

Eliminando a recursão

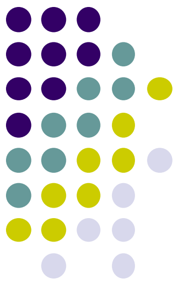
```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
aux←1;
enquanto n>1 faça
        aux←aux*n;
        n←n-1;
retorne aux;
fim
```

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 se recorrer à 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?

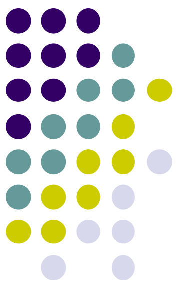
Regras para o cálculo



- 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≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne aux;
fim
```

Regras para o cálculo

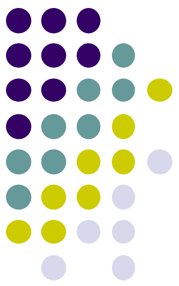


- Exemplo de uso de recorrência
  - Números de Fibonacci
    - 0,1,1,2,3,5,8,13...
    - f(0)=0, f(1)=1, f(i)=f(i-1)+f(i-2)

```
sub-rotina fib(n: numérico)
início
declare aux numérico;
se n≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne 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 o retorno da função; ou seja, T(0)=T(1)=3.



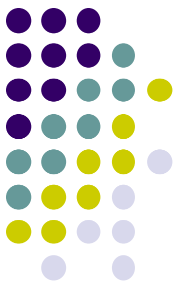
Regras para o cálculo

- 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≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne 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 o retorno da função; ou seja, a recorrência T(n)=T(n-1)+T(n-2)+4, para n>2.

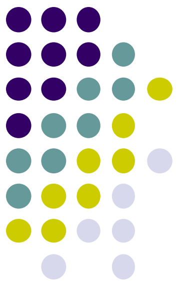
Regras para o cálculo



- Muitas vezes, a recorrência pode ser resolvida com base na prática e experiência do analista

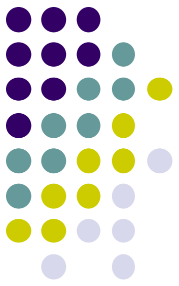
- Alguns métodos para resolver recorrências
  - Método da substituição
  - Método da árvore de recursão
  - Método mestre

Resolução de recorrências



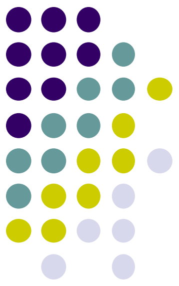
- Método da substituição
  - Supõe-se (aleatoriamente ou com base na experiência) um limite superior para a função e verifica-se se ela não extrapola este limite
    - Uso de indução matemática
  - O nome do método vem da "substituição" da resposta adequada pelo palpite
  - Pode-se "apertar" o palpite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

Resolução de recorrências

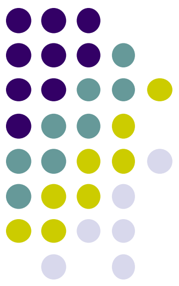


- Método da árvore de recursão
  - Esboça-se uma árvore que, nível a nível, representa as recursões sendo chamadas
  - Em seguida, em cada nível/nó da árvore, são acumulados os tempos necessários para o processamento
    - No final, tem-se a estimativa de tempo do problema
  - Este método pode ser utilizado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

Arranjo inicial

3	1	4	2

Divisão em arranjos menores

3	1

4	2

3	1	4	2

Intercalação em ordem

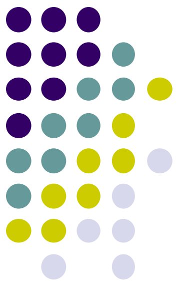
1	3

2	4

1	2	3	4

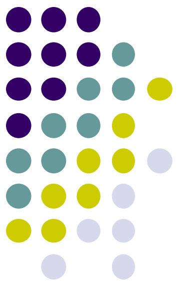
Arranjo final ordenado

Ordenação por intercalação



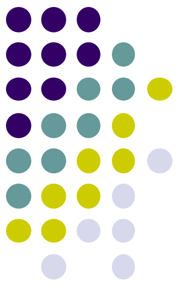
- Implemente a(s) sub-rotina(s) e calcule sua complexidade
  - Subrotina de intercalação
  - Subrotina recursiva

Ordenação por intercalação



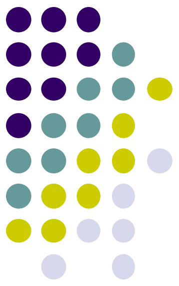
- Rotina principal: mergesort
  - Se n=1 elemento no arranjo, ordenação não é necessária: ?
  - Se n>1
    - O problema é inicialmente dividido em subproblemas: ?
    - Os subproblemas são processados: ?
    - As soluções são combinadas: complexidade da rotina auxiliar de intercalação

Ordenação por intercalação



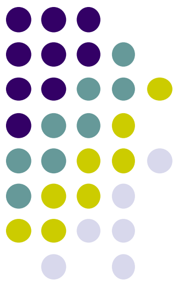
- Rotina principal: mergesort
  - Se n=1 elemento no arranjo, ordenação não é necessária: 1 operação é realizada, tempo constante O(c)
  - Se n>1
    - O problema é inicialmente dividido em subproblemas: 3 operações, tempo constante O(c)
    - Os subproblemas são processados: 2 subproblemas, sendo que cada um tem metade do tamanho original = 2T(n/2)
    - As soluções são combinadas: O(n)

Ordenação por intercalação



- Equações de complexidade do algoritmo
  - ???

Ordenação por intercalação



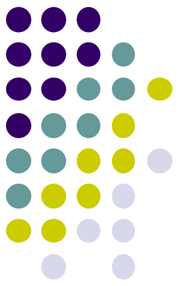
- Equações de complexidade do algoritmo

  $T(n)=O(c)=1$, se $n=1$

  $T(n)=2T(n/2) + \underbrace{O(c) + O(n)}_{}$, se $n>1$

  $O(n)$, já que $c<n$ em geral

Ordenação por intercalação

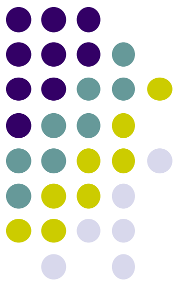


- Equações de complexidade do algoritmo

  $T(n)=1$, se $n=1$

  $T(n)=2T(n/2) + O(n)$, se $n>1$

Ordenação por intercalação

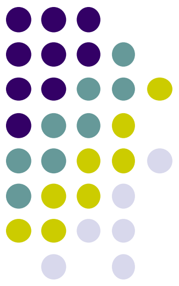


- Equações de complexidade do algoritmo

  $T(n)=1$, se $n=1$

  $T(n)=2T(n/2) + n$, se $n>1$

Ordenação por intercalação



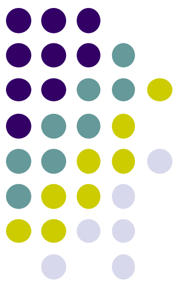
- Equações de complexidade do algoritmo

  $T(n)=1$, se $n=1$

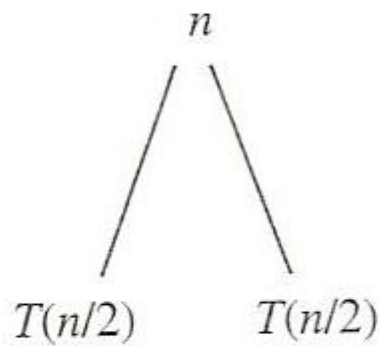
  $T(n)=2T(n/2) + n$, se $n>1$

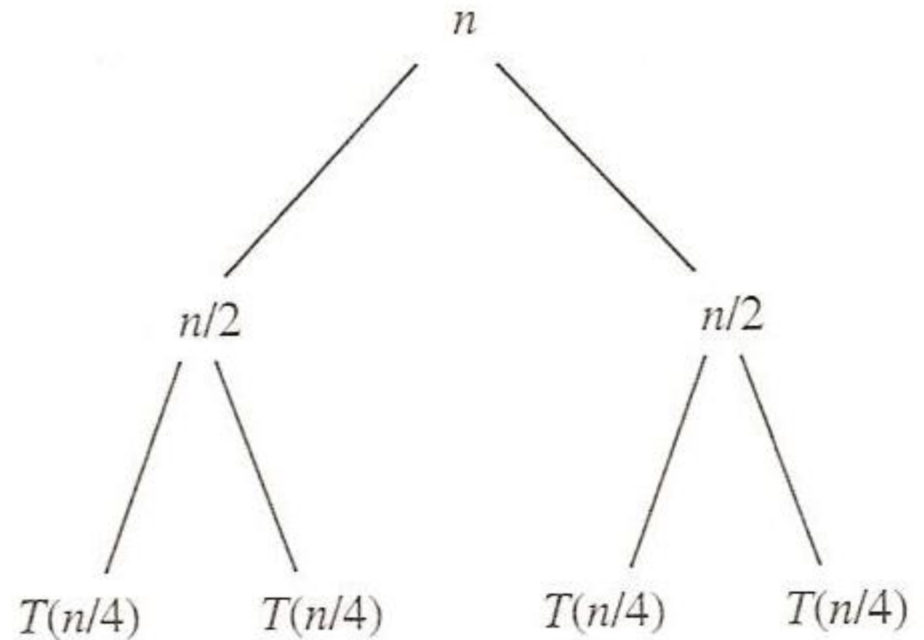
EQUAÇÃO DE RECORRÊNCIA, podendo ser resolvida via árvore de recorrência

Resolução de recorrências

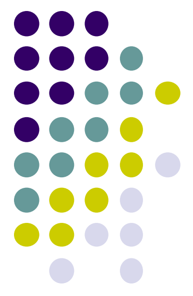


$T(n)$





(a) (b) (c)



n → n

$n/2$ $n/2$ → n

$\log n + 1$

$n/4$ $n/4$ $n/4$ $n/4$ → n

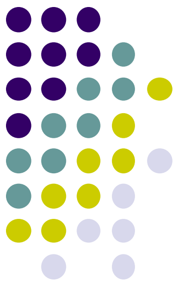
⋮

1 1 1 1 1 … 1 1 → n

Total: $n \log n + n$

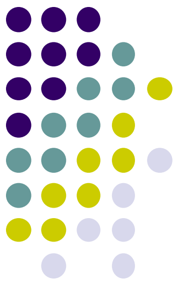
(d)

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 n o custo no nível superior da recursão (fora da recursão e, portanto, associado ao nó raiz)
  - ...
  - No fim, nota-se que a altura 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 n logn + n, ou seja, O(n log n)

Busca binária



- Busca binária em um vetor ordenado
  - Exemplo: pesquisa pelo número 3

Vetor ordenado

1	3	5	6	8	11	15	16	17

É o elemento procurado?

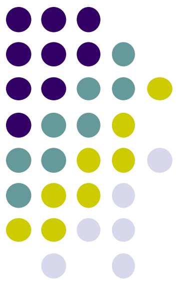
1	3	5	6	8	11	15	16	17

É o elemento procurado?

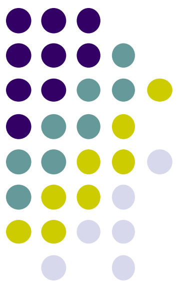
1	3	5	6	8	11	15	16	17

É o elemento procurado?

Busca binária

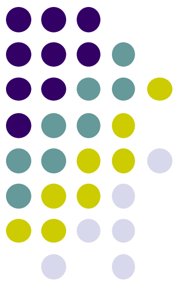


- Implemente o algoritmo recursivo da busca binária em um vetor ordenado



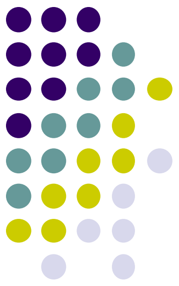
```
// A função bb recebe um número x e um vetor
// crescente v[e..d]. Ela devolve um índice m
// em e..d tal que v[m] == x. Se tal m não
// existe, devolve -1.

int
bb( int x, int e, int d, int v[]) {
   if (e > d) return -1;
   else {
      int m = (e + d)/2;
      if (v[m] == x) return m;
      if (v[m] < x)
         return bb( x, m+1, d, v);
      else
         return bb( x, e, m-1, v);
   }
}
```

Busca binária



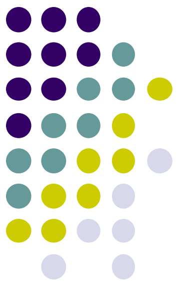
- Teste e analise o algoritmo

Busca binária

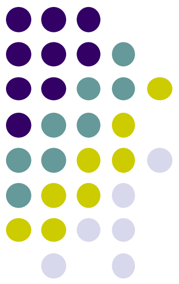


- Análise de tempo
  - Se n=1, então tempo constante O(c), não importando se achou ou não o elemento
  - Se n>1, então
    - Comparações e divisão/diminuição do problema: tempo constante O(c)
    - Processamento do subproblema: T(n/2)

Busca binária



- Equações de complexidade de tempo
  - $T(n)=O(c)$, se $n=1$
  - $T(n)=T(n/2) + O(c)$, se $n>1$

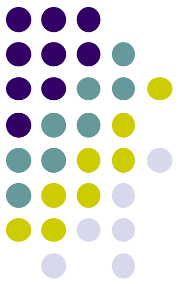


Busca binária

- Equações de complexidade de tempo
  - $T(n)=O(c)$, se $n=1$
  - $T(n)=T(n/2) + O(c)$, se $n>1$

$T(n/2)$

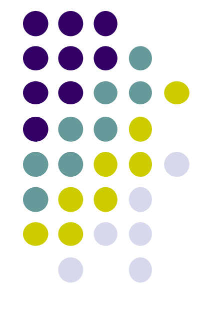
Busca binária



- Equações de complexidade de tempo
  - $T(n)=O(c)$, se $n=1$
  - $T(n)=T(n/2)$, se $n>1$

EQUAÇÃO DE RECORRÊNCIA, possível de resolver via árvore de recorrência

Busca binária



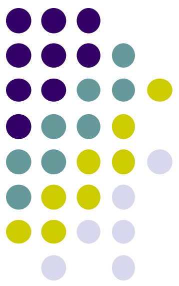
- Árvore de recorrência

T(n) ➡ c — T(n/2) ➡ c — c — T(n/4) ➡ c — c ⋮ T(1) ➡ c — c ⋮ c

altura=log n + 1

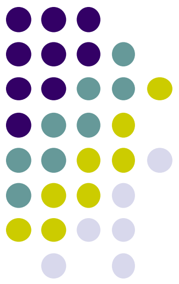
T(n) = c * (log n + 1) = c log n + c = O(log n)

Resolução de recorrências



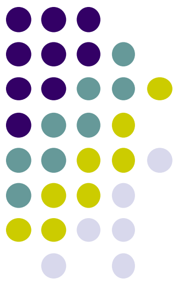
- Método mestre
  - Fornece limites para recorrências da forma $T(n)=aT(n/b)+f(n)$, em que $a \geq 1$, $b>1$ e $f(n)$ é uma função dada
  - Envolve a memorização de alguns casos básicos que podem ser aplicados para muitas recorrências simples

Método mestre



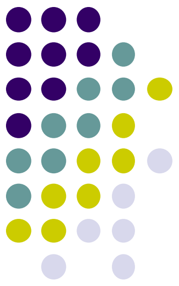
- Seja
  $T(n)=aT(n/b)+f(n)$,
  em que $a \geq 1$, $b>1$ e $f(n)$ é uma função dada.
- A equação acima descreve o tempo de cálculo de um problema de tamanho n dividido em
  - a subproblemas de tamanha n/b cada
  - Cada um dos a subproblemas são resolvidos recursivamente em tempo $T(n/b)$
  - $f(n)$ representa o custo de se dividir o problema e combinar os resultados dos subproblemas

Método mestre



- Observação em $T(n)=aT(n/b)+f(n)$
- Como tratar os casos em que n/b não gera um número inteiro
  - É possível mostrar que o arredondamento para cima ou para baixo não afeta a solução assintótica
  - Ver demonstração em **Cormen et al. 2009.**

Método mestre

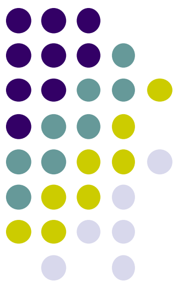


- Seja a >= 1 e b > 1 constantes. Seja f(n) uma função. Considere que T(n) seja definida para n>0 (n é inteiro) como
  - $T(n) = a\,T(n/b) + f(n)$

- Então T(n) possui umas das seguintes formas assintóticas
  - Se $f(n) = O(n^{\log_b a - e})$ para algum e>0, então $T(n) = \Theta(n^{\log_b a})$.
  - Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$
  - Se $f(n) = \Omega(n^{\log_b a + e})$ para algum e>0, e se $a f(n/b) <= c f(n)$ para alguma constante c < 1 e todo n suficientemente grande, então $T(n) = \Theta(f(n))$

- Ver demonstração em **Cormen et al. 2009.**

Método mestre



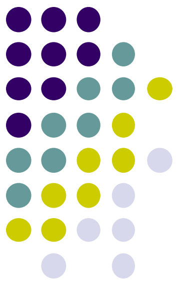
Em todos os três casos, f(n) é comparada com $n^{\log_b a}$

- Caso 1: Se $n^{\log_b a}$ é maior que f(n), a solução é $T(n) = \Theta(n^{\log_b a})$.
- Caso 3: Se f(n) é maior que $n^{\log_b a}$ então $T(n) = \Theta(f(n))$
- Caso 2: Se as funções tem o mesmo tamanho, multiplica-se por um fator logaritmo para obter $T(n) = \Theta(n^{\log_b a} \lg n)$

Em (1), f(n) tem que ser menor do que $n^{\log_b a}$ por um fator n^e para alguma constante $e > 0$

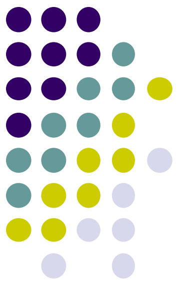
Em (3), f(n) tem que ser maior do que $n^{\log_b a}$ por um fator n^e para alguma constante $e > 0$. Além disso, a condição de regularidade a $f(n/b) <= c\, f(n)$ deve ser satisfeita

Método mestre



- Alguns *gaps* entre as 3 possibilidades possíveis
  - f(n) é assintoticamente menor que ($n^{\log_b a}$), mas não polinomialmente
  - f(n) é assintoticamente maior que ($n^{\log_b a}$), mas não polinomialmente
  - A condição de regularidade não é satisfeita

Método mestre: exemplos



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

$a = 9$; $b = 3$; $f(n) = n$; $n^{\log_b a} = n^{\log_3 9} = n^2$

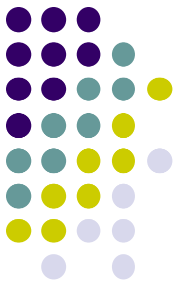
Caso 1

Fazendo $e = 1 \rightarrow f(n) = n = O\ (n^{2 - 1}) = O(n)$

$\rightarrow$ Caso 1: $T(n) = \Theta(n^{\log_b a})$

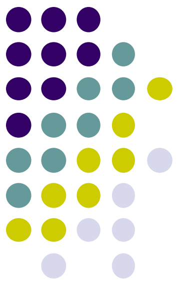
$\rightarrow T(n) = \Theta(n^2)$.

Método mestre: exemplos



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

Método mestre: exemplo



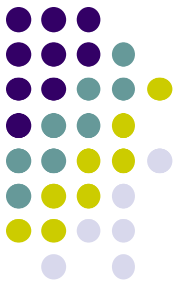
- $T(n) = T(2n/3) + 1$

$a=1$, $b = 3/2$ e $f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Caso 2: $f(n) = \Theta(n^{\log_b a})$

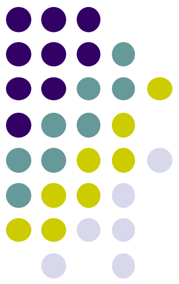
→ $T(n) = \Theta(1.\lg n)$



Método mestre: exemplo

- $T(n) = 3T(n/4) + n \lg n$

Método mestre: exemplo



- $T(n) = 3T(n/4) + n \lg n$

$a = 3;\ b = 4;\ f(n) = n \lg n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

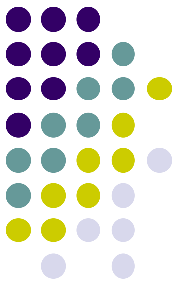
Caso 3: $f(n) = \Omega(n^{\log_b a + e})$, com $e = 0.2$

Regularidade:

$a f(n/b) = 3(n/4) \lg(n/4) <= (3/4)\, n \lg n = c\, f(n)$

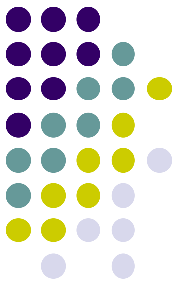
$T(n) = \Theta(n \lg n)$

Método mestre



- $T(n) = 2\,T(n/2) + n \lg n$

Método mestre



- $T(n) = 2\ T(n/2) + n \lg n$

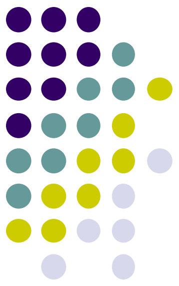
$a = 2;\ b = 2;\ f(n) = n \lg n$

$n^{\log_2 2} = n$

Caso 3:

$f(n) = n \lg n$ é assintoticamente maior que $n^{\log_2 2} = n$

Método mestre



- $T(n) = 2\,T(n/2) + n \lg n$

$a = 2$; $b = 2$; $f(n) = n \lg n$

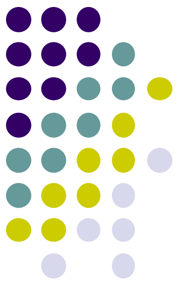
$n^{\log_2 2} = n$

Caso 3:

$f(n) = n \lg n$ é assintoticamente maior que $n^{\log_2 2} = n$

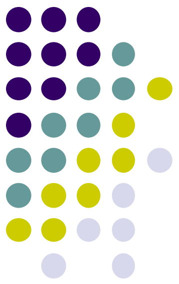
Mas não é polinomialmente maior !

Método mestre



- Usando o método mestre, calcule a complexidade da busca binária recursiva
  - T(n) = ?

Método mestre



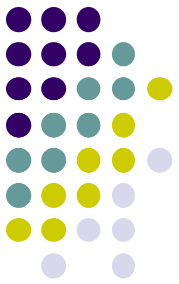
- Complexidade da busca binária recursiva

$T(n) = T(n/2) + \Theta(c)$

$n^{\log_b a} = n^{\log_2 1} = 1 = \Theta(c)$

$f(n) = \Theta(c)$

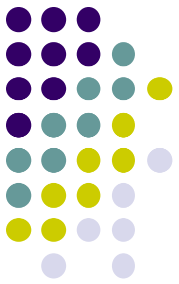
Caso 2: $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(1.\lg n)$



Método mestre

- Resolvendo o problema do mergeSort
  - $T(n) = 2T(n/2) + \Theta(n)$

Método mestre



- Resolvendo o problema do mergeSort
  - $T(n) = 2T(n/2) + \Theta(n)$

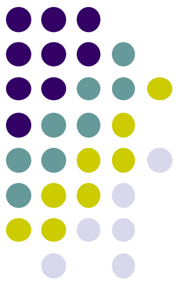
$a=2;\ b=2;\ f(n) = \Theta(n)$

$n^{\log_a b} = n^{\log_a b} = n$

Caso 2: $f(n) = \Theta(n^{\log_a b}) = \Theta(n) \rightarrow$

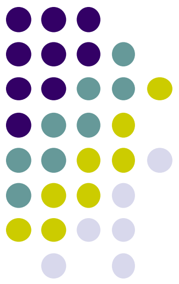
$T(n) = n^{\log_a b} \lg n = n \lg n$

Método mestre



- Algoritmo recursivo para multiplicação de matrizes (não otimizado)
  - $T(n) = 8T(n/2) + \Theta(n^2)$

Método mestre



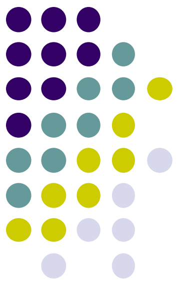
- Algoritmo recursivo para multiplicação de matrizes
- $T(n) = 8T(n/2) + \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 8} = n^3$ -- $f(n) = O(n^2)$

Caso 1: $f(n) = O(n^{3-e})$, com $e=1$

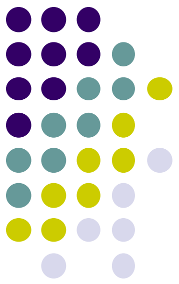
Portanto, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

Método mestre



- Algoritmo *Strassen* para multiplicação de matrizes
  - $T(n) = 7T(n/2) + \Theta(n^2)$

Método mestre



- Algoritmo *Strassen* para multiplicação de matrizes

$T(n) = 7T(n/2) + \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 7}$

Lembrando que $2.80 < \log_2 7 < 2.81$, temos

$f(n) = O(n^{\log_2 7 - e})$, para $e = 0.8$,

Caso 1: $T(n) = T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.807})$