Análise e Projeto de Algoritmos

Mestrado em Ciência da Computação

Prof. Dr. Aparecido Nilceu Marana

Faculdade de Ciências

Bauru

"I think the design of efficient algorithms is somehow the core of computer science. It's at the center of our field".

Donald Knuth

***Objetivo do curso**:*

Enfatizar a eficiência como critério para projeto de algoritmos

Porquê o estudo da Complexidade?

Muitas vezes as pessoas quando começam a estudar algoritmos perguntam-se qual a necessidade de desenvolver novos algoritmos para problemas que já têm solução.

Porquê o estudo da Complexidade?

A performance é extremamente importante na Informática, pois existe uma necessidade constante de melhorar os algoritmos. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do "tamanho" dos problemas a serem resolvidos.

Complexidade de Tempo	Tamanho máximo de problema resolvível na máquina lenta	Tamanho máximo de problema resolvível na máquina 10 vezes mais rápida
$\log_2 n$	x_0	$(x_0)^{10}$
n	x_1	$10\ x_1$
$n \log_2 n$	x_2	$10\ x_2$ (para x_2 grande)
n^2	x_3	$3{,}16\ x_3$
n^3	x_4	$2{,}15\ x_4$
2^n	x_5	$x_5 + 3{,}3$
3^n	x_6	$x_6 + 2{,}09$

Problema:

- *é simplesmente uma tarefa a ser executada;*
- *é uma função ou associação de entradas com saídas.*

Algoritmo:

- *é um método ou um processo usado para resolver um problema.*

Programa:

- *é uma instanciação de um algoritmo em uma linguagem de programação computacional.*

Problema	Algoritmo
Ordenação	Quicksort Mergesort
Pesquisa	Busca sequencial Busca binária

Quando estudamos algoritmos, um aspecto que devemos considerar, ***além de sua correção****, é a sua* ***análise de eficiência.***

DEFINIÇÕES:

- *A análise de algoritmos pode ser definida como o estudo da estimativa de recursos (tempo, espaço, etc) consumidos pelos algoritmos.*
- *Analisar um algoritmo significa prever os recursos que o algoritmo necessitará (Cormen, et al. 2001).*
- *Análise de algoritmos significa, também, estimar o grau de dificuldade dos problemas.*

Análise de Algoritmos:

- Critérios
  - Correção
  - Eficiência temporal
  - Eficiência espacial
  - Otimalidade
- Dois métodos:
  - Análise teórica
  - Analise empírica

Correção de algoritmos

We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

> **Loop invariant:** At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction:

Correção de algoritmos

```
INSERTION-SORT(A)
for j ← 2 to n
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
       A[i + 1] ← key
```


Correção de algoritmos

For insertion sort:

Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 . . j - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 . . n]$ consists of the elements originally in $A[1 . . n]$ but in sorted order. In other words, the entire array is sorted!

Análise de Algoritmos:

Abordagem Experimental

- Implementar o algoritmo e executar o programa para um conjunto de dados de teste
- Porém
  - não podemos testar todas as possíveis entradas.
  - podemos esquecer algum caso em que o algoritmo falha ou caso em que o desempenho do algoritmo é particularmente bom ou ruim
  - Os resultados dependem de aspectos de implementação

Análise de Algoritmos: *empírica*

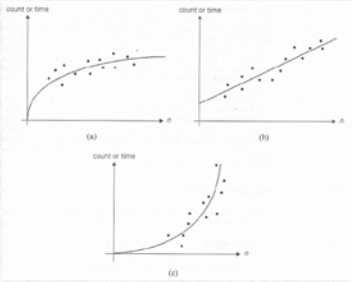
- Um plano geral para a análise empírica da eficiência de um algoritmo segue os seguintes passos:

1. Entender o propósito do experimento
2. Escolher uma métrica de eficiência M a ser medida e a unidade de medida (unidade de tempo ou número de operações básicas)
3. Escolher as características da amostra de entrada (faixa, tamanho, etc.)

Análise de Algoritmos: *empírica*

4. Escrever um programa implementando o algoritmo para a experimentação
5. Gerar uma amostra de entradas
6. Executar o algoritmo sobre as amostras e guardar os dados obtidos
7. Analisar os dados obtidos

Análise de Algoritmos: *empírica*



Análise de Algoritmos:

Abordagem Teórica

- Estudar o algoritmo em termos gerais e tentar prever aspectos gerais do seu comportamento:
  - Correção: Ele fornece uma solução válida para o problema ?
  - Eficiência: Quanto tempo ele gasta ? Quanto de memória ele usa ?

Estrutura de análise de algoritmos:

- Como analisar a eficiência de algoritmos?
- Existem dois tipos de eficiência: **eficiência temporal** e **eficiência espacial**.
  - A **eficiência temporal** indica quão rápido um algoritmo em questão é executado.
  - A **eficiência espacial** está relacionada com o espaço extra que o algoritmo necessita.

Estrutura de análise de algoritmos:

- Antigamente, ambos recursos – tempo e espaço – eram valiosos.
- Atualmente, a quantidade extra de espaço requerida por um algoritmo não é tão importante, ainda que exista uma diferença entre memória principal, secundária e *cache*.
- Porém, o tempo continua sendo importante, pois, problemas cada vez mais complexos são tratados → abordaremos somente eficiência temporal

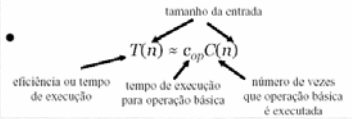
Tamanho da entrada:

- Quase todos os algoritmos levam mais tempo para ser executados sobre entrada maiores.
- Assim, é obvio investigar a eficiência de algoritmos como função do parâmetro ***n*** que indica o tamanho da entrada do algoritmo.

Tempo de execução:

- Por que não utilizar unidade "física" de tempo?
  - Dependência do *hardware*, qualidade da implementação, compilador, etc.
- Métrica que não dependa de fatores externos. Alternativas:
  - Contar o número de vezes que cada operação do algoritmo é realizada
  - Identificar a operação mais importante do algoritmo (operação básica), a operação que mais contribui para o tempo de execução total e contar o número de vezes que esta operação é realizada.

Eficiência de algoritmos:

- Eficiência temporal é analisada determinando o número de repetições de operações básicas com uma função do tamanho da entrada
- Operação básica: a operação que contribui mais para o tempo de execução do algoritmo.
- 

EM ANÁLISE DE ALGORITMOS ESTUDAMOS:

- *O conceito de taxa de crescimento de funções;*
- *O conceito de limite superior e inferior de uma taxa de crescimento, e como estimar estes limites para um algoritmo ou problema; e*
- *A diferença entre o custo de um algoritmo e o custo de um problema.*

FERRAMENTAS MATEMÁTICAS EXIGIDAS:

- *Análise Combinatória;*
- *Teoria das probabilidades;*
- *Destreza matemática:*
  - *Indução Matemática;*
  - *Séries e Produtórios;*
  - *Potências e Logaritmos, etc.*

PARA QUE UTILIZAMOS A ANÁLISE DE ALGORITMOS?

- *Para saber se os algoritmos são viáveis;*
- *Para saber qual é o melhor algoritmo para a resolução do problema;*
- *Para projetar algoritmos mais eficientes.*

CUSTO DO ALGORITMO

Quando analisamos um algoritmo nós desejamos saber qual é o seu custo.

CUSTO DO ALGORITMO

Para sabermos do custo nós necessitamos:

- *Prever a quantidade de recursos (memória - complexidade de espaço, e tempo de execução - complexidade de tempo);*
- *Do modelo de tecnologia adotado para a sua implementação.*

MODELO DE COMPUTAÇÃO

Ao invés de escolher uma máquina particular, em relação a qual a eficiência dos algoritmos seria avaliada, é certamente mais conveniente utilizar-se de um modelo matemático de um computador.

MODELO DE COMPUTAÇÃO

Modelo adotado: (modelo RAM)

- *as operações são todas executadas seqüencialmente;*
- *a execução de toda e qualquer operação toma uma unidade de tempo;*
- *a memória é infinita.*

TEMPO DE EXECUÇÃO

O tempo de execução de um algoritmo sobre uma particular entrada de dados é o número de operações primitivas ou passos executados.

Exemplos de operações: adição, subtração, multiplicação, comparação, ...

TEMPO DE EXECUÇÃO

O tempo de execução (custo) é expresso em função do tamanho da entrada de dados.

Custo = T(n): é a medida de tempo necessário para executar um algoritmo para um problema de tamanho n.

EXEMPLOS DE COMPLEXIDADE DE TEMPO

Vamos tomar dois algoritmos de ordenação e analisar a quantidade de certas operações que são executadas por cada algoritmo:

- *Primeiro Caso: Bubblesort (algoritmo da bolha)*
- *Segundo Caso: Seleção Direta*

EXEMPLOS DE COMPLEXIDADE DE TEMPO

***Bubblesort** é o mais primitivo dos métodos de ordenação de um vetor. A idéia é percorrer um vetor de n posições n vezes, a cada vez comparando dois elementos e trocando-os caso o primeiro seja maior que o segundo.*


```
OrdenaBolha
 Inteiro i, j, x , a[n]
inicio
 para i de 1 até n faça
    para j de 2 até n faça
       se (a[j-1]>a[j]) então
          x ← a[j-1];
          a[j-1] ← a[j];
          a[j] ← x;
       fim se
    fim para
 fim para
fim
```

Bubblesort

A comparação (a[j-1]>a[j]) vai ser executada n(n-1) vezes. No caso de um vetor na ordem inversa, as operações de atribuição poderão ser executadas até 3((n-1)+(n-2)+ ...+2+1)=3n(n-1)/2 vezes, já que uma troca de elementos não significa que um dos elementos trocados tenha encontrado o seu lugar definitivo.

EXEMPLOS DE COMPLEXIDADE DE TEMPO

***Seleção Direta** é uma forma intuitiva de ordenar um vetor. Primeiramente, escolhemos o menor elemento do vetor e o trocamos de posição com o primeiro. Depois, começamos do segundo e escolhemos novamente o menor dentre os restantes e o trocamos de posição com o segundo e assim por diante.*


```
SeleçãoDireta
   Inteiro i, j, x , a[n]
Inicio
 para i de 1 até n-1 faça
   k ← i;
   x ← a[i];
   para j de i+1 até n faça
      se (a[j]<x) então
         k ← j;
         x ← a[k];
      fim se
   fim para
   a[k] ← a[i];
   a[i] ← x;
 fim para
fim
```


Seleção Direta

Neste algoritmo o número de vezes que a comparação (a[j] < x) é executada no pior caso é igual a (n-1)+(n-2)+ ...+2+1=[n(n-1)]/2.

O número de trocas (a[k] ← a[i]; a[i] ← x) realizado no pior caso é igual a 2(n-1). O pior caso acontece quando o maior elemento está na primeira posição e o restante já está ordenado.

INTERPRETAÇÃO

Observe que os dois exemplos considerados foram contabilizados os números de comparações e de trocas. Estes dois parâmetros foram tomados como critérios para comparar para comparar os dois algoritmos de ordenação.

INTERPRETAÇÃO

Como já foi dito, a única forma de se poder comparar dois algoritmos é descrevendo o seu comportamento temporal/espacial em função do tamanho do conjunto de dados de entrada.

Assim, se tomarmos as operações de troca de valores como critério para calcular a eficiência dos algoritmos, podemos dizer que:

Custo Bubblesort

$T(n) = 3n(n-1)/2 = 1{,}5n^2 - 1{,}5n$ *para o pior caso;*

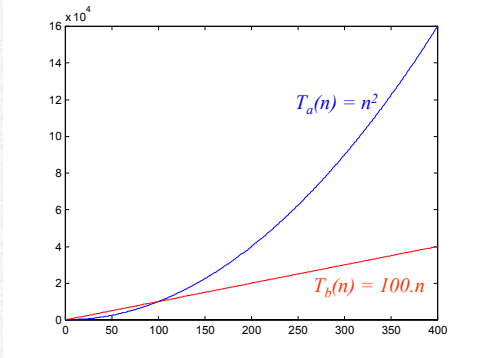
Custo Seleção Direta

$T(n) = 2(n-1) = 2n - 2$ *para o pior caso.*

Note que nos dois exemplos foram tomados como medida de desempenho a quantidade de "operações de trocas", mas em análise de algoritmos nós não nos prendemos somente a este aspecto.

COMPARAÇÃO DE ALGORITMOS

Vamos supor que nós temos dois algoritmos a e b para a solução de um problema. Se a complexidade de um é expressa por $T_a(n) = n^2$ *e a do outro por* $T_b(n) = 100.n$. *Isto significa que o algoritmo a cresce quadraticamente (uma parábola) e que o algoritmo b cresce linearmente (embora seja uma reta bem inclinada).*



COMPARAÇÃO DE ALGORITMOS

É interessante saber como o algoritmo se comporta com uma quantidade de dados realística para o problema e o que acontece quanto esta quantidade varia. A eficiência do algoritmo torna-se muito importante quando o problema a ser resolvido é de grande dimensão, ou seja, problemas definidos por grande quantidades de dados.

TIPOS DE ANÁLISE

Não estamos interessados somente no tempo de execução geral, estamos também interessados nos extremos:

***Pior caso** – significa o tempo máximo de execução.*

***Caso médio** – tempo médio de execução para todo tipo de entrada possível. Neste caso é necessário conhecer a distribuição estatística dos dados de entrada.*

***Melhor caso** - resultado do menor tempo possível.*

Pior Caso

Este método é normalmente representado por ***O ( ).*** *S*e dissermos que um determinado algoritmo é representado por g(x) e a sua complexidade Pior Caso é n, será representada por g(x) = **O**(n).

Consiste basicamente em assumir o pior dos casos que podem acontecer, sendo muito usado e sendo normalmente o mais fácil de determinar.

Exemplo:
Se existirem cinco baús, sendo que apenas um deles tem algo dentro e os outros estão vazios, a complexidade pior caso será O(5), pois no pior dos casos acerta-se o baú cheio na quinta tentativa.

Melhor Caso

Representa-se por Ω ( ).

Método que consiste em assumir que vai acontecer o melhor.

Pouco usado. Tem aplicação em poucos casos.

Exemplos:
Se tivermos uma lista de números e quisermos encontrar algum deles assume-se que a complexidade melhor caso é Ω (1), pois o número pode estar logo na cabeça da lista.

Caso Médio

Representa-se por θ( ).

Este método é o mais difícil de determinar, pois necessita de análise estatística e, portanto, de muitos testes. No entanto, é muito usado, pois é também o que representa mais corretamente a complexidade do algoritmo.

TIPOS DE ANÁLISE

Freqüência de utilização de cada um dos casos:

- *Pior caso – a maioria das vezes;*
- *Caso médio – algumas vezes, mas nem sempre é possível;*
- *Melhor caso – raramente utilizado – é bom para mostrar que o tempo mínimo de um algoritmo é ruim.*

TIPOS DE ANÁLISE

- Em geral damos mais atenção à descoberta do *tempo de execução do pior caso*, ou seja o tempo de execução mais longo para qualquer entrada de tamanho *n*.
- Por que ?

TIPOS DE ANÁLISE

- Por que ?
- Razão 1: É um limite superior sobre o tempo de execução para qualquer entrada ➔ o algoritmo nunca irá demorar mais tempo
- Razão 2: Para alguns algoritmos, o pior caso ocorre com bastante freqüência (ex. database)
- Razão 3: Muitas vezes, o caso médio é tão ruim quanto o pior caso (ex. lugar no subarranjo)

Exemplo: Pesquisa Seqüencial

• Problema: encontrar um valor específico dentro de uma seqüência de valores.

***Algoritmo:** Pesquisa Seqüencial*

1. melhor caso : T(n)=1 (quantidade de comparações)

2. pior caso: T(n)= n

3. caso médio: T(n)=(n+1)/2

T(n)= 1 p(1) + 2 p(2) + 3 p(3) + ...+ n p(n);

P(i) = 1/n se cada registro tem a mesma probabilidade de ser selecionado.

Portanto T(n) = (1/n) (1+2+3+...+n) = (1/n) [(n+1)(n/2)] = (n+1)/2

Soma de uma progressão aritmética: $(a_1 + a_n)(n/2)$.

Observações:

1. *Em análise de algoritmos normalmente nos concentramos no tempo de execução do pior caso, pois esta é a situação crítica na execução de um algoritmo.*

Observações:

2. *Na vida real nós descobrimos que o tempo do caso médio, embora seja bastante útil conhecê-lo, freqüentemente não é melhor que o pior caso. Sem contar que o cálculo do tempo para o caso médio normalmente é bem mais complexo.*

COMPLEXIDADE LOCAL (linha por linha)

A análise de complexidade local de um algoritmo procura estimar seu tempo de execução baseado na quantidade operações primitivas que o algoritmo executa.

COMPLEXIDADE LOCAL (linha por linha)

A análise é realizada para cada linha do algoritmo, contabilizando o número de operações primitivas. Essa análise nos fornece uma função matemática expressando o total de operações com base no tamanho do problema (tamanho n) e o custo (valor constante) de cada operação.

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

```
INSERTION-SORT(A)                                   cost   times
for j ← 2 to n                                      c1     n
    do key ← A[j]                                   c2     n − 1
       ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].   0      n − 1
       i ← j − 1                                    c4     n − 1
       while i > 0 and A[i] > key                   c5     Σ_{j=2}^{n} t_j
           do A[i + 1] ← A[i]                       c6     Σ_{j=2}^{n} (t_j − 1)
              i ← i − 1                             c7     Σ_{j=2}^{n} (t_j − 1)
       A[i + 1] ← key                               c8     n − 1
```


COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

Best case: The array is already sorted.

- Always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n.

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

On average, the key in $A[j]$ is less than half the elements in $A[1 . . j - 1]$ and it's greater than the other half.
$\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 . . j - 1]$ to decide where to drop *key*.
$\Rightarrow t_j = j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

COMPLEXIDADE LOCAL (linha por linha)

Exemplo: *Algoritmo de ordenação por inserção.*

Pior Caso: $T(n) = an^2 + bn + c$

Caso Médio: $T(n) = an^2 + bn + c$

Melhor Caso: $T(n) = an + b$

COMPLEXIDADE ASSINTÓTICA

A análise assintótica de algoritmos é uma forma de estimar o custo (tempo/espaço) de um algoritmo com base no comportamento assintótico de sua função custo.

COMPLEXIDADE ASSINTÓTICA

Em ***análise assintótica*** *as constantes multiplicativas e termos de menor ordem são ignorados e é adotada uma notação matemática específica para representar a complexidade o algoritmo.*

Exemplo: $T(n) = 3n^3 + 2n = \Theta(n^3)$

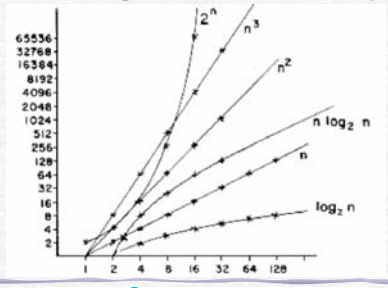
COMPLEXIDADE ASSINTÓTICA

Embora seja possível encontrar outras abordagem técnicas para análise de algoritmos, a análise assintótica é abordagem mais utilizada e encontrada na literatura.

COMPLEXIDADE ASSINTÓTICA

Uma tarefa freqüente em análise assintótica de algoritmos é a comparação de funções matemáticas. Assim, é muito importante a familiarização com muitas dessas funções e as relações de comportamento assintótico entre elas.

Principais Funções: *Ilustração de algumas funções que normalmente aparecem em análise de algoritmos.*



Exemplo: *Considere 5 algoritmos A_1 a A_5, de complexidades diferentes, para resolver um mesmo problema e suponha que uma operação leva 1 ms para ser efetuada.*

n	A_1 $T_1(n)= n$	A_2 $T_2(n)=n\log n$	A_3 $T_3(n)=n^2$	A_4 $T_4(n)=n^3$	A_5 $T_5(n)=2^n$
16	0.016s	0.064s	0.256s	4s	1m4s
32	0.032s	0.16s	1s	33s	49 Dias
512	0.512s	4.6s	4m22s	1 Dia 13h	10^{141} Séculos

Asymptotic notation

O-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

Asymptotic notation

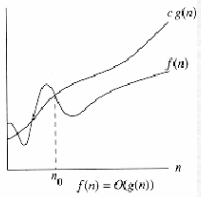
O-notation (upper bounds):

EXAMPLE: $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

Set definition of O-notation

$O(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

- A figura abaixo apresenta as funções $f(n)$ e $g(n)$ onde $f(n) = O(g(n))$



- Para todos os valores de n à direita de n_0, o valor de $f(n)$ está em ou abaixo de $g(n)$.

Set definition of O-notation

EXAMPLE: $2n^2 \in O(n^3)$

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$

Also,

n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Macro substitution

Convention: A set in a formula represents an anonymous function in the set.

EXAMPLE: $f(n) = n^3 + O(n^2)$

means

$f(n) = n^3 + h(n)$

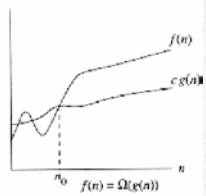
for some $h(n) \in O(n^2)$.

Ω–notation (lower bounds)

O-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

$\Omega(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

- A figura abaixo apresenta as funções $f(n)$ e $g(n)$ onde $f(n) = \Omega(g(n))$



- Para todos os valores de n à direita de n_o, o valor de $f(n)$ está em ou acima de $g(n)$.

$\Omega(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

EXAMPLE: $\sqrt{n} = \Omega(\lg n)$

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$

Also,

n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Θ-notation

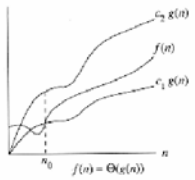
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.

Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

EXAMPLE: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

- A figura abaixo apresenta as funções $f(n)$ e $g(n)$ onde $f(n) = \Theta(g(n))$



- Para todos os valores de n à direita de n_o, o valor de $f(n)$ reside em $c_1 g(n)$ ou acima dele, e em $c_2 g(n)$ ou abaixo deste valor

Notação *o*

- A notação *o* indica um limite superior que não é assintóticamente restrito.
- Para uma dada função $g(n)$ definimos $o(g(n))$ o conjunto de funções:

$o(g(n)) = \{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_o > 0$ tal que $0 \le f(n) < c\,g(n)$ para todo $n \ge n_o\}$

Notação *o*

- As definições da notação *O* e da notação *o* são semelhantes. A principal diferença é que em $f(n) = O(g(n))$, o limite $0 \le f(n) \le c\,g(n)$ se mantém válido para alguma constante $c > 0$,
- Mas, em $f(n) = o(g(n))$, o limite $0 \le f(n) < c\,g(n)$ é válido para todas as constantes $c > 0$.
- Note que isto é exatamente o mesmo que a definição de *O*, exceto que "alguma constante" foi trocado para "para todas".

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

Notação ω

- A notação ω indica um limite inferior que não é assintóticamente restrito.
- Para uma dada função $g(n)$ definimos $\omega(g(n))$ o conjunto de funções:

$\omega(g(n))=\{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_o > 0$ tal que $0 \leq c\,g(n) < f(n)$ para todo $n \geq n_o\}$

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

A way to compare "sizes" of functions:

O	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

Comparisons of functions

Relational properties:

Transitivity:
$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
Same for O, Ω, o, and ω.

Reflexivity:
$f(n) = \Theta(f(n))$.
Same for O and Ω.

Symmetry:
$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Comparisons:

- $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

Asymptotic notation in equations

When on right-hand side: $O(n^2)$ stands for some anonymous function in the set $O(n^2)$.

$2n^2+3n+1 = 2n^2+\Theta(n)$ means $2n^2+3n+1 = 2n^2+f(n)$ *for some* $f(n) \in \Theta(n)$. In particular, $f(n) = 3n + 1$.

Limite Superior (Upper Bound)

*Dado um problema, por exemplo, a multiplicação de duas matrizes quadradas de ordem n (n*n). Conhecemos um algoritmo para resolver este problema(pelo método trivial) de complexidade $O(n^3)$. Sabemos assim que a complexidade deste problema não deve superar $O(n^3)$, uma vez que existe um algoritmo desta complexidade que o resolve.*

Limite Superior (Upper Bound)

O limite superior de um algoritmo pode mudar se alguém descobrir um algoritmo melhor. Isso de fato aconteceu com o algoritmo de Strassen que é de $O(n^{\log 7})$. Assim o limite superior do problema de multiplicação de matrizes passou a ser $O(n^{\log 7})$. Outros pesquisadores melhoraram ainda este resultado. Atualmente, o melhor resultado é o de Coppersmith e Winograd de $O(n^{2.376})$.

Limite Superior (Upper Bound)

O limite superior de um algoritmo é parecido com o recorde mundial de uma modalidade de atletismo. Ela é estabelecida pelo melhor atleta (algoritmo) do momento. Assim como o recorde mundial o limite superior pode ser melhorado por um algoritmo (atleta) mais veloz.

Limite Inferior (Lower Bound)

Às vezes é possível demonstrar que para um dado problema, qualquer que seja o algoritmo a ser usado, o problema requer pelo menos um certo número de operações. Essa complexidade é chamada Limite Inferior (Lower Bound). O limite inferior depende do problema, mas não do particular algoritmo.

Usamos a letra Ω para denotar um limite inferior.

Limite Inferior (Lower Bound)

Para o problema de multiplicação de matrizes de ordem n, apenas para ler os elementos das duas matrizes de entrada leva $O(n^2)$. Assim uma cota inferior trivial é $\Omega(n^2)$.

Limite Inferior (Lower Bound)

Na analogia anterior, um limite inferior de uma modalidade de atletismo não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta.

Um limite inferior trivial para os 100 metros seria o tempo que a velocidade da luz leva a percorrer 100 metros no vácuo.

Limite Inferior (Lower Bound)

Se um algoritmo tem uma complexidade que é igual ao limite inferior do problema então o algoritmo é ótimo.

O algoritmo de CopperSmith e Winograd é de $O(n^{2.376})$, mas o limite inferior é de $\Omega(n^2)$. Portanto, não é ótimo. Pode ser que este limite superior possa ainda ser melhorado.

Exercícios

Prove que, para todas funções $f, g : \mathbf{N} \to \mathbf{N}$,

(a) $af(n) + b = O(f(n))$ $(a, b$ constantes$)$.

Exercícios

Prove que, para todas funções $f, g : \mathbf{N} \to \mathbf{N}$,

(a) $af(n) + b = O(f(n))$ $(a, b$ constantes$)$.

Solução: Para resolver este item, precisamos assumir primeiro que $f(n)$ é assintoticamente positiva, ou seja, que existe $n_1 > 0$ tal que $f(n) > 0$ para todo $n \geq n_1$. Segundo, que $af(n) + b$ é assintoticamente positiva, o que é equivalente a assumir que $a > 0$ e $b > -a$, dado que $f(n)$ é assintoticamente positiva.

Queremos mostrar que existem $c > 0$ e $n_0 > 0$ tais que $0 \leq af(n) + b \leq cf(n)$ para todo $n \geq n_0$. Para todo $n \geq n_1$, temos que $f(n) \geq 1$. Isso implica que $af(n) + b \geq a + b \geq 0$ para todo $n \geq n_1$. Tomando-se $c = a + |b|$, temos que, para todo $n \geq n_1$,

$$af(n) + b \leq af(n) + |b| \leq af(n) + |b|f(n) = cf(n),$$

pois nesse caso $f(n) \geq 1$. Portanto, tomando-se $c = a + |b|$ e $n_0 = n_1$, temos que $0 \leq af(n) + b \leq cf(n)$ para todo $n \geq n_0$, o que implica que $af(n) + b = O(f(n))$.

Exercícios

Prove que, para todas funções $f, g : \mathbf{N} \to \mathbf{N}$,

(b) $O(f(n)) + O(f(n)) = O(f(n))$.

Exercícios

Prove que, para todas funções $f, g : \mathbf{N} \to \mathbf{N}$,

(b) $O(f(n)) + O(f(n)) = O(f(n))$.

Solução: Para provar este item, devemos mostrar que, para quaisquer funções $g(n), h(n) = O(f(n))$, temos que $g(n) + h(n) = O(f(n))$.

Sejam $g(n), h(n) = O(f(n))$. Como $g(n), h(n) = O(f(n))$, existem constantes $c_1, c_2 > 0$ e $n_1, n_2 > 0$ tais que $0 \leq g(n) \leq c_1 f(n)$, para todo $n \geq n_1$, e $0 \leq h(n) \leq c_2 f(n)$, para todo $n \geq n_2$. Tomando-se $c = c_1 + c_2$ e $n_0 = \max\{n_1, n_2\}$, temos que $0 \leq g(n) + h(n) \leq cf(n)$, para todo $n \geq n_0$. Ou seja, $g(n) + h(n) = O(f(n))$.

Exercícios

Sejam $f(n)$ e $g(n)$ funções assintoticamente positivas. Prove ou desprove cada uma das seguintes conjecturas.

(a) $f(n) = O(g(n))$ implica que $\log(f(n)) = O(\log(g(n)))$, onde $\log(g(n)) > 0$ e $f(n) \geq 1$ para todo n suficientemente grande.

Exercícios

(a) $f(n) = O(g(n))$ implica que $\log(f(n)) = O(\log(g(n)))$, onde $\log(g(n)) > 0$ e $f(n) \geq 1$ para todo n suficientemente grande.

Solução: Vamos provar que, **se as funções forem inteiras**, a conjectura é verdadeira.
Suponha que $f(n) = O(g(n))$. Isto significa que existem constantes $c > 0$ e $n_0 > 0$ tais que $f(n) \leq cg(n)$ para todo $n \geq n_0$. Podemos assumir que $c \geq 2$ (caso $c < 2$, tome $c = 2$ e, evidentemente, a afirmação acima continua válida).
Como $\log x$ é uma função crescente, temos que, para $n \geq n_0$,

$$\log(f(n)) \leq \log(cg(n)) = \log c + \log(g(n)).$$

Além disso, como $\log(g(n)) > 0$ para todo n suficientemente grande e $g(n)$ é uma função inteira, então $g(n) \geq 2$ para, digamos, $n \geq n_1$. Ou seja, existe $n_1 > 0$ tal que $\log(g(n)) \geq 1$ para todo $n \geq n_1$.
Tomemos então $c_2 := 2\log c$ e $n_2 := \max\{n_0, n_1\}$. Observe que $c_2 \geq 2$ pois $c \geq 2$. Para todo $n \geq n_2$, temos que

$$\log(f(n)) \leq \log c + \log(g(n)) \leq \frac{c_2}{2} + \frac{c_2}{2}\log(g(n)) \leq c_2 \log(g(n)).$$

A conjectura é falsa se omitirmos a hipótese de que as funções são inteiras. Veja por exemplo o caso das funções $f(n) = 2 + 2/n$ e $g(n) = 1 + 1/n$. Claramente $f(n) = O(g(n))$, porém $\lim_{n\to\infty} \log(f(n)) = 1$ e $\lim_{n\to\infty} \log(g(n)) = 0$.

Exercícios

Sejam $f(n)$ e $g(n)$ funções assintoticamente positivas. Prove ou desprove cada uma das seguintes conjecturas.

(b) $f(n) = O(g(n))$ implica que $2^{f(n)} = O(2^{g(n)})$.

Exercícios

Sejam $f(n)$ e $g(n)$ funções assintoticamente positivas. Prove ou desprove cada uma das seguintes conjecturas.

(b) $f(n) = O(g(n))$ implica que $2^{f(n)} = O(2^{g(n)})$.

Solução: A conjectura acima é falsa. Tome, por exemplo, $f(n) = 2n$ e $g(n) = n$. Evidentemente $f(n) = O(g(n))$. No entanto, não é verdade que $2^{2n} = O(2^n)$. De fato, suponha que $2^{2n} = O(2^n)$. Isso significaria que existem constantes $c > 0$ e $n_0 > 0$ tais que, para todo $n \geq n_0$,

$$2^{2n} \leq c2^n.$$

Dividindo os dois lados por 2^n, obtemos que $2^n \leq c$ para todo $n \geq n_0$, o que é um absurdo.

Recorrência

Recorrência =

= "fórmula" que define uma função em termos d'ela mesma

= algoritmo recursivo que calcula uma função

Exemplo 1

$$T(1) = 1$$
$$T(n) = T(n-1) + 3n + 2 \quad \text{para } n = 2, 3, 4, \ldots$$

Define função T sobre inteiros positivos:

n	$T(n)$
1	1
2	9
3	20
4	34
5	51
6	71

Resolver uma recorrência =
= obter uma "fórmula fechada" para $T(n)$

Método da **substituição**:
"chute" fórmula e verifique por indução

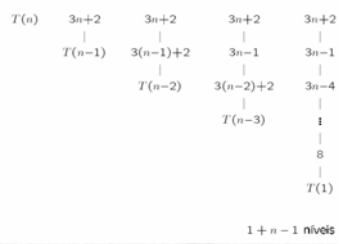
Exemplo 1 (continuação)
Eu acho que $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 4$.

Exemplo 1 (continuação)
Eu acho que $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 4$.

Verificação:
Se $n = 1$ então $T(n) = 1 = \frac{3}{2} + \frac{7}{2} - 4$.
Tome $n \geq 2$ e suponha que a fórmula está certa para $n - 1$:

$$\begin{aligned} T(n) &= T(n-1) + 3n + 2 \\ &\overset{\text{hi}}{=} \tfrac{3}{2}(n-1)^2 + \tfrac{7}{2}(n-1) - 4 + 3n + 2 \\ &= \tfrac{3}{2}n^2 - 3n + \tfrac{3}{2} + \tfrac{7}{2}n - \tfrac{7}{2} - 4 + 3n + 2 \\ &= \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 4\,. \end{aligned}$$

Como adivinhei fórmula fechada?
Árvore da recorrência:



$$\begin{aligned} T(n) &= (3n+2) + (3n-1) + \cdots + 8 + T(1) \\ &= \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 4 \end{aligned}$$

Exemplo 2

$$T(1) = 1$$
$$T(n) = 2T(n/2) + 7n + 2 \quad \text{para } n = 2, 3, 4, 5, \ldots$$

Não é uma recorrência! Não faz sentido!

Exemplo 3

$$G(1) = 1$$
$$G(n) = 2G(n/2) + 7n + 2 \quad \text{para } n = 2, 4, \ldots, 2^i, \ldots$$

n	$G(n)$
1	1
2	18
4	66
8	190
16	494

Fórmula fechada: $G(n) = ?$

Acho que G é da forma $n \lg n$

n	$G(n)$	$6n \lg n$	$7n \lg n$	$8n \lg n$	n^2
1	1	0	0	0	1
2	18	12	14	16	4
4	66	48	56	64	16
8	190	144	168	192	64
16	494	384	448	512	256
32	1214	960	1120	1280	1024
64	2878	2304	2688	3072	4096
128	6654	5376	6272	7168	16384
256	15102	12288	14336	16384	65536

Acho que a fórmula fechada é

$$G(n) = 7n \lg n + 3n - 2$$

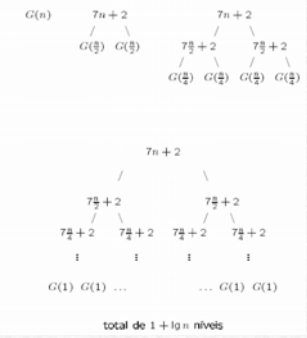
para $n = 1, 2, 4, 8, 16, \ldots$

Prova:
Se $n = 1$ então $G(n) = 1 = 7 \cdot 1 \lg 1 + 3 \cdot 1 - 2$.
Se $n \geq 2$ então

$$\begin{aligned} G(n) &= 2G(\tfrac{n}{2}) + 7n + 2 \\ &\overset{\text{hi}}{=} 2\,(7\tfrac{n}{2} \lg \tfrac{n}{2} + 3\tfrac{n}{2} - 2) + 7n + 2 \\ &= 7n(\lg n - 1) + 3n - 4 + 7n + 2 \\ &= 7n \lg n - 7n + 3n - 2 + 7n \\ &= 7n \lg n + 3n - 2 \end{aligned}$$

Como adivinhei fórmula fechada?

Árvore da recorrência:



total de $1 + \lg n$ níveis

nível	soma no nível
0	$7n + 2$
1	$7n + 4$
2	$7n + 8$
⋮	⋮
$k-1$	$7n + 2^k$
k	$2^k G(1)$

$$n = 2^k$$

$$\begin{aligned} G(n) &= 7n + 2^1 + 7n + 4^2 + \cdots + 7n + 2^{\lg n} \\ &\quad + 2^{\lg n} G(1) \\ &= 7n \lg n + (2 + 4 + \cdots + 2^{\lg n}) + 2^{\lg n} \\ &= 7n \lg n + 2 \cdot 2^{\lg n} - 2 + n \\ &= 7n \lg n + 2n - 2 + n \\ &= 7n \lg n + 3n - 2 \end{aligned}$$

Lembrete: $x^0 + \cdots + x^k = \frac{x^{k+1} - 1}{x - 1}$

Exemplo 3 (continuação)
É mais fácil mostrar que $G(n) = O(n \lg n)$.

Vou provar que $G(n) \leq 9n \lg n$
quando $n = 2, 4, 8, 16, \ldots, 2^i, \ldots$

Prova: Se $n = 2$, $G(n) = 18 = 9 \cdot 2 \cdot \lg 2$.
Se $n \geq 4$,

$$\begin{aligned} G(n) &= 2G(n/2) + 7n + 2 \\ &\overset{\text{hi}}{\leq} 2 \cdot 9(n/2) \lg(n/2) + 7n + 2 \\ &= 9n\,(\lg n - 1) + 7n + 2 \\ &= 9n \lg n - 2n + 2 \\ &< 9n \lg n \quad \text{(pois } n > 1\text{)} \end{aligned}$$

Da linha 1 para a linha 2, a hipótese de indução vale pois $2 \leq n/2 < n$.

Classe O da solução de uma recorrência

Não faço questão de solução *exata*:
basta solução aproximada

Exemplo (n é potência de 2):

$$\begin{aligned} G(1) &= 1 \\ G(n) &= 2G(n/2) + 7n + 2 \quad \text{para } n \geq 2 \end{aligned}$$

Solução exata (n é potência de 2):

$$G(n) = 7n \lg n + 3n - 2$$

$$\begin{aligned} G(1) &= 1 \\ G(n) &= 2G(n/2) + 7n + 2 \quad \text{para } n \geq 2 \end{aligned}$$

Solução aproximada:

$$G(n) = \mathrm{O}(n \lg n)$$

Em geral, é mais fácil obter e provar solução aproximada que solução exata

Dica prática

A solução da recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + 7n + 2 \quad \text{para } n = 2, 3, 4, 5, \ldots \end{aligned}$$

está na mesma classe Θ que a solução de

$$\begin{aligned} T'(1) &= 1 \\ T'(n) &= 2T'(n/2) + n \quad \text{para } n = 2^1, 2^2, 2^3, \ldots \end{aligned}$$

e na mesma classe Θ que a solução de

$$\begin{aligned} T''(4) &= 10 \\ T''(n) &= 2T''(n/2) + n \quad \text{para } n = 2^3, 2^4, 2^5, \ldots \end{aligned}$$

Recorrências com O do lado direito

A "recorrência"

$$T(n) = 2T(n/2) + \mathrm{O}(n)$$

representa todas as recorrências da forma $T(n) = 2T(n/2) + F(n)$ em que $F(n) = \mathrm{O}(n)$

Melhor:
representa todas as recorrências do tipo

$$\begin{aligned} T'(n) &= a \quad \text{para } n = n_0 - 1 \\ T'(n) &\leq 2T'(\lfloor n/2 \rfloor) + cn \quad \text{para } n \geq n_0 \end{aligned}$$

qaisquer que sejam a, $c > 0$ e $n_0 > 0$

Também representa todas as do tipo

$$\begin{aligned} T''(n) &= a \quad \text{para } n = 2^{k-1} \\ T''(n) &\leq 2T''(n/2) + cn \quad \text{para } n = 2^k, 2^{k+1}, \ldots \end{aligned}$$

quaisquer que sejam a, $c > 0$ e $k > 0$

As soluções exatas vão depender de a, c, n_0, k, mas todas estarão na mesma classe O (especificamente, em $\mathrm{O}(n \lg n)$)

Exercício A
Seja T a função definida pela recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + 2n - 2 \quad \text{para } n = 2, 3, 4, 5, \ldots \end{aligned}$$

Verifique que a recorrência é honesta, ou seja, de fato define uma função. A partir da árvore da recorrência, adivinhe uma boa delimitação assintótica para $T(n)$; dê a resposta em notação O. Prove a delimitação pelo método da substituição.

Exercício B
Resolva a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-2) + 2n + 1 \quad \text{para } n = 2, 3, 4, 5, \ldots \end{aligned}$$

Desenhe a árvore da recorrência. Dê a resposta em notação O.

Exercício C
Resolva a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 2 \\ T(n) &= T(n-2) + 2n + 1 \quad \text{para } n = 3, 4, 5, 6, \ldots \end{aligned}$$

Exercício D
Resolva a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/2) + 1 \quad \text{para } n = 2, 3, 4, 5, \ldots \end{aligned}$$

Exercício E
Resolva a recorrência

$$T(1) = 1$$
$$T(n) = T(\lfloor n/2 \rfloor) + 1 \quad \text{para } n = 2, 3, 4, 5, \ldots$$

Exercício F
Resolva a recorrência

$$T(1) = 1$$
$$T(n) = T(\lfloor n/2 \rfloor) + n \quad \text{para } n = 2, 3, 4, 5, \ldots$$

Exercício G
Resolva a recorrência

$$T(1) = 1$$
$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad \text{para } n = 2, 3, 4, 5, \ldots$$

Exercício H
Resolva a recorrência

$$T(1) = 1$$
$$T(n) = 2T(\lceil n/2 \rceil) + n \quad \text{para } n = 2, 3, 4, 5, \ldots$$

Recorrências

- **Exemplos:**

	Solução
$T(n) = \begin{cases} 1 & se\ \ n = 1 \\ T(n-1)+1 & se\ \ n > 1 \end{cases}$	$T(n)=n$
$T(n) = \begin{cases} 1 & se\ \ n = 1 \\ 2T(n/2)+n & se\ \ n \geq 1 \end{cases}$	$T(n)=n \lg n+n$
$T(n) = \begin{cases} 0 & se\ \ n = 2 \\ T(\sqrt{n})+1 & se\ \ n > 2 \end{cases}$	$T(n)= \lg \lg n$
$T(n) = \begin{cases} 1 & se\ \ n = 1 \\ T(n/3)+T(2n/3)+n & se\ \ n > 1 \end{cases}$	$T(n)= n \lg n$

Método Mestre

- Fornece um processo de "livro de receitas" para resolver recorrências da forma:

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

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

- A recorrência acima, descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas de tamanho n/b.
- O custo de dividir o problema e combinar os resultados é descrito por $f(n)$

Método Mestre

- O custo de dividir o problema e combinar os resultados é descrito por $f(n)$
- Interpretamos n/b com o significado de $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$.
- Então, $T(n)$ pode ser limitado assintóticamente como veremos a seguir . . .

Método Mestre

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 $T(n) = \Theta(f(n))$.

Método Mestre

- Nos três casos estamos comparando a função $f(n)$ com a função $n^{\log_b a}$
- A solução para a recorrência é dada pela maior das duas funções:
  - No caso 1 a função $n^{\log_b a}$ é maior, então a solução será $T(n)=\Theta(n^{\log_b a})$
  - No caso 3 a função $f(n)$ é maior, então a solução será $T(n) = \Theta(f(n))$
  - No caso 2 as funções são de mesma dimensão, sendo introduzido um fator $lg\ n$ e a solução será $T(n) = \Theta(n^{\log_b a}\ lg\ n) = \Theta(f(n)\ lg\ n)$

Método Mestre

- Atenção! O teorema não cobre todos os casos possíveis.
- Apenas aqueles em que $f(n)$ é menor do que $n^{\log_b a}$ por um fator polinomial e aqueles em que $f(n)$ é maior do que $n^{\log_b a}$ por um fator polinomial.
- Lacunas entre os casos 1 e 2 e entre os casos 2 e 3 ⇨ o método não poderá ser usado.

Resumo

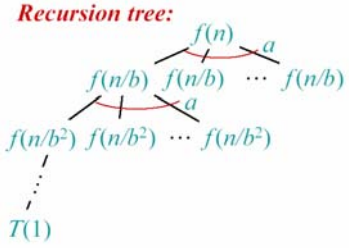
- Definição de recorrências
- 2 métodos principais para resolver recorrências:
  - sustituição retrógradas
  - método mestre
- Outros métodos:
  - Método da substituição
  - Árvore de recursão

Resumo

- Método Mestre:
  - Usado para muitas recorrências "dividir e conquistar" da forma
    $$T(n) = aT(n/b) + f(n),$$
    onde $a \geq 1$, $b > 1$, e $f(n) > 0$
  - Baseado no *teorema mestre*
  - Compara $n \log_b a$ versus $f(n)$

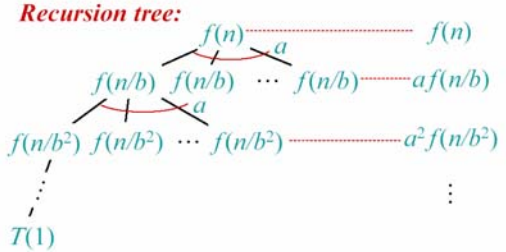
Idea of master theorem

Recursion tree:



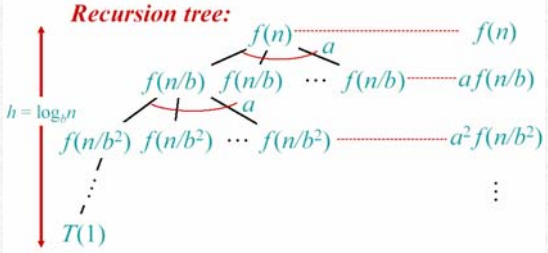
Idea of master theorem

Recursion tree:



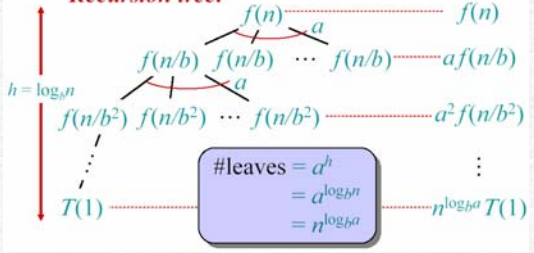
Idea of master theorem

Recursion tree:



Idea of master theorem

Recursion tree:



Ex. $T(n) = 4T(n/2) + n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$

Case 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

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

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

Case 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n).$

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3).$

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.