Funções

Antonio Alfredo Ferreira Loureiro

loureiro@dcc.ufmg.br

http://www.dcc.ufmg.br/~loureiro

Sumário

- Introdução
- Sequência como função
- Função como codificação/decodificação de bits
- Autômato finito
- Função injetiva e função hash
- Funções sobrejetiva, bijetiva e inversa
- Princípio da casa de pombo
- Composição de funções
- Função de complexidade

Introdução

- Sejam dois conjuntos A e B. Suponha que cada elemento de A possa ser associado a um elemento específico de B. A relação entre elementos de A e B é chamada de função.

- Funções são normalmente representadas por uma única letra como, por exemplo, f, g, h, F, G. Funções especiais são denotadas por strings, como log, exp e sin.

Introdução

- Definição: A função f de um conjunto X para um conjunto Y é uma relação entre elementos de X e elementos de Y com a propriedade que cada elemento de X está relacionado a um único elemento de Y.
  - A notação $f : X \to Y$ significa que f é uma função de X para Y.
  - O conjunto X é chamado de domínio e o conjunto Y de co-domínio.
  - Faixa de f ou imagem de X para f: conjunto de todos os valores que f pode assumir:

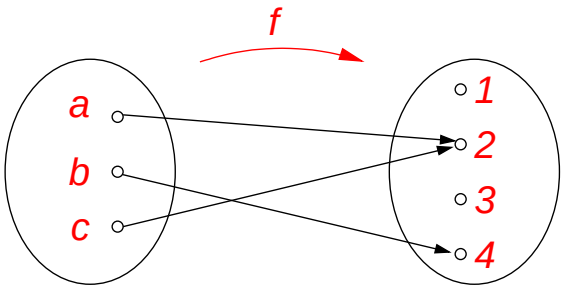
$$\text{Faixa de } f\text{: } \{y \in Y | y = f(x), \text{ para algum } x \in X\}$$

  - Imagem inversa de y = $\{x \in X | f(x) = y\}$

- Implicação:
  - ➜ Cada elemento de X pode ser associado a um e somente um elemento de Y.

Introdução

- Sejam os conjuntos finitos X e Y definidos da seguinte forma: $X = \{a, b, c\}$ e $Y = \{1, 2, 3, 4\}$. A função $f : X \rightarrow Y$ pode ser definida por um diagrama, como por exemplo:



  - Domínio de f = $\{a, b, c\}$;
    Co-domínio de f = $\{1, 2, 3, 4\}$.
  - $f(a) = 2$; $f(b) = 4$; $f(c) = 2$.
  - Faixa de f = $\{2, 4\}$.
  - Imagem inversa de 1 = $\emptyset$;
    Imagem inversa de 2 = $\{a, c\}$;
    Imagem inversa de 3 = $\emptyset$;
    Imagem inversa de 4 = $\{b\}$;
  - A função f representada como um conjunto de pares ordenados = $\{(a, 2), (b, 4), (c, 2)\}$.

Introdução

- Exemplos de funções:
  - $f : x \to x^2$.
    $f : \mathbb{R} \to \mathbb{R}$ é a função quadrado.
  - $f : n \to n + 1$.
    $f : \mathbb{Z} \to \mathbb{Z}$ é a função sucessor.
  - $f : r \to 2$.
    $f : \mathbb{Q} \to \mathbb{N}$ é a função constante.

- Definição (Igualdade de funções): Sejam f e g funções definidas de X em Y. Então,

$$f = g, \text{ sse } f(x) = g(x), \forall x \in X.$$

Introdução

- Exemplos:

  Sejam f e g funções definidas para todo $x \in \mathbb{R}$.

  (a) $f(x) = |x|$ e $g(x) = \sqrt{x^2}$. $\quad f = g$?

  Sim. $|x| = \sqrt{x^2}, \forall x \in \mathbb{R}$.

  (b) Sejam as funções

$$f + g: \ \mathbb{R} \to \mathbb{R} \quad \text{e}$$
$$g + f: \ \mathbb{R} \to \mathbb{R}$$

  definidas como:

$$(f+g)(x) = f(x) + g(x), \quad \forall x \in \mathbb{R} \quad \text{e}$$
$$(g+f)(x) = g(x) + f(x), \quad \forall x \in \mathbb{R}.$$

  $f + g = g + f$?

$$\begin{aligned} (f+g)(x) &= f(x) + g(x) && \text{definição de } f+g \\ &= g(x) + f(x) && \text{comutatividade da adição} \\ &= (g+f)(x) && \text{definição de } g+f \end{aligned}$$

Sequências como funções

- Uma sequência é uma função definida no conjunto dos inteiros a partir de um valor específico.
- Por exemplo, a sequência

$$1, \quad -\frac{1}{2}, \quad \frac{1}{3}, \quad -\frac{1}{4}, \ldots, \frac{(-1)^n}{n+1}, \ldots$$

pode ser definida como uma função f dos inteiros não-negativos para os números reais, isto é,

$$\begin{array}{rcl} 0 & \rightarrow & 1 \\ 1 & \rightarrow & -\frac{1}{2} \\ 2 & \rightarrow & \frac{1}{3} \\ & \vdots & \\ n & \rightarrow & \frac{(-1)^n}{n+1} \end{array}$$

Formalmente, $f : \mathbb{Z}^0 \rightarrow \mathbb{R}$, onde

$$f(n) = \frac{(-1)^n}{n+1}$$

Sequências como funções

- A função que define essa sequência é única?
  Não. Por exemplo, $f : \mathbb{N}^* \to \mathbb{R}$, onde

$$f(n) = \frac{(-1)^{n+1}}{n}$$

Funções de codificação/decodificação de bits

- Mensagens transmitidas através de canais de comunicação são frequentemente codificadas de formas especiais para reduzir a possibilidade de serem corrompidas devido à interferência de ruídos nas linhas de transmissão.

- Uma possível forma de codificação é repetir cada bit três vezes. Assim, a mensagem

  `00101111`

  seria codificada como

  `000000111000111111111111`

Funções de codificação/decodificação de bits

- Seja $\Sigma = \{0, 1\}$. Σ^* é o conjunto de todos os strings que podem ser formados com 0 e 1. Seja L o conjunto de todos os strings formados de Σ que possuem triplas de bits idênticos.

- Função de codificação:
  $E : \Sigma^* \rightarrow L$, onde $E(s)$ é o string obtido de s trocando cada bit de s pelo mesmo bit repetido três vezes.

- Função de decodificação:
  $D : L \rightarrow \Sigma^*$, onde $D(t)$ é o string obtido de t trocando cada três bits consecutivos idênticos de t por uma única cópia desse bit.

- Qual é a capacidade de correção deste método?
  ➜ Um bit trocado em cada tripla de bits idênticos enviados pela origem.

Função da distância de Hamming

- A função da distância de Hamming é usada em Teoria da Informação. Essa função fornece o número de bits que dois strings de tamanho n diferem posição a posição.

- Seja $\Sigma = \{0, 1\}$ e Σ^n o conjunto de todos os strings de tamanho n. A função $H : \Sigma^n \times \Sigma^n \to \mathbb{Z}^+$ é definida da seguinte forma:

  Para cada par de strings $(s, t) \in \Sigma^n \times \Sigma^n$, $H(s, t)$ é o número de posições nos quais s e t têm valores diferentes.

- Exemplo:

  Seja $n = 5$.
  - $H(11000, 00000) = 2$
  - $H(11111, 00000) = 5$
  - $H(10001, 01111) = 4$

Funções booleanas

- Definição: Uma função Booleana f é uma função cujo domínio é o conjunto de todas n-tuplas ordenadas de 0's e 1's e cujo co-domínio é $\{0, 1\}$.

  Formalmente, o domínio de uma função Booleana pode ser descrito como o produto Cartesiano de n cópias do conjunto $\{0, 1\}$, que é representado por $\{0, 1\}^n$. Logo,

$$f : \{0, 1\}^n \rightarrow \{0, 1\}.$$

- Que tipo de circuito lógico é definido por este tipo de função?
  - ➜ Circuito combinatório, onde o valor da saída só depende dos valores da entrada.

Funções booleanas

- Exemplo: $f(x_1, x_2, x_3) = (x_1 + x_2 + x_3) \bmod 2$

Entrada			Saída
x_1	x_2	x_3	$(x_1 + x_2 + x_3) \bmod 2$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Verificando se uma função é bem-definida

- Seja uma função f definida do conjunto dos racionais para o conjunto dos inteiros, ou seja, $f : \mathbb{Q} \to \mathbb{Z}$, sendo que

$$f : \frac{m}{n} \to m, \forall \text{ inteiros } m,\ n \text{ e } n \neq 0$$

- A função f é bem-definida?
  - ➜ Não.

$$\begin{aligned} \frac{1}{2} &= \frac{3}{6} \\ \therefore f(\tfrac{1}{2}) &= f(\tfrac{3}{6}) \end{aligned}$$

Mas,

$$\begin{aligned} f(\tfrac{1}{2}) &= 1 \\ f(\tfrac{3}{6}) &= 3 \\ \therefore f(\tfrac{1}{2}) &\neq f(\tfrac{3}{6}) \end{aligned}$$

Função $\times$ Estado

- Os problemas em Ciência da Computação podem ser vistos, de uma forma geral, conforme o diagrama abaixo:

- Em muitos casos, os valores das saídas dependem não somente dos valores da entrada, mas também do estado corrente do sistema. Neste caso, a saída não pode ser representada por uma função somente da entrada.

- Alguns dos sistemas onde isso ocorre frequentemente são:
  - Sistemas reativos
  - Sistemas concorrentes
  - Sistemas distribuídos
  - Sistemas digitais

Autômato finito

- Um autômato finito (AF), em inglês
  - "Finite Automaton" ou
  - "Finite State Automaton" ou
  - "Finite State Machine"

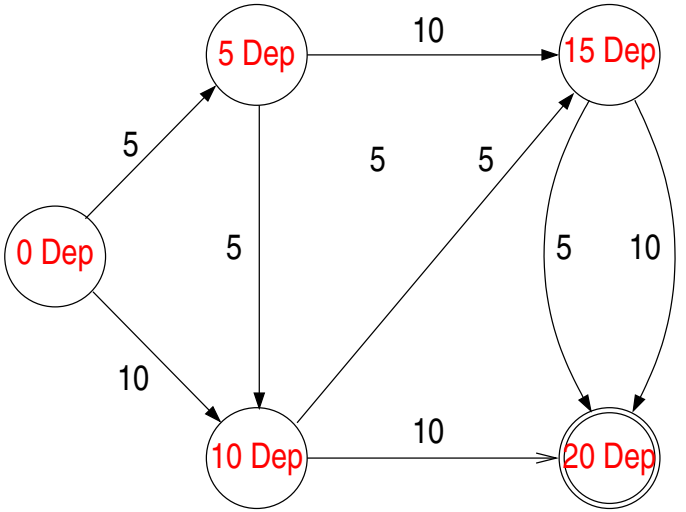
  é um modelo matemático de um sistema que possui entradas e saídas discretas.

- O sistema pode estar num estado dentre um conjunto finito de estados ou configurações.

- O estado do sistema "sumariza" a informação relacionada com entradas passadas que é necessária para determinar o comportamento do sistema em entradas subsequentes.

Autômato finito

- Exemplo: O mecanismo de controle de um elevador não precisa “lembrar” todas as requisições anteriores de serviço, mas somente o andar corrente, a direção de movimento e as requisições pendentes.
- Computador digital: pode ser modelado por um autômato finito.
- Exemplo: máquina de doce que custa 20 centavos e aceita moedas de 5 e 10 centavos



Autômato finito

- Definição: um autômato finito é uma quintupla $A = (I, S, s_0, Q, N)$, onde

  I: alfabeto de entrada para o autômato, ou seja, conjunto de símbolos válidos.

  S: conjunto de estados do autômato.

  s_0: estado inicial do autômato, onde $s_0 \in S$.

  Q: conjunto de estados finais, onde $Q \subseteq S$.

  N: função de transição ou de próximo estado, onde $N : S \times I \rightarrow S$.

- Observação importante: essa definição define um autômato determinístico.

Autômato finito

- Seja um autômato $A = (I, S, s_0, Q, N)$, onde

  $I =$ $\{0, 1\}$

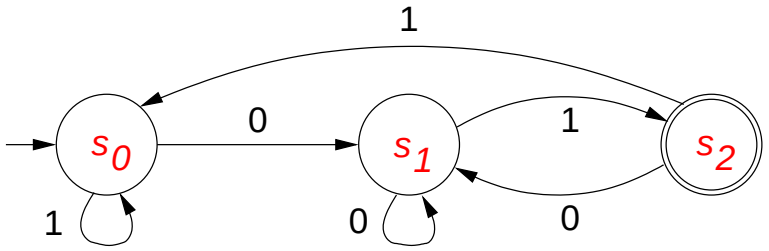
  $S =$ $\{s_0, s_1, s_2\}$

  $s_0 =$ estado inicial do autômato, onde $s_0 \in S$.

  $Q =$ $\{s_2\}$, onde $Q \subseteq S$.

  $N =$ $\{(s_0, 0, s_1), (s_0, 1, s_0), (s_1, 0, s_1), (s_1, 1, s_2), (s_2, 0, s_1), (s_2, 1, s_0)\}$.

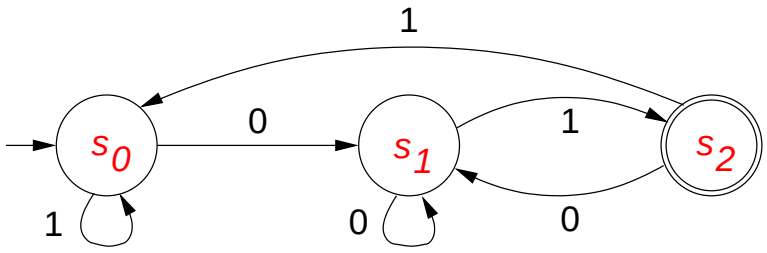
- A operação de um autômato finito é normalmente descrito por um diagrama de transição de estados.



Autômato finito

- Uma outra forma de representar a operação do autômato (função de próximo estado N) é através da tabela do "próximo estado."

Estado		Entrada 0	Entrada 1
⟨Inicial⟩	s_0	s_1	s_0
	s_1	s_1	s_2
⟨Final⟩	s_2	s_1	s_0



Autômato finito com saída

- Uma limitação do autômato finito apresentado é que a saída é restrita a um sinal binário SIM/NÃO que indica se a entrada é aceita ou não, respectivamente.

- Existem dois modelos que consideram um alfabeto de saída diferente de sim/não (os dois modelos são equivalentes):
  - Máquina de Moore (Moore machine)
  - Máquina de Mealy (Mealy machine)

- Máquina de Moore:
  é uma sêxtupla $(I, S, s_0, N, \Delta, \lambda)$, onde I, S, s_0 e N são como antes. Δ é o alfabeto de saída para o autômato e λ é a função de saída, onde $\lambda : S \rightarrow \Delta$.
  ➜ Saída é associada com o estado.

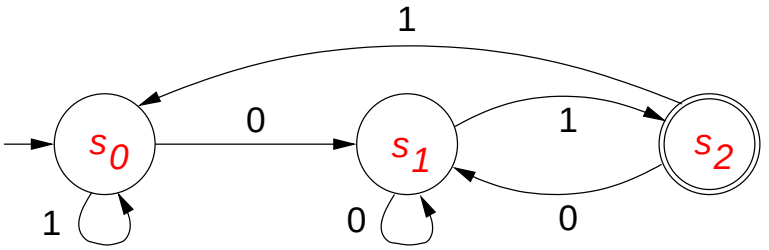
- Máquina de Mealy:
  é uma sêxtupla $(I, S, s_0, N, \Delta, \lambda)$, onde todos os elementos da tupla são idênticos aos da Máquina de Moore exceto que λ é definida de $S \times I \rightarrow \Delta$.
  ➜ Saída é associada com a transição.

Linguagem aceita por um autômato

- Definição: Seja A um AF com alfabeto de entrada I. Seja I^* o conjunto de todos os strings sobre I e seja w um string em I^*. Diz-se que w **é aceito por** A sse A começa no estado inicial e após todos os símbolos de w serem fornecidos em sequência para A, o autômato pára num estado final.

- A linguagem aceita por A, representada por $L(A)$, é o conjunto de todos os strings aceitos por A.
  - ➜ $L(A)$ é chamada de linguagem regular, e A representa um autômato que aceita uma expressão regular.

- Qual é a linguagem aceita por A?



  - ➜ Todos os strings que terminam com a sequência 0 1, i.e., $(0^*\,1^*)^*0\,1$, onde o sobrescrito $*$ significa a ocorrência de 0 ou mais vezes do símbolo ou sequência em questão.

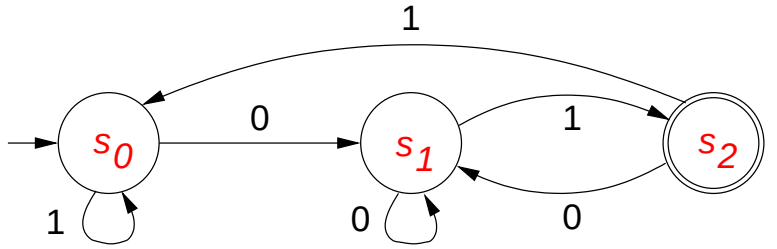
Função do estado final

- Suponha que um autômato esteja num estado–não necessariamente o inicial–e recebe uma sequência de símbolos. Em que estado o autômato terminará?
  - Depende da combinação de símbolos de entrada e estados, e é dado pela função do estado final.

- Definição: Seja A um autômato e I^* definidos como antes e seja a função do estado final $N^* : S \times I^* \to S$. Se $s \in S$ e $w \in I^*$ então $N^*(s, w)$ é o estado para o qual o autônomo pára quando todos os símbolos de w são fornecidos em sequência para A a partir do estado s.

Função do estado final

- Exemplo: dado o autômato abaixo, determine $N^*(s_1, 10110)$



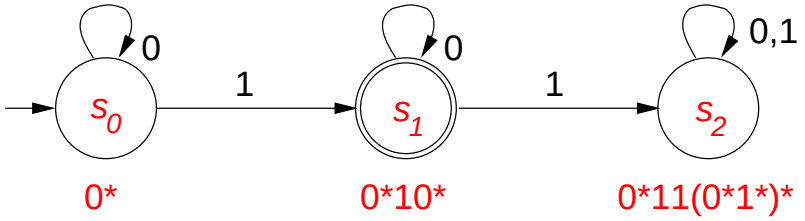
$$s_1 \xrightarrow{1} s_2 \xrightarrow{0} s_1 \xrightarrow{1} s_2 \xrightarrow{1} s_0 \xrightarrow{0} s_1$$

ou seja,

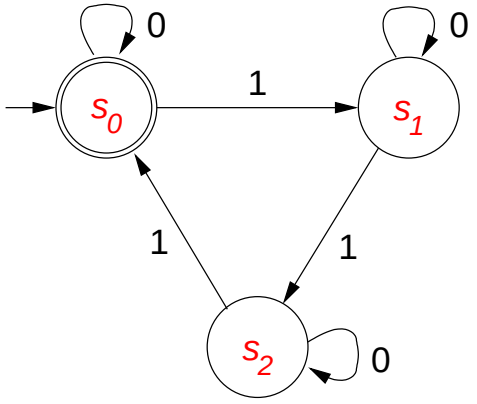
$$N^*(s_1, 10110) = s_1.$$

Projetando um AF que aceita uma linguagem

- Projete um AF para aceitar strings de 0's e 1's que contém exatamente um único 1.

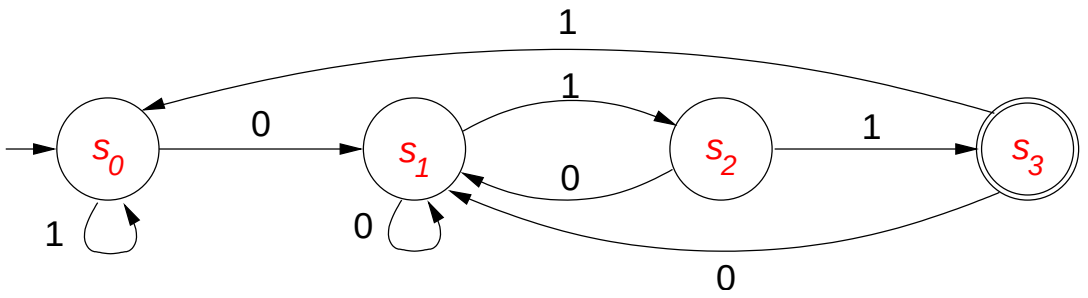


- Projete um AF para aceitar strings de 0's e 1's para os quais o número de 1's é divisível por 3.



➜ Linguagem aceita: $0^*(10^*10^*10^*)^*$

Implementando um AF através de um programa



Seja o autômato ao lado que reconhece todos os strings que terminam em 011.

Um possível algoritmo em alto nível que imita o funcionamento do autômato é:

EXECUTAAUTÔMATO(A)

```
 1  state ← 0
 2  symbol ← "primeiro símbolo do string de entrada"
 3  while symbol ≠ ε
 4  do choose state of
 5          0: if symbol = 0 then state ← 1 else state ← 0
 6          1: if symbol = 0 then state ← 1 else state ← 2
 7          2: if symbol = 0 then state ← 1 else state ← 3
 8          3: if symbol = 0 then state ← 1 else state ← 0
 9     symbol ← "próximo símbolo do string de entrada"
10  return state    ▷ O estado final é 3 sse o string termina em 011ε
```

Implementando um AF através de um programa

Um outro possível algoritmo em alto nível que imita o funcionamento do autômato e usa a função do próximo estado é:

EXECUTAAUTÔMATO(A)

```
 1  state ← 0
    ▷ Para cada linha abaixo será atribuído a cada coluna o valor correspondente
 2  N[0, ∗] ← [1, 0]
 3  N[1, ∗] ← [1, 2]
 4  N[2, ∗] ← [1, 3]
 5  N[3, ∗] ← [1, 0]
 6  symbol ← "primeiro símbolo do string de entrada"
 7  while symbol ≠ ε
 8  do state ← N[state, symbol]
 9       symbol ← "próximo símbolo do string de entrada"
10  return state    ▷ O estado final é 3 sse o string termina em 011ε
```

Função injetiva

- Definição: Seja $F : X \to Y$. F é uma função um-para-um ou injetiva sse para todos $x_1, x_2 \in X$,

$$\text{se } F(x_1) = F(x_2) \text{ então } x_1 = x_2,$$

ou

$$\text{se } x_1 \neq x_2 \text{ então } F(x_1) \neq F(x_2)$$

Função injetiva: Aplicação função hash

- Motivação: suponha que deseja-se procurar (pesquisar) o mais rápido possível alguns valores (chaves) que não estão ordenados. Uma possível solução é definir uma função que usa a própria chave para saber a sua localização no conjunto de valores.
- Esta função é conhecida como função hash.
- Objetivo ideal: função hash seja injetiva para que não haja "colisões de chaves."
- Para tornar este método aceitável temos que resolver dois problemas:
  (i) Encontrar uma função $h(k)$ que possa espalhar as chaves de forma uniforme pela tabela.
  (ii) Encontrar um mecanismo para resolver o problema de localizar um registro com chave k entre aquelas cujas chaves colidem na mesma entrada da tabela.

Função hash

- Por melhor que seja a função $h(k)$ e por mais esparsa que seja a ocupação da tabela colisões ocorrem!

- Paradoxo do aniversário:
  - Quando 23 ou mais pessoas estão juntas existe mais de 50% de chance que duas pessoas tenham a mesma data de aniversário.

  ➜ Logo, uma tabela com 365 entradas e com 23 chaves ou mais, cujos endereços são calculados como se fossem uniformes e randômicos, mais da metade das vezes, duas ou mais chaves vão ter o mesmo endereço na tabela.

Função hash

- Uma função de transformação deve mapear chaves em inteiros (índices) dentro do intervalo $[0..M-1]$ onde M é o tamanho da tabela.

- A função de transformação ideal é aquela que:
  1. Seja simples de ser computada, e
  2. Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

- Método mais usado:
  - Usa o resto da divisão inteira por M:

$$h(k) = k \bmod M$$

onde k é um inteiro correspondente à chave e que representa um índice na tabela.

Exemplo de função hash

- Seja a i-ésima letra do alfabeto representada pelo número i, onde $A = 1, \ldots$.
- Seja a função hash

$$h(k) = k \bmod M$$

  onde $M = 7$.

- A inserção das chaves L, U, N, E, S, nesta ordem, fornece os seguintes valores:

$$\begin{aligned} h(\mathsf{L}) &= 12 \bmod 7 = 5 \\ h(\mathsf{U}) &= 21 \bmod 7 = 0 \\ h(\mathsf{N}) &= 14 \bmod 7 = 0 \\ h(\mathsf{E}) &= 5 \bmod 7 = 5 \\ h(\mathsf{S}) &= 19 \bmod 7 = 5 \end{aligned}$$

- Uma possível inserção na tabela seria:

0	1	2	3	4	5	6
U	N	S			L	E

Exemplo de função hash

- Transformação de chaves não numéricas em números:

$$k = \sum_{i=1}^{n} \text{Chave}[i] \times p[i],$$

  onde
  - n é o número de caracteres da chave;
  - Chave$[i]$ corresponde à representação ASCII do i-ésimo caractere da chave;
  - $p[i]$ é um inteiro de um conjunto de pesos gerados randomicamente para $1 \leq i \leq n$.

➜ Vantagem em usar pesos:
  - Dois conjuntos diferentes de pesos $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, leva a duas funções de transformação $h_1(k)$ e $h_2(k)$ diferentes.

Função hash perfeita

- Se $h(k_i) = h(k_j)$ sse $i = j$, então não há colisões, e a função de transformação é chamada de função de transformação perfeita ou função hash perfeita (hp).

- Se o número de chaves N e o tamanho da tabela M são iguais então temos uma função de transformação perfeita mínima.

- Se $k_i \leq k_j$ e $h(k_i) \leq h(k_j)$, então a ordem lexicográfica é preservada. Neste caso, temos uma função de transformação perfeita mínima com ordem preservada.

Função hash perfeita: Comentários

- Não há necessidade de armazenar a chave, pois o registro é localizado sempre a partir do resultado da função de transformação.

- Uma função de transformação perfeita é específica para um conjunto de chaves conhecido.

Funções sobrejetiva, bijetiva e inversa

- Definição (função sobrejetiva):
  Seja $F : X \to Y$. F é uma função sobrejetiva sse para todo $y \in Y$ é possível achar um $x \in X$ tal que $F(x) = y$.

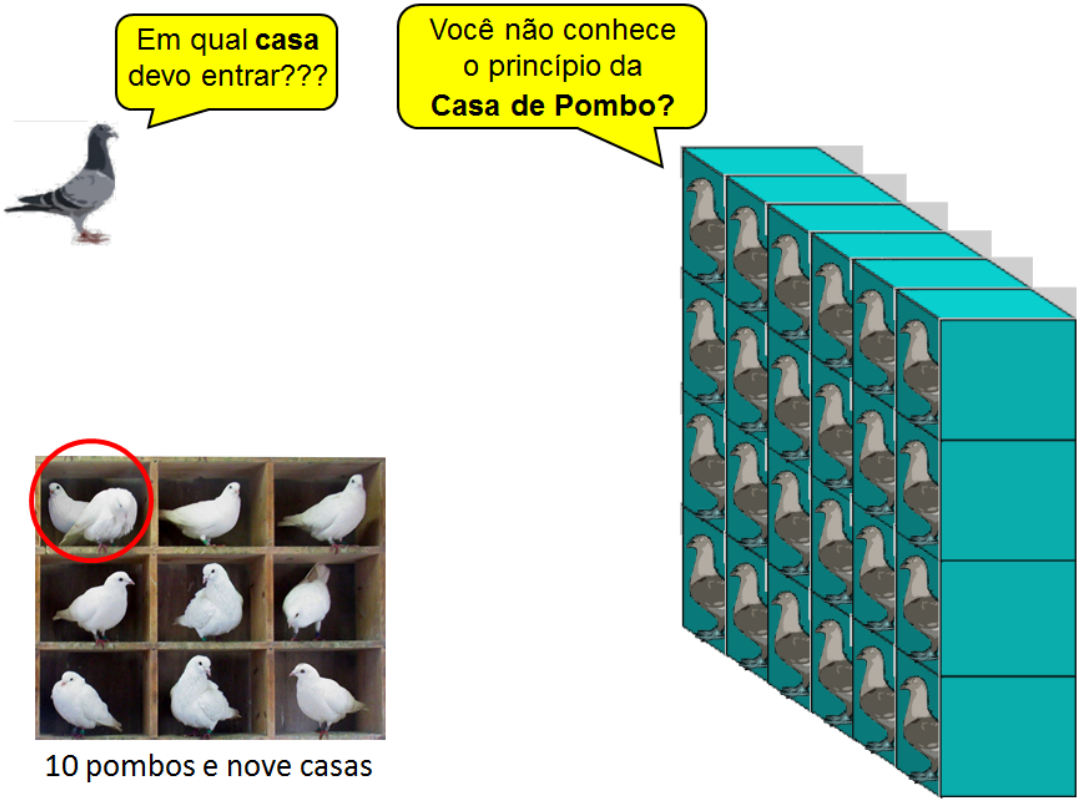
- Definição (função bijetiva):
  Seja $F : X \to Y$. F é uma função bijetiva sse F é injetiva e sobrejetiva.

- Definição (função inversa):
  Seja $F : X \to Y$ uma função bijetiva. Existe uma função $F^{-1} : Y \to X$ tal que

$$F^{-1}(y) = x \Leftrightarrow y = F(x)$$

O princípio da "Casa de Pombo" ou *The Pigeonhole Principle*

- Princípio (1ª versão):
  Se n pombos ("pigeons") entram em m casas num pombal ("pigeonholes") e $n > m$, então pelo menos uma das casas deve conter dois ou mais pombos.



10 pombos e nove casas

Peter Dirichlet (1805-1859), matemático alemão. Foi o primeiro a expressar esse princípio em 1834, que tem importantes aplicações em teoria dos números.

O princípio da "Casa de Pombo" ou *The Pigeonhole Principle*

- Princípio (2ª versão):
  Uma função definida de um conjunto finito para outro conjunto finito menor não pode ser injetiva. Existem pelo menos dois elementos no domínio que têm a mesma imagem no co-domínio.

➜ Aplicações deste princípio estão "por toda parte" em Ciência da Computação.

➜ Exemplo: função hash em pesquisa e criptografia.

O princípio da "Casa de Pombo": Exemplos

- Num grupo de seis pessoas, existem obrigatoriamente pelo menos duas que nasceram no mesmo mês?
  - Não.

- E se considerarmos 13 pessoas?
  - Sim.

- Existem pelo menos duas pessoas em BH que têm o mesmo número de fios de cabelo?
  - Sim, já que a população de BH é superior a 2 milhões de habitantes e sabe-se que uma pessoa tem no máximo 300 mil fios de cabelo.

- Você acaba de acordar, mas ainda não abriu os olhos, e abre uma gaveta de meias que tem cinco pares de meias verde-limão e cinco amarelo-ouro. Qual é o número mínimo de meias que você deve pegar para ter um par da mesma cor?
  - Três. Duas podem ser diferentes mas a terceira tem que ser obrigatoriamente de uma das duas cores e assim tem-se um par.

O princípio da "Casa de Pombo": Exemplos

- Seja um triângulo equilátero com lado igual a 1. Se cinco pontos são selecionados no interior do triângulo então existe um par de pontos que está a uma distância menor que $1/2$?
  - Sim. Basta dividir o triângulo em quatro triângulos equiláteros com lado igual a 1/2.

- Seja $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Se cinco inteiros são selecionados de A, existe pelo menos um par de números cuja soma é 9?
  - Sim. Existem quatro pares que têm soma 9: $\{1, 8\}$, $\{2, 7\}$, $\{3, 6\}$, $\{4, 5\}$. Logo, pode-se selecionar quatro números, cada um num conjunto. Como o quinto cairá em um dos quatro então tem-se a soma 9.

- E se selecionarmos apenas quatro números?
  - Não.

O princípio da "Casa de Pombo": Exemplos

Seja S um conjunto com cinco números inteiros positivos ($\mathbb{Z}^+$) e distintos cujo maior número é 8. Prove que existem pelo menos duas somas iguais dos números dos subconjuntos não-vazios de S.

Prova:

- Seja s_A a soma dos números de um subconjunto não-vazio de S.
- Qualquer subconjunto de S pode ter no máximo uma soma

$$s_A \leq 4 + 5 + 6 + 7 + 8 = 30$$

(Caso A tenha os maiores números e considerando o subconjunto com todos eles)

e no mínimo uma soma de

$$s_A \geq 1.$$

(Caso A tenha o número 1 e considerando o subconjunto com ele apenas)

i.e.,

$$1 \leq s_A \leq 30.$$

O princípio da "Casa de Pombo": Exemplos

– S possui $2^5 - 1 = 31$ subconjuntos não-vazios.
(Lembre-se que o conjunto potência de um conjunto de n elementos possui 2^n elementos incluindo um elemento que é o conjunto vazio.)

– Existem 30 "pigeonholes", ou seja, qualquer soma desses subconjuntos está entre 1 e 30.

– Além disso, existem 31 conjuntos ("pigeons"), que devem ter somas entre 1 e 30.

➜ Assim, pelo princípio da "Casa de Pombo" existem pelo menos dois subconjuntos que têm a mesma soma.

O princípio da "Casa de Pombo": Exemplos

Seja L a linguagem que consiste de todos os strings da forma $a^k b^k$, onde k é um inteiro positivo. Simbolicamente, L é a linguagem sobre o alfabeto $\Sigma = \{a, b\}$ definida por

$$L = \{s \in \Sigma^* | s = a^k b^k, k \geq 1\}.$$

Prove que não existe um autômato finito que aceita L.

O princípio da "Casa de Pombo": Exemplos

Prova (por contradição):

- Suponha que exista um autômato finito A que aceita L.
- A tem um conjunto finito de estados s_1, s_2, ..., s_n, onde n é um inteiro positivo.
- Considere agora os strings que começam com a, a^2, a^3, a^4, ... No entanto, existem infinitos strings que começam com a e um número finito de estados.
- Logo, pelo princípio da "Casa de Pombo" existe um estado s_m (estado que conta a's), com $m \leq n$, e dois strings de entrada a^p e a^q, onde ou p ou q é maior que n e os dois strings são relacionados com esse estado.
- Isto significa que depois de entrar com p a's, o estado do autômato é s_m e entrando com p b's o autômato vai para o estado final s_m.
- Mas isso também implica que $a^q b^p$ vai para o estado final já que a^q também pára em s_m antes de entrar com os b's.
- Por suposição, A aceita L. Já que s é aceito por A, $s \in L$. Mas pela definição de L, L consiste apenas dos strings que têm o mesmo número de a's e b's e já que $p \neq q$, $s \notin L$. Logo, tem-se que $s \in L$ e $s \notin L$, que é uma contradição.
- A suposição é falsa e não existe um autômato que aceita L.

Generalização do princípio

- Princípio (3ª versão):
  Para qualquer função f, definida de um conjunto finito X para um conjunto finito Y e para qualquer inteiro positivo k, se $|X| > k \times |Y|$, então existe algum $y \in Y$ tal que y é a imagem de pelo menos $k + 1$ elementos distintos de X.

- Princípio (4ª versão—contrapositivo):
  Para qualquer função f, definida de um conjunto finito X para um conjunto finito Y e para qualquer inteiro positivo k, se para cada $y \in Y$, $f^{-1}(y)$ tem no máximo k elementos então X tem no máximo $k \times |Y|$ elementos.

Generalização do princípio

- Use a generalização do princípio da "Casa de Pombo" para mostrar que num grupo de 85 pessoas, a primeira letra dos nomes de pelo menos quatro pessoas é a mesma.
  - $85 > 3 \cdot 26 = 78$. Logo, a letra inicial dos nomes de pelo menos quatro pessoas é a mesma.

- O mesmo resultado pode ser dado usando a forma contrapositiva e a negação.
  - Suponha que não existam quatro pessoas que têm a mesma letra inicial no primeiro nome. Assim, no máximo três pessoas têm a mesma letra. Pela forma contrapositiva da generalização do princípio da "Casa de Pombo" isto implica num total de $3 \cdot 26 = 78$ pessoas. Mas isto é uma contradição pois existem 85 pessoas. Logo, pelo menos quatro pessoas têm a mesma letra inicial no primeiro nome.

Princípio da "Casa de Pombo"

- Definição (conjunto finito): Um conjunto X é finito sse é o conjunto vazio ou existe uma correspondência um-para-um do conjunto $\{1, 2, \ldots, n\}$ para X, onde n é um inteiro positivo. No primeiro caso, o número de elementos é zero e no segundo caso é n. Um conjunto que não é finito é chamado de infinito.

- **Teorema** (O princípio da "Casa de Pombo"):
  Para qualquer função f de um conjunto finito X para um conjunto finito Y, se a cardinalidade de X é maior que a de Y então f não é injetiva.

- **Teorema**:
  Sejam X e Y conjuntos finitos com o mesmo número de elementos e suponha que f é uma função de X para Y. Então f é injetiva sse f é sobrejetiva. (f é uma função sobrejetiva sse para todo $y \in Y$ é possível achar um $x \in X$ tal que $f(x) = y$.)

Composição de funções

- Definição (composição de funções): Sejam $f : X \to Y'$ e $g : Y \to Z$ funções com a propriedade que a faixa de f é um subconjunto do domínio de g, i.e., $Y' \subseteq Y$. Defina uma nova função $g \circ f : X \to Z$ da seguinte forma:

$$(g \circ f)(x) = g(f(x)), \forall x \in X.$$

  A função $g \circ f$ é chamada de composição de f e g.

- Exemplo:
  - $f : \mathbb{Z} \to \mathbb{Z}$, $f(n) = n + 1$.
  - $g : \mathbb{Z} \to \mathbb{Z}$, $g(n) = n^2$.

  - $f \circ g \stackrel{?}{=} g \circ f$.

    $(g \circ f)(n) = g(f(n)) = g(n+1) = (n+1)^2, \forall n \in \mathbb{Z}$

    $(f \circ g)(n) = f(g(n)) = f(n^2) = n^2 + 1, \forall n \in \mathbb{Z}$

    $\therefore f \circ g \neq g \circ f.$

Composição com a função identidade

- $f \circ i_x = f(i_x) = f(x)$,
  onde i_x é a função identidade.
- $i_x \circ f = i_x(f(x)) = f(x)$,
  onde i_x é a função identidade.
- **Teorema**:
  Se f é uma função de X para Y, e i_x é a função identidade em X e i_y é a função identidade em Y, então

$$f \circ i_x = f$$

$$i_y \circ f = f$$

- **Teorema**:
  Se $f : X \to Y$ é uma função bijetiva com função inversa $f^{-1} : Y \to X$, então

$$f^{-1} \circ f = i_x$$

$$f \circ f^{-1} = i_y$$

Composição de função injetiva e função sobrejetiva

- **Teorema**:
  Se $f : X \to Y$ e $g : Y \to Z$ são funções injetiva, então $g \circ f$ é injetiva.

- **Teorema**:
  Se $f : X \to Y$ e $g : Y \to Z$ são funções sobrejetiva, então $g \circ f$ é sobrejetiva.

Cardinalidade de conjuntos

- Definição (mesma cardinalidade): Sejam A e B quaisquer conjuntos. A tem a mesma cardinalidade de B sse existe uma correspondência um-para-um de A para B. Em outras palavras, A tem a mesma cardinalidade que B sse existe uma função f de A para B que é injetiva e sobrejetiva (i.e., bijetiva).

- **Teorema**:
  Para todos os conjuntos A, B e C,
  (a) A tem a mesma cardinalidade que A.
  ➜ propriedade reflexiva da cardinalidade.
  (b) Se A tem a mesma cardinalidade que B, então B tem a mesma cardinalidade que A.
  ➜ propriedade simétrica da cardinalidade.
  (c) Se A tem a mesma cardinalidade que B, e B tem a mesma cardinalidade que C então A tem a mesma cardinalidade que C.
  ➜ propriedade transitiva da cardinalidade.

- Definição (mesma cardinalidade): A e B têm a mesma cardinalidade sse, A tem a mesma cardinalidade que B ou B tem a mesma cardinalidade que A.

Conjuntos contáveis

- Definição (conjunto contável infinito): Um conjunto é chamado contável infinito sse ele possui a mesma cardinalidade do conjunto dos inteiros positivos $\mathbb{Z}^+$. Um conjunto é chamado contável sse é finito ou contável infinito. Um conjunto que não é contável é chamado de incontável.

- Exemplo: Mostre que o conjunto $\mathbb{Z}$, conjunto de todos os inteiros é contável.

$$\begin{array}{rcl} 0 & \rightarrow & 1 \\ 1 & \rightarrow & 2 \\ -1 & \rightarrow & 3 \\ 2 & \rightarrow & 4 \\ -2 & \rightarrow & 5 \\ \vdots & \vdots & \vdots \end{array}$$

- Exemplo: O conjunto de todos os números racionais positivos são contáveis.

Conjuntos contáveis

- **Teorema**:
  O conjunto de todos os números reais entre 0 e 1 é incontável.

- **Teorema**:
  Qualquer subconjunto de um conjunto contável é contável.

- Exemplo: O conjunto de todos os números reais tem a mesma cardinalidade que o conjunto dos números reais entre 0 e 1.

- Exemplo: O conjunto de todos os programas de computador numa dada linguagem de programação é contável.

Como medir o custo de execução de um algoritmo?

- **Função de Custo** ou **Função de Complexidade**
  - $f(n) =$ medida de custo necessário para executar um algoritmo para um problema de tamanho n.
  - Se $f(n)$ é uma medida da quantidade de tempo necessário para executar um algoritmo em um problema de tamanho n, então f é chamada *função de complexidade de tempo de algoritmo*.
  - Se $f(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo em um problema de tamanho n, então f é chamada *função de complexidade de espaço de algoritmo*.

- Observação: tempo não é tempo!
  - É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

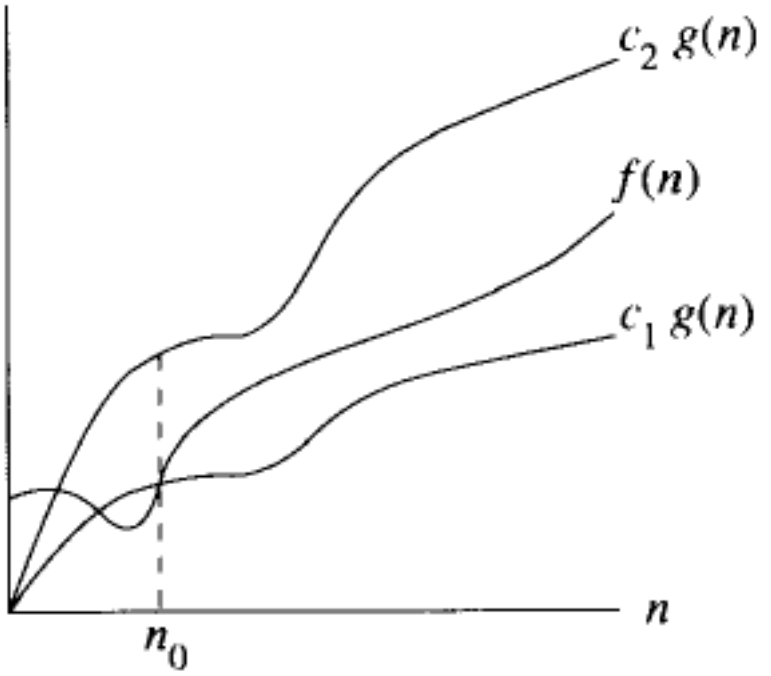
Custo assintótico de funções

- É interessante comparar algoritmos para valores grandes de n.
- O *custo assintótico* de uma função $f(n)$ representa o limite do comportamento de custo quando n cresce.
- Em geral, o custo aumenta com o tamanho n do problema.
- Observação:
  - Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado.

Notação assintótica de funções

- Existem três notações principais na análise assintótica de funções:
  - Notação Θ.
  - Notação O (“O” grande).
  - Notação Ω (ômega grande).

Notação Θ



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

Notação Θ

- A notação Θ limita a função por fatores constantes.

- Escreve-se $f(n) = \Theta(g(n))$, se existirem constantes positivas c_1, c_2 e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ está sempre entre $c_1 g(n)$ e $c_2 g(n)$ inclusive.
  - ➜ Pode-se dizer que $g(n)$ é um limite assintótico firme (em inglês, *asymptotically tight bound*) para $f(n)$.

$$f(n) = \Theta(g(n)),$$
$$\exists\, c_1 > 0, c_2 > 0, n_0, \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$$

Observe que a notação Θ define um conjunto de funções:

$$\Theta(g(n)) =$$
$$\{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists\, c_1 > 0, c_2 > 0, n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}.$$

Notação Θ: Exemplo 1

- Mostre que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Para provar esta afirmação, devemos achar constantes $c_1 > 0, c_2 > 0, n > 0$, tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq n_0$.

Se dividirmos a expressão acima por n^2 temos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

Notação Θ: Exemplo 1

A inequação mais a direita será sempre válida para qualquer valor de $n \geq 1$ ao escolhermos $c_2 \geq \frac{1}{2}$.

Da mesma forma, a inequação mais a esquerda será sempre válida para qualquer valor de $n \geq 7$ ao escolhermos $c_1 \leq \frac{1}{14}$.

Assim, ao escolhermos $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, podemos verificar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Note que existem outras escolhas para as constantes c_1 e c_2, mas o fato importante é que *a escolha existe*.

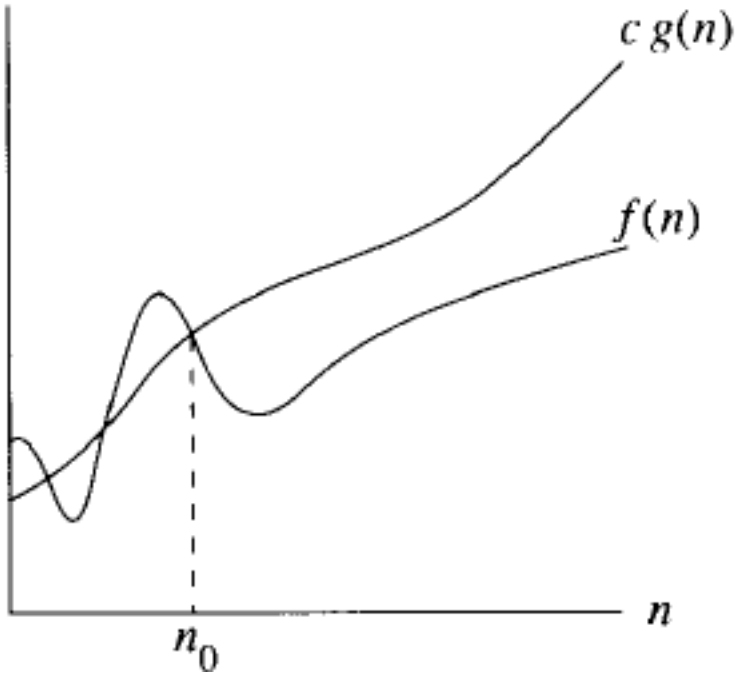
Note também que a escolha destas constantes depende da função $\frac{1}{2}n^2 - 3n$.

Uma função diferente pertencente a $\Theta(n^2)$ irá provavelmente requerer outras constantes.

Notação Θ: Exemplo 2

- Usando a definição formal de Θ prove que $6n^3 \neq \Theta(n^2)$.

Notação O



$$f(n) = O(g(n))$$

Notação O

- A notação O define um limite superior para a função, por um fator constante.
- Escreve-se $f(n) = O(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é menor ou igual a $cg(n)$.
  - ➜ Pode-se dizer que $g(n)$ é um limite assintótico superior (em inglês, *asymptotically upper bound*) para $f(n)$.

$$f(n) = O(g(n)), \exists\, c > 0, n_0, \mid 0 \leq f(n) \leq cg(n),\ \forall n \geq n_0.$$

Observe que a notação O define um conjunto de funções:

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c > 0, n_0,\ 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}.$$

Notação O

- Quando a notação O é usada para expressar o tempo de execução de um algoritmo no pior caso, está se definindo também o limite (superior) do tempo de execução desse algoritmo para *todas* as entradas.

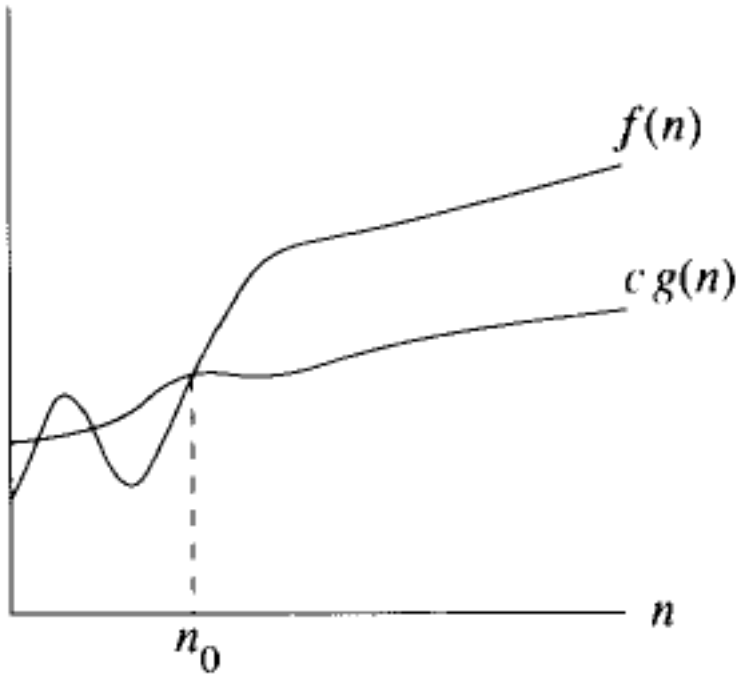
- Por exemplo, o algoritmo de ordenação por inserção (a ser estudado) é $O(n^2)$ no pior caso.
  - ➜ Este limite se *aplica* para *qualquer* entrada.

Notação O

- Tecnicamente é um abuso dizer que o tempo de execução do algoritmo de ordenação por inserção *é* $O(n^2)$ (i.e., sem especificar se é para o pior caso, melhor caso, ou caso médio):
  - ➜ O tempo de execução desse algoritmo depende de como os dados de entrada estão arranjados.
  - ➜ Se os dados de entrada já estiverem ordenados, este algoritmo tem um tempo de execução de $O(n)$, ou seja, o tempo de execução do algoritmo de ordenação por inserção no *melhor caso* é $O(n)$.

- O que se quer dizer quando se fala que "o tempo de execução" é $O(n^2)$" é que no pior caso o tempo de execução é $O(n^2)$.
  - ➜ Ou seja, não importa como os dados de entrada estão arranjados, o tempo de execução em qualquer entrada é $O(n^2)$.

Notação Ω



$$f(n) = \Omega(g(n))$$

Notação Ω

- A notação Ω define um limite inferior para a função, por um fator constante.

- Escreve-se $f(n) = \Omega(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é maior ou igual a $cg(n)$.
  - ➜ Pode-se dizer que $g(n)$ é um limite assintótico inferior (em inglês, *asymptotically lower bound*) para $f(n)$.

$$f(n) = \Omega(g(n)), \exists\, c > 0, n_0,\ |0 \leq cg(n) \leq f(n), \forall n \geq n_0.$$

Observe que a notação Ω define um conjunto de funções:

$$\Omega(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists\, c > 0, n_0,\ |0 \leq cg(n) \leq f(n), \forall n \geq n_0\}.$$

Notação Ω

- Quando a notação Ω é usada para expressar o tempo de execução de um algoritmo no melhor caso, está se definindo também o limite (inferior) do tempo de execução desse algoritmo para *todas* as entradas.

- Por exemplo, o algoritmo de ordenação por inserção é $\Omega(n)$ no melhor caso.
  ➜ O tempo de execução do algoritmo de ordenação por inserção é $\Omega(n)$.

- O que significa dizer que "o tempo de execução" (i.e., sem especificar se é para o pior caso, melhor caso, ou caso médio) é $\Omega(g(n))$?
  – O tempo de execução desse algoritmo é pelo menos uma constante vezes $g(n)$ para valores suficientemente grandes de n.

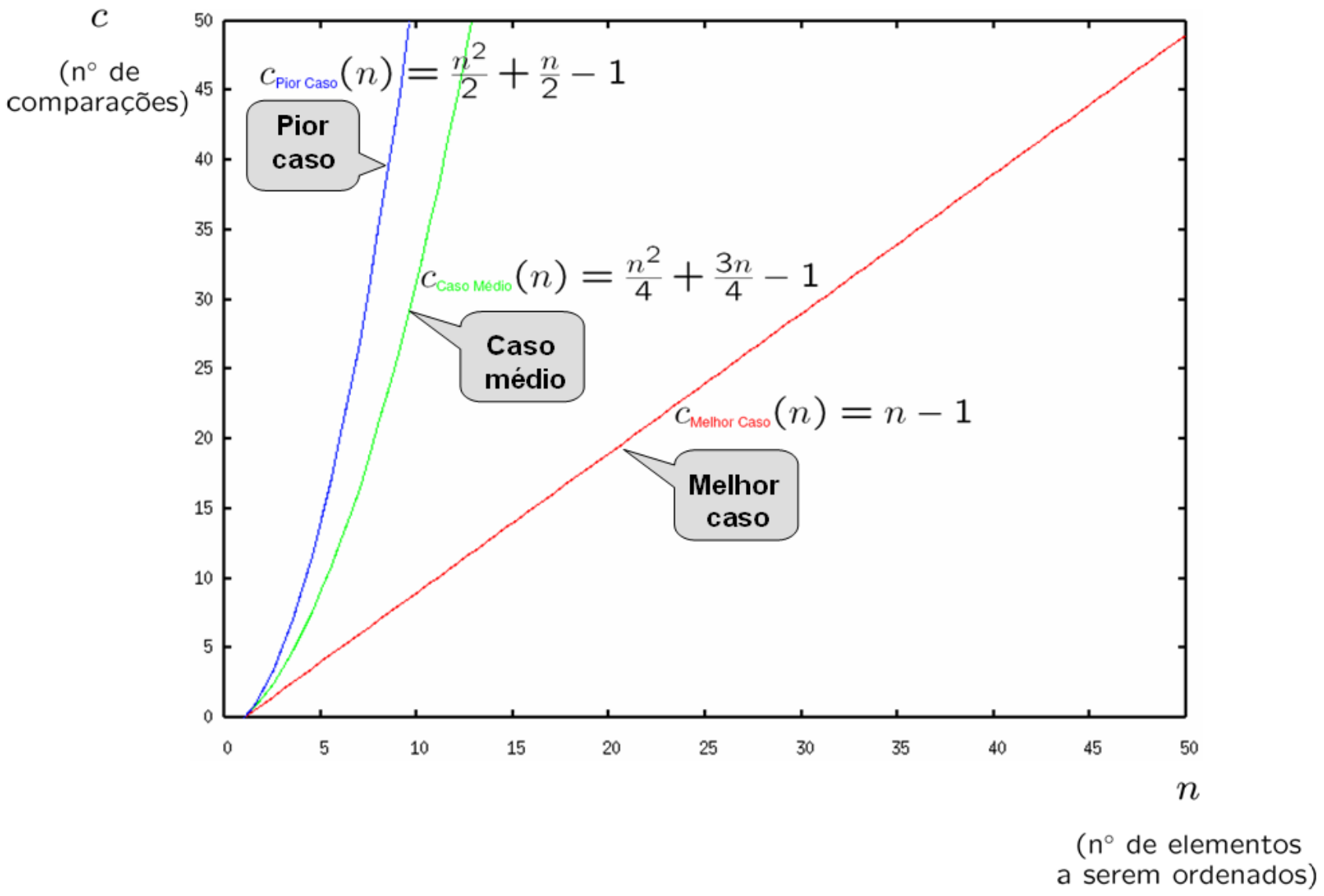
Limites do algoritmo de ordenação por inserção

- O tempo de execução do algoritmo de ordenação por inserção está entre $\Omega(n)$ e $O(n^2)$.

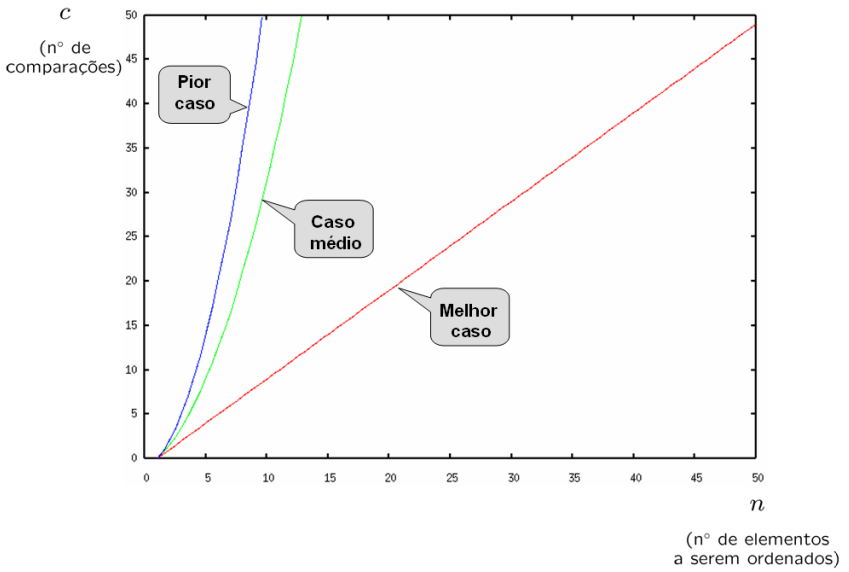
- Estes limites são assintoticamente os mais firmes possíveis.
  - Por exemplo, o tempo de execução deste algoritmo *não é* $\Omega(n^2)$, pois o algoritmo executa em tempo $\Theta(n)$ quando a entrada já está ordenada.

- Não é contraditório dizer que o tempo de execução deste algoritmo no *pior caso* é $\Omega(n^2)$, já que existem entradas para este algoritmo que fazem com que ele execute em tempo $\Omega(n^2)$.

Funções de custo (nº de comparações) do algoritmo de ordenação por Inserção



Funções de custo e notações assintóticas do algoritmo de ordenação por Inserção



Pior Caso:

$$c_{\text{Pior Caso}}(n) = \frac{n^2}{2} + \frac{n}{2} - 1 = \begin{matrix} O \\ \Theta \\ \Omega \end{matrix} (\ n^2\ )$$

Caso Médio:

$$c_{\text{Caso Médio}}(n) = \frac{n^2}{4} + \frac{3n}{4} - 1 = \begin{matrix} O \\ \Theta \\ \Omega \end{matrix} (\ n^2\ )$$

Melhor caso:

$$c_{\text{Melhor Caso}}(n) = n - 1 = \begin{matrix} O \\ \Theta \\ \Omega \end{matrix} (\ n\ )$$

indica a notação normalmente usada para esse caso.

Teorema

Para quaisquer funções $f(n)$ e $g(n)$,

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

se e somente se,

$$f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$$

Mais sobre notação assintótica de funções

- Existem duas outras notações na análise assintótica de funções:
  - Notação o (“o” pequeno).
  - Notação ω (“ômega” pequeno).

- Estas duas notações não são usadas normalmente, mas é importante saber seus conceitos e diferenças em relação às notações O e Ω, respectivamente.

Notação o

- O limite assintótico superior definido pela notação O pode ser assintoticamente firme ou não.
  - Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente firme, mas o limite $2n = O(n^2)$ não é.

- A notação o é usada para definir um limite superior que não é assintoticamente firme.

- Formalmente a notação o é definida como:

$$f(n) = o(g(n)),\ \forall c > 0 \text{ e } n_0 \mid 0 \leq f(n) < cg(n),\ \forall\, n \geq n_0$$

- Exemplo, $2n = o(n^2)$ mas $2n^2 \neq o(n^2)$.

Notação o

- As definições das notações O (o grande) e o (o pequeno) são similares.
  - A diferença principal é que em $f(n) = O(g(n))$, a expressão

    $$0 \leq f(n) \leq cg(n)$$

    é válida para pelo menos uma constante $c > 0$.

- Intuitivamente, a função $f(n)$ tem um crescimento muito menor que $g(n)$ quando n tende para infinito. Isto pode ser expresso da seguinte forma:

  $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

  ➜ Alguns autores usam este limite como a definição de o.

Notação ω

- Por analogia, a notação ω está relacionada com a notação Ω da mesma forma que a notação o está relacionada com a notação O.
- Formalmente a notação ω é definida como:

$$f(n) = \omega(g(n)),\ \forall c > 0 \text{ e } n_0 \mid 0 \leq cg(n) < f(n),\ \forall\, n \geq n_0$$

- Por exemplo, $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$.
- A relação $f(n) = \omega(g(n))$ implica em

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

se o limite existir.

Classes de comportamento assintótico
Complexidade constante

- $f(n) = O(1)$
  - O uso do algoritmo independe do tamanho de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.

- O que significa um algoritmo ser $O(2)$ ou $O(5)$?

Classes de comportamento assintótico
Complexidade logarítmica

- $f(n) = O(\log n)$
  - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.
  - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.

- Exemplo, supondo que a base do logaritmo seja 2:
  - Para $n = 1\,000$, $\log_2 \approx 10$.
  - Para $n = 1\,000\,000$, $\log_2 \approx 20$.
  - Algoritmo de pesquisa binária.

Classes de comportamento assintótico
Complexidade linear

- $f(n) = O(n)$
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - Esta é a melhor situação possível para um algoritmo que tem que processar n ou produzir n elementos de saída.
  - Cada vez que n dobra de tamanho, o tempo de execução também dobra.

- Exemplo:
  - Algoritmo de pesquisa sequencial.
  - Algoritmo para teste de planaridade de um grafo.

Classes de comportamento assintótico
Complexidade linear logarítmica

- $f(n) = O(n \log n)$
  - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções.
  - Caso típico dos algoritmos baseados no paradigma *divisão-e-conquista*.

- Exemplo, supondo que a base do logaritmo seja 2:
  - Para $n = 1\,000\,000$, $\log_2 \approx 20\,000\,000$.
  - Para $n = 2\,000\,000$, $\log_2 \approx 42\,000\,000$.
  - Algoritmo de ordenação MergeSort.

Classes de comportamento assintótico
Complexidade quadrática

- $f(n) = O(n^2)$
  - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro do outro.

- Exemplo:
  - Para $n = 1\,000$, o número de operações é da ordem de $1\,000\,000$.
  - Sempre que n dobra o tempo de execução é multiplicado por 4.
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos *relativamente* pequenos.
  - Algoritmos de ordenação simples como seleção e inserção.

Classes de comportamento assintótico
Complexidade cúbica

- $f(n) = O(n^3)$
  - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas *relativamente* pequenos.

- Exemplo:
  - Para $n = 100$, o número de operações é da ordem de $1\,000\,000$.
  - Sempre que n dobra o tempo de execução é multiplicado por 8.
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos *relativamente* pequenos.
  - Algoritmo para multiplicação de matriz.

Classes de comportamento assintótico
Complexidade exponencial

- $f(n) = O(2^n)$
  - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático.
  - Eles ocorrem na solução de problemas quando se usa a *força bruta* para resolvê-los.

- Exemplo:
  - Para $n = 20$, o tempo de execução é cerca de $1\,000\,000$
  - Sempre que n dobra o tempo de execução fica elevado ao quadrado.
  - Algoritmo do Caixeiro Viajante.

Hierarquias de funções

A seguinte hierarquia de funções pode ser definida do ponto de vista assintótico:

$$1 \prec \log\log n \prec \log n \prec n^{\epsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}$$

onde ϵ e c são constantes arbitrárias com $0 < \epsilon < 1 < c$.

Hierarquias de funções

- Usando MatLab, ou um outro pacote matemático/software gráfico, desenhe os gráficos dessas funções, quando $n \to \infty$.

Hierarquias de funções

Onde as seguintes funções se encaixam nessa hierarquia?

(a) $\pi(n) = \frac{n}{\ln n}$. Esta função define o número de primos menor ou igual a n.

(b) $e^{\sqrt{\log n}}$.
Dica: $e^{f(n)} \prec e^{g(n)} \iff \lim_{n\to\infty}(f(n) - g(n)) = -\infty$.

Hierarquias de funções
Preliminares

A hierarquia apresentada está relacionada com funções que vão para o infinito. No entanto, podemos ter o oposto dessas funções já que elas nunca são zero. Isto é,

$$f(n) \prec g(n) \iff \frac{1}{g(n)} \prec \frac{1}{f(n)}.$$

Assim, todas as funções (exceto 1) tendem para zero:

$$\frac{1}{c^{c^n}} \prec \frac{1}{n^n} \prec \frac{1}{c^n} \prec \frac{1}{n^{\log n}} \prec \frac{1}{n^c} \prec \frac{1}{n^\epsilon} \prec \frac{1}{\log n} \prec \frac{1}{\log \log n} \prec 1$$

Hierarquias de funções
Solução de (a)

- $\pi(n) = \frac{n}{\ln n}$

Temos que (note que a base do logaritmo não altera a hierarquia):

$$\frac{1}{n^\epsilon} \prec \frac{1}{\ln n} \prec 1$$

Multiplicando por n, temos:

$$\frac{n}{n^\epsilon} \prec \frac{n}{\ln n} \prec n,$$

ou seja,

$$n^{1-\epsilon} \prec \pi(n) \prec n$$

Note que o valor $1 - \epsilon$ ainda é menor que 1.

Hierarquias de funções
Solução de (b)

- $e^{\sqrt{\log n}}$

Dado a hierarquia:

$$1 \prec \ln \ln n \prec \sqrt{\ln n} \prec \epsilon \ln n$$

e elevando a e, temos que:

$$e^1 \prec e^{\ln \ln n} \prec e^{\sqrt{\ln n}} \prec e^{\epsilon \ln n}$$

Simplificando temos:

$$e \prec \ln n \prec e^{\sqrt{\ln n}} \prec n^{\epsilon}$$

Algoritmo exponencial $\times$ Algoritmo polinomial

- Funções de complexidade:
  - Um algoritmo cuja função de complexidade é $O(c^n), c > 1$, é chamado de *algoritmo exponencial* no tempo de execução.
  - Um algoritmo cuja função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio de grau n, é chamado de *algoritmo polinomial* no tempo de execução.

- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.

- Essa é a razão porque algoritmos polinomiais são muito mais úteis na prática do que algoritmos exponenciais.
  - Geralmente, algoritmos exponenciais são simples variações de pesquisa exaustiva.

Algoritmo exponencial $\times$ Algoritmo polinomial

- Os algoritmos polinomiais são geralmente obtidos através de um entendimento mais profundo da estrutura do problema.

- Tratabilidade dos problemas:
  - Um problema é considerado **intratável** se ele é tão difícil que não se conhece um algoritmo polinomial para resolvê-lo.
  - Um problema é considerado **tratável** (bem resolvido) se existe um algoritmo polinomial para resolvê-lo.

➜ Aspecto importante no projeto de algoritmos.

Objetos recursivos

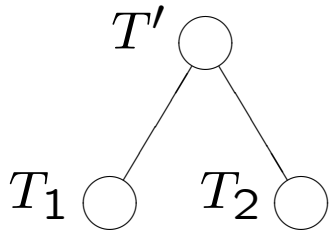
- Um objeto é recursivo quando é definido parcialmente em termos de si mesmo.

- Exemplo 1: Números naturais:
  (a) 1 é um número natural.
  (b) o sucessor de um número natural é um número natural.

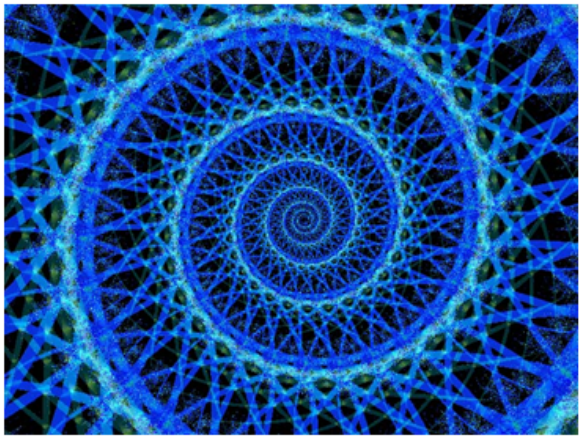
- Exemplo 2: Função fatorial:
  (a) $0! = 1$.
  (b) se $n > 0$ então $n! = n \cdot (n-1)!$

Objetos recursivos

- Exemplo 3: Árvores.
  (a) A árvore vazia é uma árvore.
  (b) se T_1 e T_2 são árvores então T' é um árvore.

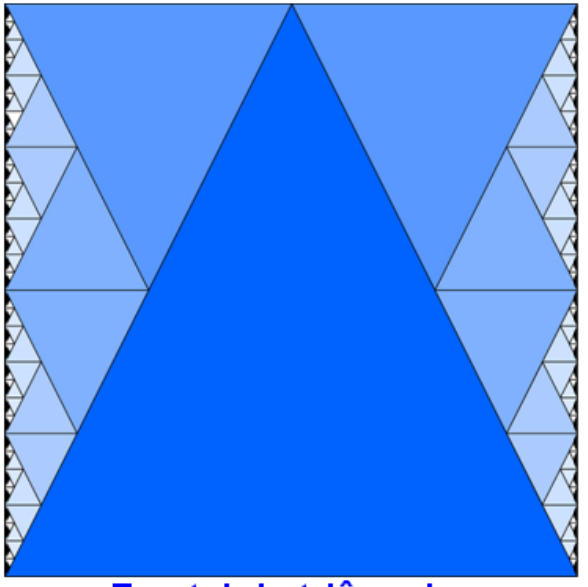


Objetos recursivos: Exemplos

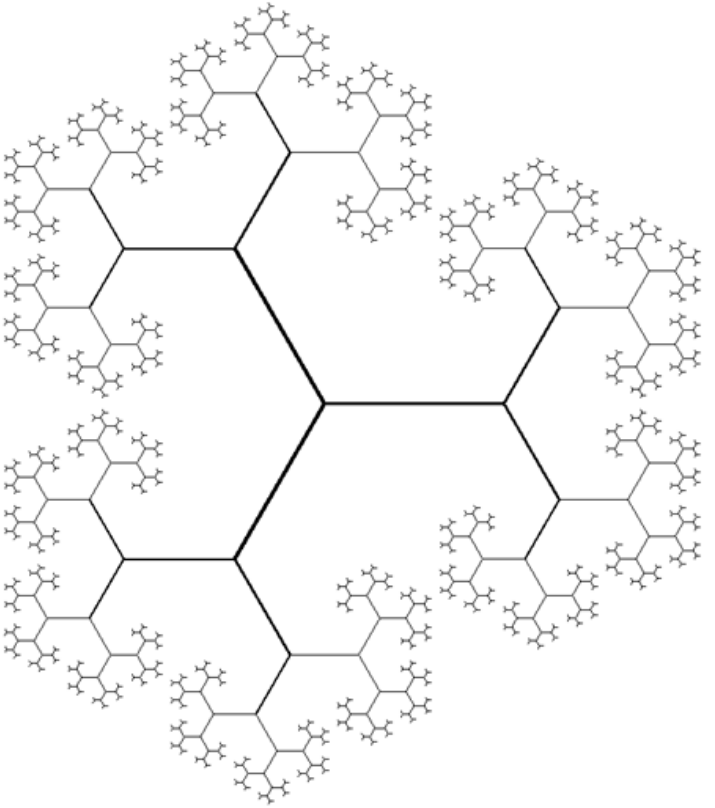


Fractal espiral

Fractal de Fern



Fractal de triângulos



Derivação binária

Objetos recursivos: Exemplos

Foto recursiva

Imagem recursiva

Pensamento recursivo

Poder da recursão

- Definir um conjunto infinito de objetos através de um comando finito.
- Um problema recursivo P pode ser expresso como $P \equiv \mathcal{P}[S_i, P]$, onde $\mathcal{P}$ é a composição de comandos S_i e do próprio P.
- Importante: constantes e variáveis locais a P são duplicadas a cada chamada recursiva.

Problema de terminação

- Definir um condição de terminação.
- Idéia:
  - Associar um parâmetro, por exemplo n, com P e chamar P recursivamente com $n - 1$ como parâmetro.
  - A condição $n > 0$ garante a terminação.
  - Exemplo:

$$P(n) \equiv \textbf{if } n > 0 \textbf{ then } \mathcal{P}[S_i; P(n-1)].$$

- Importante: na prática é necessário:
  - mostrar que o nível de recursão é finito, e
  - tem que ser mantido pequeno! Por que?

Razões para limitar a recursão

- Memória necessária para acomodar variáveis a cada chamada.
- O estado corrente da computação tem que ser armazenado para permitir a volta da chamada recursiva.

Exemplo:

```
FATORIAL(n)
     ▷ F: variável auxiliar
1    if n > 0
2    then F ← n × FATORIAL(n − 1)
3    else F ← 1
4    return F
```

FATORIAL(4) →		
	1	4×FATORIAL(3)
	2	3×FATORIAL(2)
	3	2×FATORIAL(1)
	4	1×FATORIAL(0)
	1	

Quando não usar recursividade

- Algoritmos recursivos são apropriados quando o problema é definido em termos recursivos.

- Entretanto, uma definição recursiva não implica necessariamente que a implementação recursiva é a melhor solução!

- Casos onde evitar recursividade:
  - P $\equiv$ if condição then $(S_i; P)$
    Exemplo: P $\equiv$ **if** $i < n$ **then** $(\overbrace{i := i + 1; F := i \times F}; P)$

Eliminando a recursividade de Cauda *(Tail recursion)*

```
FATORIAL(n)
      ▷ F, i: variáveis auxiliares
 1  i ← 0
 2  F ← 1
 3  while i < n
 4  do i ← i + 1
 5        F ← F × i
 6  return F
```

Logo,

$$P \equiv \textbf{ if } B \textbf{ then } (S; P)$$

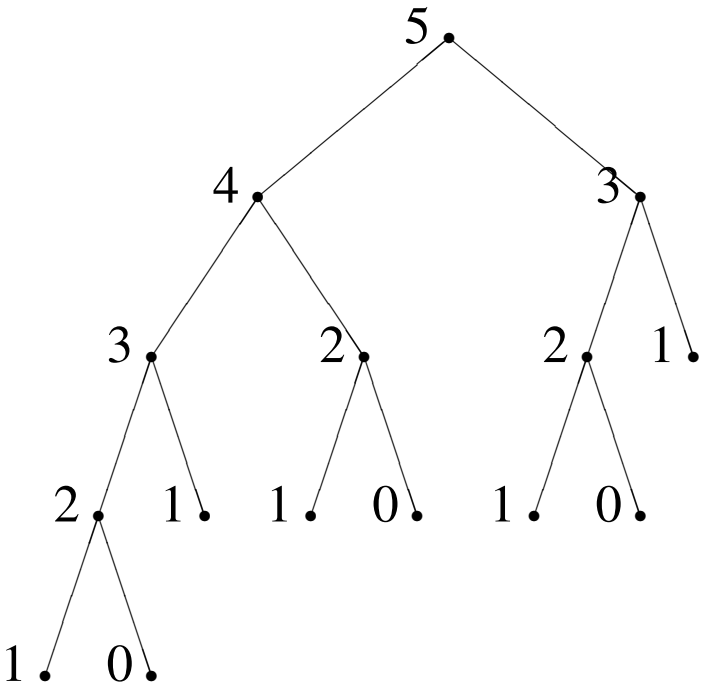
deve ser transformado em

$$P \equiv (x = x_0;\ \textbf{while } B \textbf{ do } S).$$

Outro exemplo

```
FIB(A)
      ▷ F: variável auxiliar
 1    if n = 0
 2    then F ← 0
 3    else if n = 1
 4          then F ← 1
 5          else F ← FIB(n − 1) + (n − 2)
 6    return F
```

Observação: para cada chamada a FIB(n), FIB é ativada 2 vezes



Solução Óbvia

```
FIB(n)
    ▷ F, Fant, i, Temp: variáveis auxiliares
 1  i ← 1
 2  F ← 1
 3  Fant ← 0
 4  while i < n
 5  do Temp ← F
 6       F ← F + Fant
 7       Fant ← Temp
 8       i ← i + 1
 9  return F
```

- Complexidade de tempo: $T(n) = n - 1$.
- Complexidade de espaço: $E(n) = O(1)$.

Comentários sobre recursividade

- Evitar o uso de recursividade quando existe uma solução óbvia por iteração!

- Exemplos:
  - Fatorial.
  - Série de Fibonacci.

Análise de algoritmos recursivos

- Comportamento é descrito por uma equação (função) de recorrência.

- Enfoque possível:
  - Usar a própria recorrência para substituir para $T(m)$, $m < n$ até que todos os termos tenham sido substituídos por fórmulas envolvendo apenas $T(0)$ ou o caso base.

Análise da função FATORIAL

Seja a seguinte função para calcular o fatorial de n:

```
FATORIAL(n)
    ▷ F: variável auxiliar
1   if n > 0
2   then F ← n × FATORIAL(n − 1)
3   else F ← 1
4   return F
```

Seja a seguinte equação de recorrência para esta função:

$$T(n) = \begin{cases} d & n = 1 \\ c + T(n-1) & n > 1 \end{cases}$$

Esta equação diz que quando $n = 1$ o custo para executar FATORIAL é igual a d. Para valores de n maiores que 1, o custo para executar FATORIAL é c mais o custo para executar $T(n-1)$.

Resolvendo a equação de recorrência

Esta equação de recorrência pode ser expressa da seguinte forma:

$$\begin{aligned} T(n) &= c + T(n-1) \\ &= c + (c + T(n-2)) \\ &= c + c + (c + T(n-3)) \\ \vdots &= \vdots \\ &= c + c + \cdots + (c + T(1)) \\ &= \underbrace{c + c + \cdots + c}_{n-1} + d \end{aligned}$$

Em cada passo, o valor do termo T é substituído pela sua definição (ou seja, esta recorrência está sendo resolvida pelo método da expansão). A última equação mostra que depois da expansão existem $n-1$ c's, correspondentes aos valores de 2 até n. Desta forma, a recorrência pode ser expressa como:

$$T(n) = c(n-1) + d = O(n).$$

Alguns somatórios úteis

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1} (a \neq 1)$$

$$\sum_{i=0}^{k} 2^k = 2^{k+1} - 1$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

Algumas recorrências básicas: Caso 1

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + 1 \quad (n \geq 2) \\ T(1) &= 0 \qquad\qquad\quad (n = 1) \end{aligned}$$

Vamos supor que:

$$n = 2^k \Rightarrow k = \log n$$

Resolvendo por expansão temos:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 1 \\ &= (T(2^{k-2}) + 1) + 1 \\ &= (T(2^{k-3}) + 1) + 1 + 1 \\ \vdots &\quad \vdots \\ &= (T(2) + 1) + 1 + \cdots + 1 \\ &= (T(1) + 1) + 1 + \cdots + 1 \\ &= 0 + \underbrace{1 + \cdots + 1}_{k} \\ &= k \end{aligned}$$

$$\begin{aligned} T(n) &= \log n \\ T(n) &= O(\log n) \end{aligned}$$

Algumas recorrências básicas: Caso 2

$$\begin{array}{lcll} T(n) & = & 2T(\frac{n}{2}) + n & (n \geq 2) \\ T(1) & = & 0 & (n = 1) \end{array}$$

Vamos supor que $n = 2^k \Rightarrow k = \log n$. Resolvendo por expansão temos:

$$\begin{array}{rcl} T(2^k) & = & 2T(2^{k-1}) + 2^k \\ & = & 2(2T(2^{k-2}) + 2^{k-1}) + 2^k \\ & = & 2(2(2T(2^{k-3}) + 2^{k-2}) + 2^{k-1}) + 2^k \\ \vdots & & \vdots \\ & = & 2(2(\cdots(2(2T(1) + 2^2) + 2^3) + \cdots) + 2^{k-1}) + 2^k \\ & = & (k-1)2^k + 2^k \\ & = & k2^k \end{array}$$

$$\begin{array}{rcl} T(n) & = & n \log n \\ T(n) & = & O(n \log n) \end{array}$$

Teorema Mestre

Recorrências da forma

$$T(n) = aT(\tfrac{n}{b}) + f(n),$$

onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva podem ser resolvidas usando o Teorema Mestre. Note que neste caso não estamos achando a forma fechada da recorrência mas sim seu comportamento assintótico.

Teorema Mestre

Sejam as constantes $a \geq 1$ e $b > 1$ e $f(n)$ uma função definida nos inteiros não-negativos pela recorrência:

$$T(n) = aT(\tfrac{n}{b}) + f(n),$$

onde a fração n/b pode significar $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. A equação de recorrência $T(n)$ pode ser limitada assintoticamente da seguinte forma:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $\boxed{T(n) = \Theta(n^{\log_b a})}$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.
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 n suficientemente grande, então $\boxed{T(n) = \Theta(f(n))}$.

Comentários sobre o teorema Mestre

- Nos três casos estamos comparando a função $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a solução da recorrência é determinada pela maior das duas funções.

- Por exemplo:
  - No primeiro caso a função $n^{\log_b a}$ é a maior e a solução para a recorrência é $T(n) = \Theta(n^{\log_b a})$.
  - No terceiro caso, a função $f(n)$ é a maior e a solução para a recorrência é $T(n) = \Theta(f(n))$.
  - No segundo caso, as duas funções são do mesmo "tamanho." Neste caso, a solução fica multiplicada por um fator logarítmico e fica da forma $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

Tecnicalidades sobre o teorema Mestre

- No primeiro caso, a função $f(n)$ deve ser não somente menor que $n^{\log_b a}$ mas ser polinomialmente menor. Ou seja, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator de n^{ϵ}, para alguma constante $\epsilon > 0$.

- No terceiro caso, a função $f(n)$ deve ser não somente maior que $n^{\log_b a}$ mas ser polinomialmente maior e satisfazer a condição de "regularidade" que $af(n/b) \leq cf(n)$. Esta condição é satisfeita pela maior parte das funções polinomiais encontradas neste curso.

Tecnicalidades sobre o teorema Mestre

- Teorema **não** cobre todas as possibilidades para $f(n)$:
  - Entre os casos 1 e 2 existem funções $f(n)$ que são menores que $n^{\log_b a}$ mas não são polinomialmente menores.
  - Entre os casos 2 e 3 existem funções $f(n)$ que são maiores que $n^{\log_b a}$ mas não são polinomialmente maiores.

  ➜ Se a função $f(n)$ cai numa dessas condições ou a condição de regularidade do caso 3 é falsa, então não se pode aplicar este teorema para resolver a recorrência.

Uso do teorema: Exemplo 1

$$T(n) = 9T(\tfrac{n}{3}) + n.$$

Temos que,

$$a = 9, b = 3, f(n) = n$$

Desta forma,

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

Como $f(n) = O(n^{\log_3 9 - \epsilon})$, onde $\epsilon = 1$, podemos aplicar o caso 1 do teorema e concluir que a solução da recorrência é

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

Uso do teorema: Exemplo 2

$$T(n) = T(\tfrac{2n}{3}) + 1.$$

Temos que,

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

Desta forma,

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

O caso 2 se aplica já que $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$. Temos, então, que a solução da recorrência é

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

Uso do teorema: Exemplo 3

$$T(n) = 3T(\tfrac{n}{4}) + n \log n$$

Temos que,

$$a = 3, b = 4, f(n) = n \log n.$$

Desta forma,

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793}).$$

Como $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, onde $\epsilon \approx 0.2$, o caso 3 se aplica se mostrarmos que a condição de regularidade é verdadeira para $f(n)$.

Uso do teorema: Exemplo 3

Para um valor suficientemente grande de n:

$$af(\tfrac{n}{b}) = 3(\tfrac{n}{4})\log(\tfrac{n}{4}) \leq (\tfrac{3}{4})n\log n = cf(n).$$

para $c = \frac{3}{4}$. Consequentemente, usando o caso 3, a solução para a recorrência é

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

Uso do teorema: Exemplo 4

$$T(n) = 2T(\tfrac{n}{2}) + n \log n.$$

Temos que,

$$a = 2, b = 2, f(n) = n \log n.$$

Desta forma,

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

Aparentemente o caso 3 deveria se aplicar já que $f(n) = n \log n$ é assintoticamente maior que $n^{\log_b a} = n$. Mas no entanto, não é polinomialmente maior. A fração $f(n)/n^{\log_b a} = (n \log n)/n = \log n$ que é assintoticamente menor que n^{ϵ} para toda constante positiva ϵ. Consequentemente, a recorrência cai na situação entre os casos 2 e 3 onde o teorema não pode ser aplicado.

Uso do teorema: Exercício 5

$$T(n) = 4T(\tfrac{n}{2}) + n.$$

Uso do teorema: Exercício 6

$$T(n) = 4T(\tfrac{n}{2}) + n^2.$$

Uso do teorema: Exercício 7

$$T(n) = 4T(\tfrac{n}{2}) + n^3.$$

Uso do teorema: Exercício 8

O tempo de execução de um algoritmo A é descrito pela recorrência

$$T(n) = 7T(\tfrac{n}{2}) + n^2.$$

Um outro algoritmo A' tem um tempo de execução descrito pela recorrência

$$T'(n) = aT'(\tfrac{n}{4}) + n^2.$$

Qual é o maior valor inteiro de a tal que A' é assintoticamente mais rápido que A?

Notação assintótica em funções

Normalmente, a notação assintótica é usada em fórmulas matemáticas. Por exemplo, usando a notação O pode-se escrever que $n = O(n^2)$. Também pode-se escrever que

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n).$$

Como se interpreta uma fórmula como esta?

Notação assintótica em funções

- Notação assintótica sozinha no lado direito de uma equação, como em $n = O(n^2)$.
  - Sinal de igualdade significa que o lado esquerdo é um membro do conjunto $O(n^2)$;
  - $n \in O(n^2)$ ou $n \subseteq n^2$.
- Nunca deve-se escrever uma igualdade onde a notação O aparece sozinha com os lados trocados.
  - Caso contrário, poderia se deduzir um absurdo como $n^2 = n$ de igualdades como em $O(n^2) = n$.
- Quando se trabalha com a notação O e em qualquer outra fórmula que envolve quantidades não precisas, o sinal de igualdade é unidirecional.
  - Daí vem o fato que o sinal de igualdade (`"="`) realmente significa $\in$ ou $\subseteq$, usados para inclusão de conjuntos

Notação assintótica em funções

Se uma notação assintótica aparece numa fórmula, isso significa que essa notação está substituindo uma função que não é importante definir precisamente (por algum motivo). Por exemplo, a equação

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

significa que

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

onde $f(n)$ é alguma função no conjunto $\Theta(n)$. Neste caso, $f(n) = 3n + 1$ que de fato está em $\Theta(n)$.

Notação assintótica em funções

O uso da notação assintótica desta forma ajuda a eliminar detalhes que não são importantes. Por exemplo, pode-se expressar uma equação de recorrência como:

$$T(n) = 2T(n-1) + \Theta(n).$$

Se se deseja determinar o comportamento assintótico de $T(n)$ então não é necessário determinar exatamente os termos de mais baixa ordem. Entende-se que eles estão incluídos numa função $f(n)$ expressa no termo $\Theta(n)$.

Notação assintótica em funções

Em alguns casos, a anotação assintótica aparece do lado esquerdo de uma equação como em:

$$2n^2 + \Theta(n) = \Theta(n^2).$$

A interpretação de tais equações deve ser feita usando a seguinte regra:

- É possível escolher uma função $f(n)$ para o lado esquerdo da igualdade de tal forma que existe uma função $g(n)$ para o lado direito que faz com que a equação seja válida.
- O lado direito da igualdade define um valor não tão preciso quanto o lado esquerdo. Por exemplo,

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) \quad (1)$$

$$2n^2 + \Theta(n) = \Theta(n^2). \quad (2)$$

Notação assintótica em funções

As equações (1) e (2) podem ser interpretadas usando a regra acima:

- A equação (1) diz que existe alguma função $f(n) \in \Theta(n)$ tal que

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

  para todo n.

- A equação (2) diz que para qualquer função $g(n) \in \Theta(n)$, existe uma função $h(n) \in \Theta(n^2)$ tal que

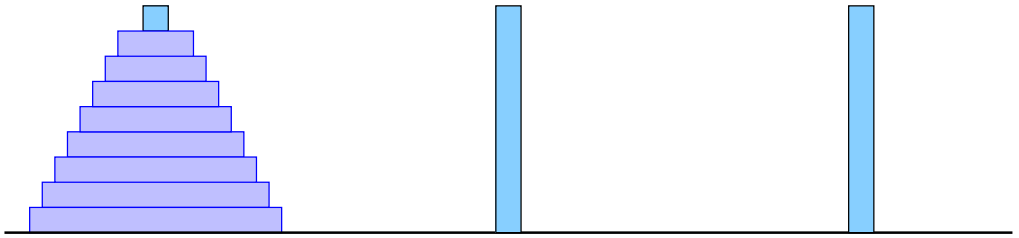
$$2n^2 + g(n) = h(n)$$

  para todo n. Note que esta interpretação implica que $2n^2+3n+1 = \Theta(n^2)$, que é o que estas duas equações querem dizer.

Modelagem usando equação de recorrência
Torre de Hanoi

Edouard Lucas (1842–1891), matemático francês. Propôs o jogo "Torre de Hanoi" em 1883. Escreveu o trabalho de matemática recreativa *Récréations Mathématiques* em quatro volumes (1882–94) que se tornou um clássico.

- Configuração inicial:
  - O jogo começa com um conjunto de oito discos empilhados em tamanho decrescente em uma das três varetas.



Modelagem usando equação de recorrência
Torre de Hanoi

- Objetivo:
  - Transferir toda a torre para uma das outras varetas, movendo um disco de cada vez, mas nunca movendo um disco maior sobre um menor.

- Soluções particulares:
  - Seja $T(n)$ o número mínimo de movimentos para transferir n discos de uma vareta para outra de acordo com as regras definidas no enunciado do problema.
  - Não é difícil observar que:

$$
\begin{aligned}
T(0) &= 0 && \text{[nenhum movimento é necessário]} \\
T(1) &= 1 && \text{[apenas um movimento]} \\
T(2) &= 3 && \text{[três movimentos usando as duas varetas]}
\end{aligned}
$$

Modelagem usando equação de recorrência
Torre de Hanoi

- Generalização da solução:
  - Para três discos, a solução correta é transferir os dois discos do topo para a vareta do meio, transferir o terceiro disco para a outra vareta e, finalmente, mover os outros dois discos sobre o topo do terceiro.
  - Para n discos:
    1. Transfere-se os $n-1$ discos menores para outra vareta (por exemplo, a do meio), requerendo $T(n-1)$ movimentos.
    2. Transfere-se o disco maior para a outra vareta (um movimento).
    3. Transfere-se os $n-1$ discos menores para o topo do disco maior, requerendo-se $T(n-1)$ movimentos novamente.

Modelagem usando equação de recorrência
Torre de Hanoi

- Equação de recorrência para este problema pode ser expressa por:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 2T(n-1)+1, \quad \text{para } n > 0 \end{aligned}$$

Modelagem usando equação de recorrência
Torre de Hanoi

Para pequenos valores de n temos:

n	0	1	2	3	4	5	6
$T(n)$	0	1	3	7	15	31	63

Esta recorrência pode ser expressa por

$$T(n) = 2^n - 1.$$

Modelagem usando equação de recorrência
Torre de Hanoi

Provando por indução matemática temos:

Caso base. Para $n = 0$ temos que $T(0) = 2^0 - 1 = 0$, que é o valor presente na equação de recorrência.

Indução. Vamos supor que a forma fechada seja válida para todos os valores até $n = k$, i.e., $T(k) = 2^k - 1$. Vamos provar que esta forma fechada é válida para $n = k + 1$, i.e., $T(k + 1) = 2^{k+1} - 1$.

$$T(k+1) = 2T(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1.$$

$\therefore$ A forma fechada proposta também é válida para n.

Modelagem usando equação de recorrência
Estratégia para resolução da equação

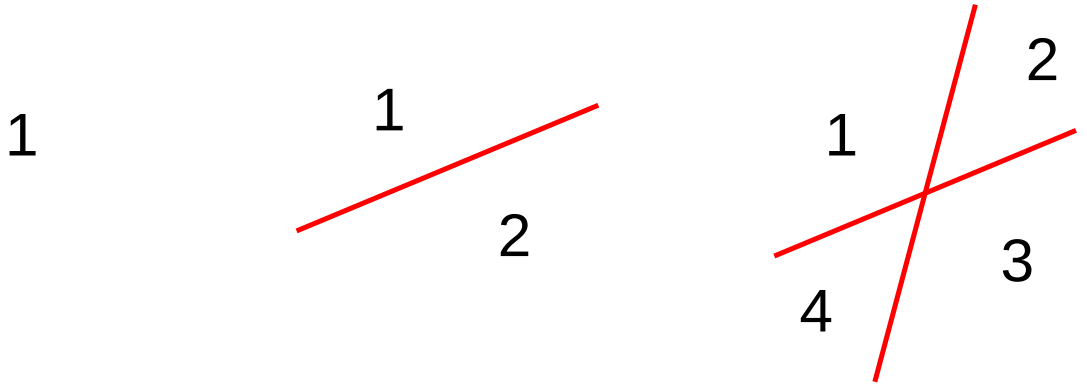
A recorrência da Torre de Hanoi aparece em várias aplicações de todos os tipos. Normalmente, existem três etapas para achar uma forma fechada para o valor de $T(n)$:

1. Analisar pequenos casos. Com isto podemos ter um entendimento melhor do problema e, ao mesmo tempo, ajudar nos dois passos seguintes.
2. Achar e provar uma recorrência para o valor de $T(n)$.
3. Achar e provar uma forma fechada para a recorrência.

Modelagem usando equação de recorrência
Linhas no plano

- Problema:
  - Qual é o número máximo de regiões L_n determinado por n retas no plano?

➜ Lembre-se que um plano sem nenhuma reta tem uma região, com uma reta tem duas regiões e com duas retas têm quatro regiões.



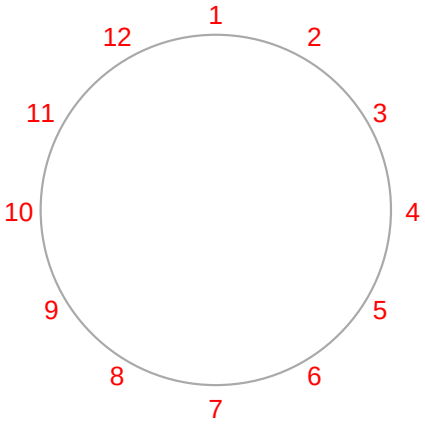
$L_0 = 1$ $L_1 = 2$ $L_2 = 4$

Modelagem usando equação de recorrência
O número de Josephus

Durante os anos em que serviu no exército romano, Josephus vivia num alojamento com vários outros soldados. Toda semana devia ser feita uma limpeza geral que sempre tomava o dia todo. Os soldados decidiram entre si que toda semana seria "sorteado" um soldado que não faria a limpeza. O sorteio seria feito da seguinte forma: os soldados formariam um círculo, percorreriam o círculo no sentido horário e cada segundo soldado seria escolhido para a limpeza. O soldado que ficasse por último seria o "sorteado," ou seja, não participaria da limpeza. Nos dias que Josephus queria tirar um dia de folga ele rapidamente calculava onde ele deveria ficar nesse círculo, apenas conhecendo o número total de soldados que participariam da limpeza naquele dia.

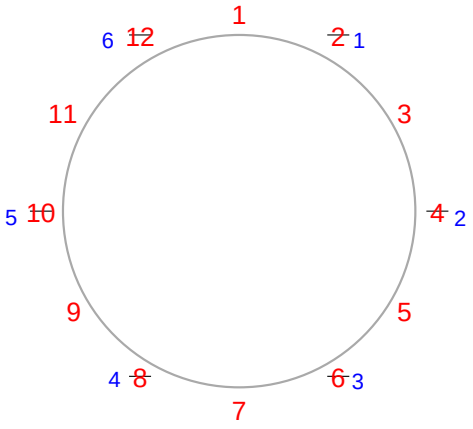
Modelagem usando equação de recorrência
O número de Josephus

Por exemplo, suponha a configuração inicial com 12 soldados, conforme figura ao lado.

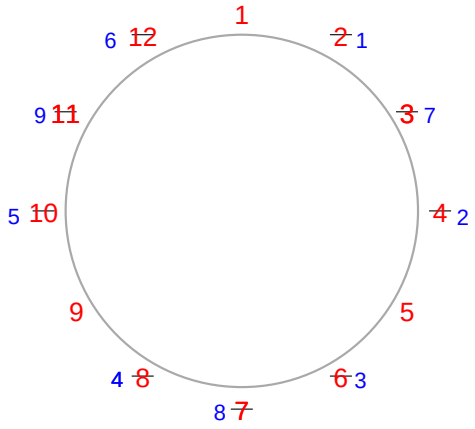


Neste caso, na primeira rodada seriam selecionados os soldados 2, 4, 6, 8, 10 e 12, na segunda rodada os soldados 3, 7 e 11, e, finalmente na terceira rodada os soldados 5 e 11, sendo que o soldado 9 seria o sorteado.

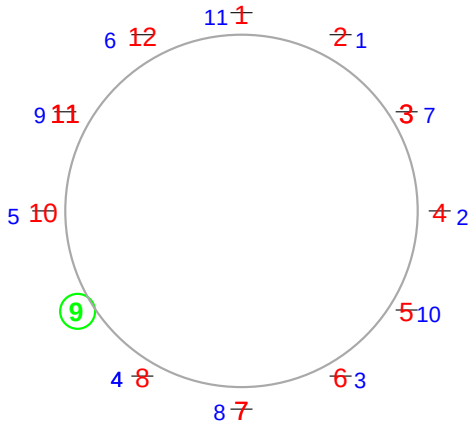
1^a rodada



2^a rodada



3^a rodada



Princípios para análise de algoritmos
Tempo de execução

1. Comando de Atribuição, Leitura, ou Escrita: $O(1)$
   - Exceções: linguagens que permitem atribuições que envolvem vetores de tamanho arbitrariamente grandes.
2. Sequência de comandos:
   - Maior tempo de execução de qualquer comando da sequência.
3. Comando de decisão:
   - Avaliação da condição: $O(1)$.
   - Comandos executados dentro do comando condicional: Regra 2.

Princípios para análise de algoritmos
Tempo de execução

4. Anel:
   - Avaliação da condição de terminação: $O(1)$.
   - Comandos executados dentro do corpo do anel: Regra 2.
   - ➜ Os dois itens ficam multiplicados pelo número de iterações do anel.

5. Programa com procedimentos não recursivos:
   - Tempo de execução de cada procedimento deve ser computado separadamente um a um, iniciando com os procedimentos que não chamam outros procedimentos.
   - Em seguida, devem ser avaliados os procedimentos que chamam os procedimentos que não chamam outros procedimentos, utilizando os tempos dos procedimentos já avaliados.
   - Este processo é repetido até chegar no programa principal.

Princípios para análise de algoritmos
Tempo de execução

6. Programa com procedimentos recursivos:
   - Para cada procedimento é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento.
   - Obter e resolver a equação de recorrência.

Operações com a notação O

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \cdot O(f(n)) &= O(f(n)) \qquad c = \text{constante} \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

Operações com a notação O

Duas operações importantes:

- Regra da soma: $O(f(n)) + O(g(n))$
  - Suponha três trechos de programas cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$.
  - O tempo de execução dos dois primeiros trechos é $O(\max(n, n^2))$, que é $O(n^2)$.
  - O tempo de execução de todos os três trechos é então

    $$O(\max(n^2, n \log n)),$$

    que é $O(n^2)$.

- Regra do produto: $O(f(n))O(g(n))$
  - Produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é

    $$n \log n + kn + O(\sqrt{n} \log n).$$