TEORIA DE ALGORITMOS

Procedimento Efetivo

Um procedimento é uma sequência finita de instruções bem definidas (sem ambiguidade), cada uma das quais podem ser mecanicamente executadas num tempo finito com uma quantidade finita de esforço. É um conjunto de regras que nos diz precisamente como se comportar.

Efetividade

Cada instrução deve ser efetiva, isto é, os componentes de cada instrução devem ser suficientemente básicos e bem claros tal que a instrução possa ser executada num tempo finito e com esforço finito.

Definitude

Cada passo do procedimento deve ser bem definido (não ambíguo):

IF N=M DISPLAY S AND STOP

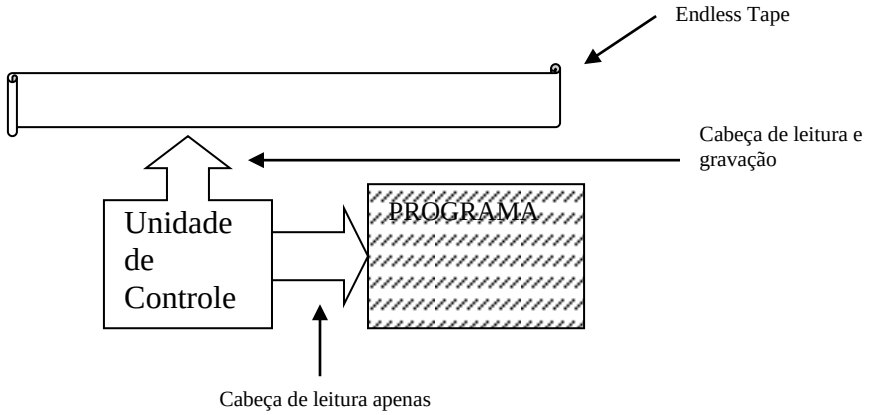
Esta instrução não é bem definida pois não especifica o que fazer se n < m ou n > m. Cada passo deve ser finito, não dependendo de uma quantidade infinita de dados para sua complementação ou da complementação de um processo infinito. É requerido que o processo eventualmente termine se aplicado a qualquer dado que esteja dentro do seu domínio de aplicabilidade. Seu comportamento, quando aplicado a dados fora de seu domínio não é de interesse, exceto que o processo não deve dar respostas incorretas. Contudo pode acontecer de ele não terminar ou terminar de uma maneira não esperada.

Descrição

O processo deve possuir uma descrição finita em tamanho e pode ser executado mecanicamente, de uma maneira finita. Por exemplo, uma lista infinita de instruções para uma máquina não pode ser considerada como especificando um processo efetivo, a menos que se possa ser substituído por uma lista finita equivalente.

Máquina de Turing

Uma máquina de Turing (Alan Turing[1]), é definida como sendo composta de:
1. Unidade de Memória (uma fita magnética infinita nas duas pontas.
2. Unidade de Controle (que executa um programa na memória)
3. Programa



A memória é uma fita infinita para a possibilidade do programa nunca parar.
A noção de procedimento efetivo é um conceito informal, e é equivalente a um conceito formal (máquina de Turing).

Um procedimento para resolver um dado problema existe se se puder programar uma máquina de Turing para resolvê-lo. Qualquer computação que pode ser efetuada por um computador pode ser simulada numa máquina de Turing. A inversa é falsa porque a máquina de Turing tem memória infinita.
Podemos especificar um procedimento informalmente e, invocando a Tese de Church[2], concluir que existe um programa formal correspondente.
Qualquer procedimento pode ser representado em linguagem de Turing (pode ser executado pela máquina de Turing). Essa é a Tese de Church, que não pode ser provada porque não há uma definição formal de *procedimento*.
Pode ser provado que os programas escritos em linguagem de Turing podem ser escritos em programas equivalentes em outras linguagens de programação.

Algoritmo

Um algoritmo é um procedimento (programa) que sempre termina após um número finito de passos para qualquer conjunto miscível de entradas, havendo-as ou não.

É geralmente aceito que qualquer uma das definições formais (Turing) de algoritmos corresponde à nossa noção intuitiva (definição informal), pois não há como provar isso. Até agora, qualquer algoritmo que satisfaça uma definição informal pode ser expresso em qualquer sistema formal.

[1] Alan Turing (1912-1954), matemático inglês, criador do dispositivo matemático que leva seu nome.
[2] Alonzo Church (1903-1995), matemático americano, conjecturou que "qualquer coisa computável pode ser computada por uma máquina de Turing.

Qualquer coisa que um computador possa fazer pode ser precisamente descrita.
Qualquer procedimento que possa ser precisamente descrito pode ser programado para execução num computador.

A primeira frase é verdadeira. Mostraremos que a segunda frase não é verdadeira.

Indução Matemática

Indução como um termo matemático refere-se a uma técnica para provar teoremas sobre números inteiros. Na prova de uma proposição por indução temos a **base** e o **passo indutivo**.

Base – A proposição é válida para todos os objetos com o número 1.
Passo Indutivo – Se a proposição vale para todos os objetos com número n, então ela vale para todos os objetos com número n+1.

Exemplo 1.

Seja T(n) o seguinte teorema sobre inteiros:

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

Então, uma prova por indução consiste dos dois passos seguintes:

1) Prove que T(1) é verdadeiro.
2) Prove que se T(1), T(2), ..., T(n-1), T(n) são todos verdadeiros, então T(n+1) será verdadeiro também.

I) $\sum_{i=1}^{1} i = \frac{1(1+1)}{2} = 1$. Verdade!

II) Assumimos agora que para qualquer k ≤ n o teorema T é verdadeiro. Daí, podemos concluir que é verdade que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ [A] e desejamos provar que $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$ [B].

Adicionando (n+1) nos dois lados da equação [A], obtemos

$$\sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1) \quad [C]$$

O lado esquerdo desta equação é igual ao lado esquerdo da equação a ser provada ([B]), pois

$1+2+3+...+n = \sum_{i=1}^{n} i$ e

$$1+2+3+...+n+(n+1) = \sum_{i=1}^{n} i + (n+1) = \sum_{i=1}^{n+1} i$$

Manipulado o lado direito de [C] temos

$$\frac{n(n+1)+2(n+1)}{2} = \frac{n^2+n+2n+2}{2} = \frac{n^2+3n+2}{2} \quad [D]$$

Note que o numerador da última fração é uma equação do segundo grau que, ao ser resolvida (pela fórmula de Baskara) nos dá como raízes o -1 e o -2.

Como qualquer equação do segundo grau cujas raízes são *a* e *b* podem ser representadas por (x-*a*)(x-*b*), usando isso podemos reescrever o numerador da última fração em [D], ficando:

$$\frac{n^2+n+2n+2}{2}=\frac{(n+1)(n+2)}{2} \quad [E]$$

Ora, o lado direito de [E] é exatamente a parte direita de [B], o que satisfaz o cálculo de T(n+1) na prova de indução e, por isso, provamos o teorema T.

Exemplo 2.

O Passo Indutivo.

Há pouca dificuldade em se encontrar a soma dos n primeiros números inteiros. Vimos que a fórmula n(n+1)/2 nos dá o resultado que pode ser descoberto e provado de muitas maneiras, uma das quais vimos no exemplo 1.
É mais difícil encontrar a fórmula para a soma dos n primeiros quadrados:

$1+4+9+16+...+n^2$.

É bem natural procurar alguma espécie de paralelismo entre as duas regras acima e observá-las juntas:

n	1	2	3	4	5	6
1+2+..+n	1	3	6	10	15	21
$1^2+2^2+...n^2$	1	5	14	30	55	91

Como estão relacionadas as duas últimas linhas? Examinemos entre as duas com a segunda sobre a primeira:

n	1	2	3	4	5	6
$\frac{1^2+2^2+...n^2}{1+2+...+n}$	1	5/3	7/3	3	11/3	13/3

Aqui, a regra é óbvia e é quase impossível não enxergá-la se reescrevermos as frações assim:

$$\frac{3}{3}\frac{5}{3}\frac{7}{3}\frac{9}{3}\frac{11}{3}\frac{13}{3}$$

Os numeradores claramente é a sequência de números ímpares começando em 3. Com n=1, a expressão que gera uma sequência impar é 2n+1. Então, poderemos dizer que:

$\frac{1^2+2^2+...+n^2}{1+2+...+n}=\frac{2n+1}{3}$. Como já conhecemos a expressão que calcula a soma dos n primeiros inteiros, temos que:

$$\frac{1^2+2^2+...+n^2}{\frac{n(n+1)}{2}}=\frac{2n+1}{3}$$

O n(n+1)/2 está dividindo na primeira fração. Passando ele para o lado direito, ele vai para lá multiplicando. Então:

$$1^2 + 2^2 + ... + n^2 = \frac{2n+1}{3} \bullet \frac{n(n+1)}{2} = \frac{n(n+1)(2n+1)}{6} \qquad [F]$$

Talvez você ficou tentado a levar o n(n+1)/2 para outro lado invertendo o numerador com o denominador. Não, o n(n+1)/2 é simplesmente o *b* na expressão $\frac{a}{b} = c$, onde fazemos *a = c.b*. Você só faz a inversão quando *b* é uma fração que vai ser operada com o *a*, assim: $\frac{\frac{a}{b}}{d} = c \Rightarrow \frac{a}{1} \times \frac{d}{b} = c \Rightarrow a \times \frac{d}{b} = c$

Se você continuar a operação para isolar *a* vai ver que é mais rápido levar *b/d* multiplicando o lado direito:

$a \times \frac{d}{b} = c \Rightarrow a \times d = c \times b \Rightarrow a = c \times \frac{b}{d}$. Viu?

Será que a expressão [F] é verdadeira para qualquer n?
Certamente que é verdade para os casos particulares com n = 1, 2, 3, 4, 5, 6. Ela vale para n=7? Vamos ver. Estamos dizendo que

$$1 + 4 + 9 + 16 + 25 + 36 + 49 = \frac{7(7+1)(2 \times 7 + 1)}{6} \Rightarrow 140 = \frac{56 \times 15}{6} = \frac{840}{6} = 140$$

E, de fato, valeu.
Poderíamos, é claro, testar para n=8, etc., mas a tentação não é tão forte. Somos inclinados a acreditar que a expressão valerá para os próximos casos também. Como poderíamos testar mais eficientemente a conjectura?

Se a conjectura é verdadeira para todos os casos, ela deve ser independente da variação dos casos, deve valer na transição de um caso para o outro.

Supostamente $1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$ [G]. Ainda, se esta fórmula é válida no caso geral, ela deve valer no próximo caso, n+1:

$$1^2 + 2^2 + ... + n^2 + (n+1)^2 = \frac{(n+1)(n+2)(2n+3)}{6} \qquad [H]$$

Aqui está uma oportunidade de checar eficientemente a conjectura. Vamos subtrair [G] de [H] teremos. Subtraindo o lado esquerdo de [G] do lado esquerdo de [H] ficamos com $(n+1)^2$ do lado esquerdo que é igual ao lado direito de [H] menos o lado direito de [G]:

$$(n+1)^2 = \frac{(n+1)(n+2)(2n+3)}{6} - \frac{n(n+1)(2n+1)}{6}$$

É esta consequência da conjectura verdade? Vamos tentar simplificar o lado direito da equação:

$$\frac{(n+1)(n+2)(2n+3)}{6} - \frac{n(n+1)(2n+1)}{6} = \frac{(n+1)}{6}\left[(n+2)(2n+3) - n(2n+1)\right] =$$

$$= \frac{n+1}{6}\left[2n^2 + 3n + 4n + 6 - 2n^2 - n\right] = \frac{n+1}{6}\left[6n + 6\right] = \frac{n+1}{6} \times 6\left[n+1\right] = (n+1)(n+1) = (n+1)^2$$

A consequência examinada é incontestavelmente verdadeira. A conjectura passou por um teste severo.

O Passo Demonstrativo

A verificação de qualquer consequência aumenta nossa confiança na conjectura, mas a verificação da consequência recentemente examinada pode fazer mais: ela pode provar a conjectura:

É supostamente verdade que $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

É incontestavelmente verdade que

$$1^2 + 2^2 + \ldots + n^2 + (n+1)^2 = \frac{(n+1)(n+2)(2n+3)}{6}$$

No primeiro exemplo foi dada a fórmula para a prova. Mas, pode acontecer de se ter que conseguir uma fórmula, a qual pode ser muito trabalhosa.
Como se viu, uma vez obtida a fórmula que supostamente nos dá a proposição, basta prová-la depois indutivamente. Mas, para obter-se a fórmula tem-se que adivinhar fórmula e fazer testes.
É o bastante saber duas coisas sobre a conjectura:

1. Ela vale para n=1.
2. Sendo válida para n, vale também para n+1. Então a conjectura é válida para todos os inteiros.

Existem também outros tipos de indução, nas quais os objetos manipulados não são mais números inteiros, como em provas indutivas de programas, onde se manipula instruções. Se bem que nada impede que estas instruções, por sua vez, manipulem números inteiros direta ou indiretamente (através do "tamanho" de um objeto, por exemplo). É o que veremos a seguir.

Indução em Programas (Procedimentos)

Vamos agora obter uma analogia para desenvolvimento de algoritmos. Queremos obter um algoritmo **A** que solucionará um problema com entrada **x** de tamanho **n**. Postulamos que a entrada pode ser representada como **n** elementos individuais, isto é, x = (x_1, x_2, ...,x_n). Então, a técnica de indução consiste nos seguintes passos:

1. Encontre um algoritmo B que resolva o problema para n=1.
2. Encontre um algoritmo C que tome como entrada $\overline{x_i}$ mais a solução para x_2, x_3, ..., x_i e mais x_{i+1} e produza a solução $\overline{x_{i+1}}$.

Dados estes dois algoritmos B e C, a solução $\overline{x_n}$ para o problema original é encontrada por:

Algoritmo A:

inicio

$\overline{x_1} \leftarrow B(x_1);$

$i \leftarrow 1;$

$Enquanto(i < n)$

$faça\{\overline{x_{i+1}}; i \leftarrow C(\overline{x_i}, x_{i+1}); i+1\}$

fim

Outro Exemplo.

Encontrando o maior elemento num conjunto X, que é uma n-upla, supondo todos os elementos diferentes entre si, queremos encontrar a localização e o valor do maior elemento na upla.

Dado este problema, vamos desenvolver um algoritmo usando a técnica de indução. Primeiro, o algoritmo B, é bastante trivial, desde que numa 1-upla o único elemento é exatamente o maior, teremos que

Algoritmo B (Encontre o maior elemento numa unpla)
inicio:
valmax ← x(1);
indmax ← 1;
fim.

Agora, o critério para o algoritmo C é que *valmax* contém o valor máximo entre os i primeiros valores e que *indmax* aponte para sua localização. Adicionalmente, é-nos dado x(i+1) para o qual não temos ainda uma ordenação. Duas possibilidades existem:

1) x(i+1) é maior que *valmax*, caso em que devemos atualizar nossa solução.
2) x(i+1) é menor que *valmax*, o que significa que a velha solução é também a atual.

Daí,

Algoritmo C (Parte indutiva do elemento máximo)
inicio:
Se x(i+1) > valmax então
valmax, indmax ← x(i+1), i+1;
fim.

Juntando estes dois algoritmos como foi ilustrado acima, resulta na seguinte solução:

Algoritmo A
procedimento maxelemento(x, n; x)
inicio { Encontre o maior elemento}
valmax ← x(1);
indmax ← 1;
i ← 1;
Enquanto (i < n) faça
inicio {verifica elemento i+1}
Se x(i+1) > valmax então
valmax, indmax ← x(i+1), i+1;
i ← i+1;
fim
fim

Consistente com nossa discussão anterior, o bloco de comandos na iteração dependente do tamanho de n é selecionado como a construção dominante. Mais especificamente, a comparação "x(i+1) > valmax" pode ser tomada como a instrução dominante. A comparação é executada exatamente (n-1) vezes.

Nos exemplos que se seguem, as demonstrações serão "mais matemáticas" do que a anterior e também a indução será usada para testar programas quanto à sua finitude (se termina ou não) e quanto à sua resposta (correta ou não). Para cada programa específico isto pode ser feito.

O Algoritmo de Euclides para o Máximo Divisor Comum

Passo 1: (m, n)
Adote como valores iniciais de x e y os valores m e n, respectivamente.
Passo 2:
Adote como valor de r o resto da divisão do valor de x pelo valor de y.
Passo 3:
Adote como o novo valor de x o valor de y, e como novo valor de y o valor de r.
Passo 4:
Se o valor de r é zero então o valor de x é o valor do MDC, e fim. Caso contrário volte para o passo 2.

No caso do algoritmo de Euclides, podemos mostrar que a sequência de cálculos é finita, provando a seguinte proposição:

Se no passo 2 do procedimento os valores de x e y são inteiros e positivos, então os passos 2, 3 e 4 serão executados um número finito de vezes, com os cálculos terminando no passo 4.

A demonstração é por indução sobre o valor de y.
Se y = 1, então teremos, pela execução do passo 2, r = 0. Consequentemente, os passos 2, 3 e 4 serão executados uma única vez e os cálculos terminam no passo 4.

Suponhamos agora que a proposição é verdadeira para qualquer x > 0 e qualquer y, com 1 <= y < k, e demonstraremos que ela é verdadeira para y = k.

Por definição do resto da divisão de inteiros positivos (r = D – qd), teremos após a execução do passo 2: 0 <= r < k.
Se r = 0, então a execução termina, como anteriormente, numa única vez.
Se r > 0, então, com a execução dos passos 3 e 4 teremos x = k > 0 e y = r, com 0 < r < k e a execução volta para o passo 2.
Por hipótese de indução, os passos 2, 3 e 4 serão executados um número finito p de vezes, com os cálculos terminando no passo 4. Ao todo teremos então p+1 (um do passo 1) execuções para y = k. Notemos ainda que os valores iniciais x = m e y =n resultantes da execução do passo 1 satisfazem as condições da proposição acima. Podemos concluir, portanto, que a execução do algoritmo de Euclides termina para quaisquer inteiros m e n.
A prova de que o algoritmo realmente devolve o MDC(m,n) fica adiada para a seção Asserções.

Iteração

Iteração é a execução repetitiva de uma dada instrução até que uma certa condição seja satisfeita. Note que uma instrução pode ser formada de várias outras menores. A execução iterativa duma instrução n vezes tem o mesmo efeito que escrever-se a instrução n vezes e executar uma vez cada uma. Porém, nem sempre esse n é conhecido e muitas vezes seu tamanho proíbe que se repita (escrevendo) uma instrução n vezes. Por isso, escrevemos um número m de linhas de instruções, em que esse m é muito menor que n e iteramos, isto é, repetimos as instruções tantas vezes quanto necessário para equivaler a escrever a instrução n vezes. Sem iteração, alguns programas, como o algoritmo de Euclides não poderiam ser escritos, para certos valores a partir de um máximo permitido.
Vimos que o algoritmo de Euclides é executado p+1 vezes. Para um computador que tivesse uma memória com p palavras não haveria, portanto, condições de calcular o MDC nele pelo algoritmo de Euclides. Veremos que outro método (recursão) diferente de iteração, mas que não é melhor, poderá ser usado.

Comandos Iterativos:

1) ENQUANTO E(m,x) FAÇA x ← g(m,x);
2) REPITA x ← g(m,x); ATÉ QUE E(m,x);

São os dois principais comandos iterativos usados aqui, onde:
m é uma n-upla de argumentos iniciais.
x é uma p-upla de variáveis.
g é uma função (computação) que manuseia estes valores, primeiro inicializando com m e depois calculando os novos componentes de x.
E é a condição, que depende de m e de x.

Exemplo:

```
REPITA
 r ← resto(x,y);
 x,y ← y,r;
ATÉ QUE r=0;
```

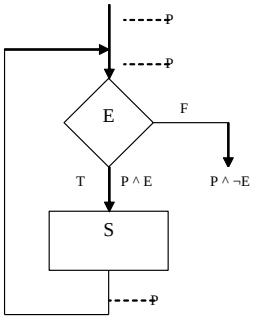
```
REPITA
 x,y,r ← g1(m,n,r,x,y),
         g2(m,n,r,x,y),
         g3(m,n,r,x,y);
ATÉ QUE E;
```

$g_1(m,n,r,x,y) = y$
$g_2(m,n,r,x,y) = resto(x,y)$
$g_3(m,n,r,x,y) = resto(x,y)$
$E \equiv (r = 0)$

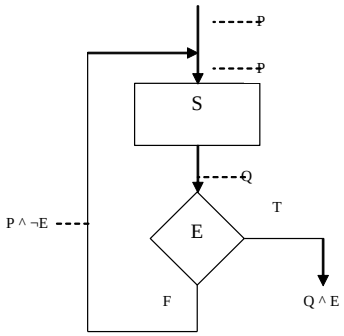
Para cada chamada de g surgirão novos valores para as componentes de x. A execução só parará quando surgirem valores de x que satisfaçam E. Em parando a execução, teremos estes valores de x como finais, juntamente com a condição de parada satisfeita (não-E para o ENQUANTO e E para o REPITA, isto é, FALSE e TRUE).

Iteração e Indução (Asserções)

Baseando-se no princípio de indução matemática, deduz-se as seguintes regras para demonstração de proposições que relacionam os valores de m e x e os comandos iterativos:

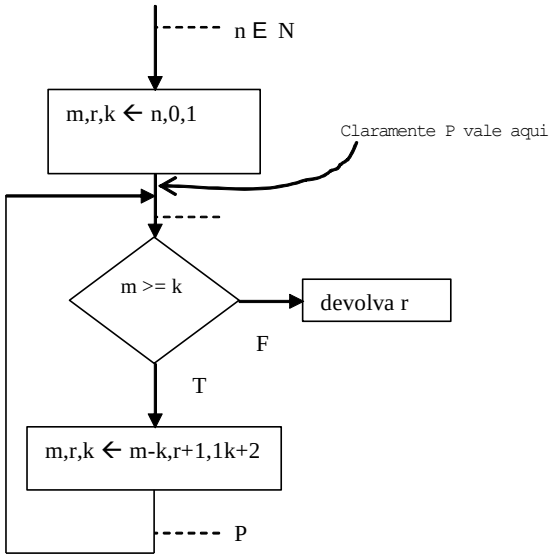


1) Para o ENQUANTO (enquanto E faça S), se:
   a. A proposição P(m,x) é verdadeira antes da execução do ENQUANTO.
   b. A veracidade de P(m,x) e de E(m,x) implica na veracidade de P(m,x) após a execução de S.
   c. Então: a execução do comando ENQUANTO não termina ou termina com P(m,x) verdadeiro e E(m,x) falso.



2) Para o REPITA (repita S até que E)
   a. P(m,x) é verdadeiro antes da execução de S e uma certa propriedade Q(m,x) vale depois da execução de S (Q pode até ser P).
   b. A veracidade de Q(m,x) e de não-E implicar a veracidade de P(m,x) após a execução de S
   c. Então: a execução do REPITA não termina ou termina com Q e E verdadeiros.

Note que a indução é feita sobre o número de vezes que o comando S é executado. A premissa (a) é a <u>base de indução</u>. A premissa (b) é o <u>passo indutivo</u>.
No exemplo abaixo, mostre que a proposição P é válida sempre que o controle passa por aquele ponto onde está P.



Podemos descobrir as seguintes igualdades:

$m_{i+1} = m_i - k_i$
$r_{i+1} = r_i + 1$
$k_{i+1} = k_i + 2$

Testamos agora se P vale após esses assinalamentos:

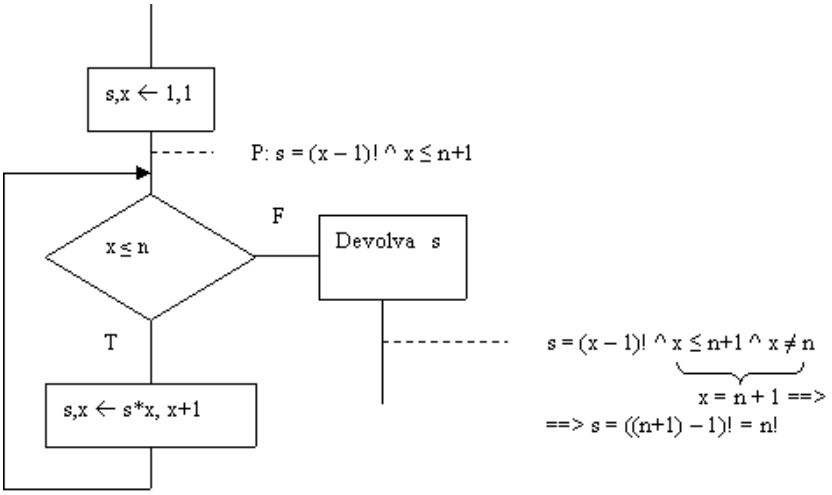
De P temos que $r_i^2 + m_i = n$. Então, após o comando $r_i^2{}_{+1}+m_{i+1} = (r_i+1)^2 + m_i - k_i = r_i^2 + 2r_i + 1 + m_i - k_i$, mas, de P: $k_i = 2r_i + 1$ ==> $r_i^2{}_{+1}+m_{i+1} = r_i^2 + 2r_i + 1 + m_i - 2r_i - 1 = r_i^2 + m_i = n$. Daqui provamos que $r^2 + m = n$ continua valendo.

De P, $k_i = 2r_i + 1$. Então $k_{i+1} = 2r_{i+1} + 1$?
Usando as três igualdades descobertas acima e substituindo:

$k_i + 2 = 2(r_i + 1) + 1$ ==> $k_i + 2 = 2r_i + 3$ ==> $k_i = 2r_i + 1$.

Assim provamos que $k = 2r + 1$ continua valendo. Daí, P vale no ponto indicado.

Exemplo 2 – Fatorial de n (n!)



Quando $x \le n$ *temos que* :

$s_{i+1} = s_i \times x_i$

Por P, temos que :

$s_i = (x_i - 1)!$

Será que : $s_{i+1} = (x_{i+1} - 1)!$?

Desenvolvendo da primeira exp *ressão* :

$s_i \times x_i = ((x_i + 1) - 1)! \Rightarrow$

$s_i \times x_i = x_i!$

Usando P, temos que :

$s_i \times x_i = x_i \times (x_i - 1)! \Rightarrow$

$s_i = (x_i - 1)!$ *ou*

$s = (x - 1)!$

O equivalente matemático :

$x_{i+1} = x_i + 1$

Por P, temos que :

$x_i = n + 1$

Será que : $x_{i+1} \le n + 1$?

Desenvolvendo da primeira exp *ressão* :

$x_i + 1 \le n + 1 \Rightarrow$

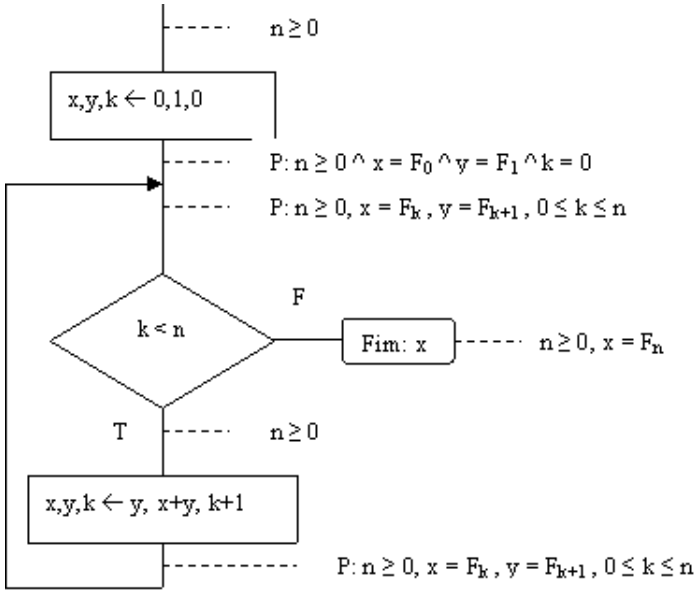
$x_i \le n$

Após a execução do comando teremos :

$x_i \le n + 1$ *ou*

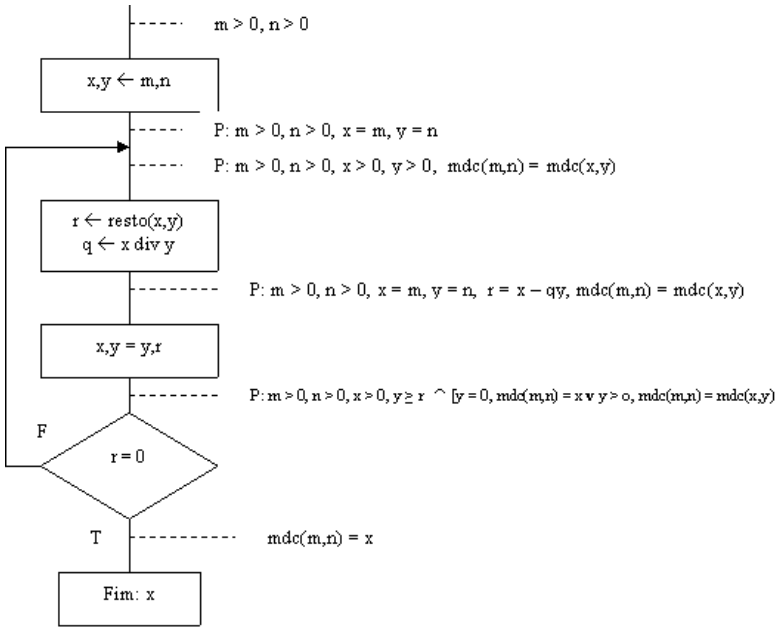
$x \le n + 1$

Exemplo 3 – Fibonacci[3] ($F_m = F_{m-2} + F_{m-1}$, $m \geq 2$)



$$x_{i+1} = y_i,\ y_{i+1} = x_i + y_i,\ k_{i+1} = k_i + 1$$
$$P : x_i = F_{ki},\ y_i = F_{ki} + 1,\ 0 \leq ki \leq n$$
$$x_{i+1} = F_{ki+1} \Rightarrow y_i = F_{ki} + 1$$

Exemplo 4 – Euclides (MDC)



[3] A sequência de Fibonacci inicia-se com 0, 1. O próximo valor é a soma dos dois últimos. Assim, para n = 7 teremos a sequência: 0, 1, 1, 2, 3, 5, 8, 13.

Recursão

Recursão é um conceito cujo uso nos permite especificar uma função em termos dela mesma, isto é, o objeto é definido em termos dele mesmo.
Consideremos a seguinte definição indutiva da função fatorial:

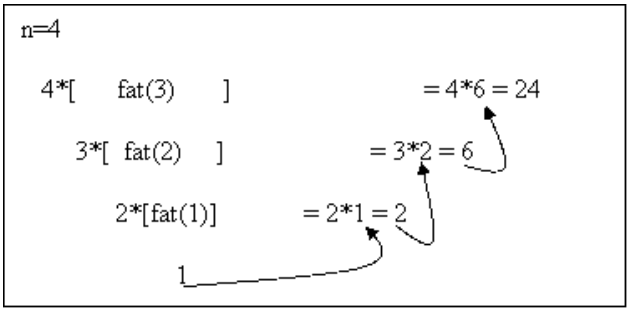
$$n! = \begin{cases} 1, \ se\ n = 0 \\ n[(n-1)!], \ se\ n > 0 \end{cases}$$

que pode ser transformada no seguinte programa (modificando um pouco):

```
procedimento fat(n):
        Se n < 2 então devolva 1
                senão devolva n*fat(n-1);
```

O programa foi modificado um pouco porque tanto 0! quanto 1! valem 1, assim consideramos apenas valores a partir de 2, para n.

Vamos ver o que acontece quando n = 4:



O que ocorre é que as primeiras multiplicações são suspensas até que o segundo fator seja um número, isto é, primeiro é feita a última multiplicação, depois a penúltima, e assim por diante, até à primeira.
Acontece como se surgisse uma cópia do programa a cada chamada, sendo que a cópia i entrega seu resultado para a cópia i – 1:

```
( procedimento fat(4)
  Se 4 < 2 então devolva 1
  senão
    devolva 4 * [ procedimento fat(3)
                  Se 3 < 2 então devolva 1
                  senão
                    devolva 3 * [procedimento fat(2)
                                 Se 2 < 2 então devolva 1
                                 senão
                                   devolva 2 * [procedimento fat(1)
                                                Se 1 < 2 então devolva 1
                                               ] = 2* 1
                                 ] = 3 * 2 * 1
                ] = 4 * 3 * 2 * 1
)
```

Para se descobrir se um processo pode ser feito recursivamente, deve-se encontrar uma relação entre o processo de um dado tamanho n e processos equivalentes menores, isto é, quando o processo original for dividido em processos menores, estes devem ter as mesmas características do processo original.

Qual é o Melhor Algoritmo?

Dado um conjunto de algoritmos para resolver um problema específico, como escolher o melhor entre eles, isto é, o mais eficiente?

- A medida não pode ser feita pelo tempo que um computador x gastaria para executar cada programa, pois para um mesmo programa A, o tempo do computador x será certamente diferente do tempo de outro computador y diferente de x.
- Não podemos também nos basear no número de instruções (escolher como melhor o menor), pois há programas grandes que são bem mais rápidos que seu equivalente menor.

O que mais influência na performance de um procedimento são os comandos iterativos, pois os demais podem ser ignorados para grandes tempos de execução.
Para aqueles programas que tem dados de entrada, claro está que o tempo de execução depende também do tamanho desta entrada.
A comparação de dois algoritmos para resolver um mesmo problema dá o grau de complexidade de um relativamente ao outro. Cada um deles, como já visto, pode ser representado por uma função matemática. O maior expoente que se tem na função que dá a complexidade do algoritmo é a *ordem de complexidade*.

A tabela abaixo mostra o tamanho da entrada (n) que pode ser processada por seis diferentes algoritmos com diferentes ordens de complexidade, dando também o tempo de execução.

Ordem	Tempo		
	1 segundo	1 hora	1 ano
$\log_2 n$	n= 10^{301}	10^{10^5}	$10^{10^{10}}$
n	1000	10^6	10^{10}
$n.\log_2 n$	140	10^5	10^9
n^2	31	1897	10^5
2^n	9	21	34
2^{2^n}	3	4	5

Então, o melhor é o que tem ordem igual a $\log_n 2$ e o pior o que tem ordem igual a 2^{2^n}, pois o primeiro roda em um segundo com uma imensa quantidade de dados, enquanto o último faz os mesmos um segundo com apenas três quantidades.

Assuma, por exemplo, que o problema é procurar numa lista de n nomes um dado nome. Um algoritmo linear (ordem n) A resolve o problema para uma lista de 1000 nomes em um segundo. Se tivéssemos um algoritmo A2 de ordem $\log_n 2$, então A2 poderia resolver com uma lista de 10^{301} nomes e no mesmo tempo de um segundo! Do mesmo modo, se um outro algoritmo A3 tem ordem $n.\log_n 2$, somente uma lista de 140 nomes poderia ser processada por ele no mesmo tempo de 1s.

Qual dos dois algoritmos para o cálculo de n! vistos anteriormente é o mais eficiente, isto é, qual é o que tem menor ordem O(n)?
Uma medida aceitável de eficiência pode ser o número total de operações primitivas (somas, subtrações, comparações) executadas e mesmo quantas vezes o laço é refeito até o final.
No caso de n! o programa iterativo fará (n+1) operações para cada n, enquanto o programa recursivo fará exatamente n operações para cada valor de n. Assim, o algoritmo recursivo é mais eficiente.

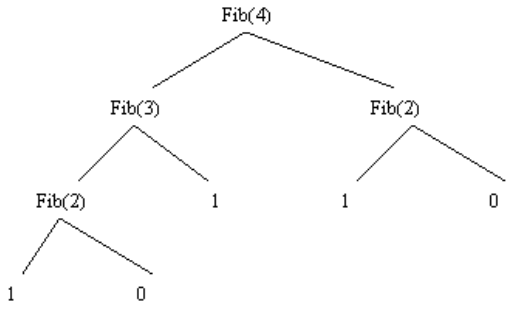
Um outro exemplo de programa recursivo é um calcula os números de Fibonacci. Criaremos também a versão iterativa e veremos que recursão nem sempre é melhor que iteração.
A sequência de Fibonacci é definida indutivamente por

$$Fn = \begin{cases} 0, \ se\ n = 0 \\ 1, \ se\ n = 1 \\ F_{n-1} + F_{n-2}, \ se\ n > 1 \end{cases}$$

que, alterando um pouco, pode ser transformada no seguinte programa recursivo:

```
procedimento Fib(n):
        Se n < 2 então devolva 1
        senão devolva Fib(n-1) + Fib(n-2)
```

Vejamos o que acontece para n = 4.



Vê-se que Fib(2) é calcula mais de uma vez.
Vamos calcular a função que dá a ordem de complexidade deste programa calculando o número de operações necessárias para o cômputo de Fn. Este número depende de n.

- Se n = 0, teremos uma operação só.
- Se n = 1, teremos apenas uma operação também.
- Agora, se n ≥ 2, teremos:
  - 1 operação no início (a comparação n <2) mais
  - 1 operação de (n-1) mais
  - 1 operação de (n-2) mais
  - 1 operação de Fib(n-1)+Fib(n-2) e mais o cálculo de Fib(n-1) e Fib(n-2), ou seja:

$$C(n) = \begin{cases} 1, \ se\ n \le 1 \\ 4 + C(n-1) + C(n-2), \ se\ n > 1 \end{cases}$$

que pode ser posto em termos de Fibonacci. Para isto temos que fazer bastante cálculos e "adivinhar" a fórmula. Veja, na expressão acima, que n ≤ 1 equivale a n < 2. Então, trabalhando em cima de C(n):

$C(0) = 1$

$C(1) = 1$

$C(2) = 4 + C(1) + C(0) = 6$

$C(3) = 4 + C(2) + C(1) = 11$

$C(4) = 4 + C(3) + C(2) = 21$

$C(5) = 36$

$C(6) = 61$

$C(7) = 101$

$C(8) = 166$

$C(9) = 271$

Com um pouquinho de trabalho vê-se que a fórmula sai fácil. Veja que, a partir de C(2), a diferença entre C(i) e C(i+1) é dada por: 5*1, 5*2, 5*3, 5*5, 5*8, 5*13 e 5*21 (para n igual a 9), ou seja, sempre 5 multiplicado pela sequência 1, 2, 3, 5, 8, 13, 21, que é a sequência normal de Fibonacci. Podemos então, concluir que:

$C(n) - C(n-1) = 5*F_{n-1}$. Daí, $C(n) = 5*F_{n-1} + C(n-1)$

Devemos, agora, colocar *C(n-1)* em termos de Fibonacci, usando a fórmula de C(n):

$\underline{C(n) = C(n-1) + 5F_{n-1}}$

$C(1) = C(0) + 5F_0 = 1$

$C(2) = C(1) + 5F_1 = 1 + 5 = 6$

$C(3) = C(2) + 5F_2 = 6 + 5 = 11$

$C(4) = C(3) + 5F_3 = 11 + 10 = 21$

$C(5) = C(4) + 5F_4 = 21 + 15 = 36$

$C(6) = C(5) + 5F_5 = 36 + 25 = 61$

$C(7) = C(6) + 5F_6 = 61 + 40 = 101$

Olhando para a coluna de C(n-1) e fazendo alguns testes notamos, por exemplo, que $C(3) = 5F_4 - 4$, de onde concluiremos que:

$C(n-1) = 5F_n - 4$ *e substituindo na fórmula de* $C(n): C(n) = 5F_n - 4 + 5F_{n-1} \Rightarrow$

$\Rightarrow C(n) = 5F_n + 5F_{n-1} - 4 \Rightarrow$ **$C(n) = 5(F_n + F_{n-1}) - 4$**

Esta fórmula não funciona para n = 0, mas podemos incluir este caso fazendo a extensão para n + 1. Da fórmula inicial $C(n) = C(n-1) + 5F_{n-1}$, teremos que:

$$C(n+1) = C(n) + 5F_n \Rightarrow C(n) = C(n+1) - 5F_n$$

Tentando derivar a fórmula novamente:

$C(0) = C(1) - 5F_0 = 1 - 0 = 1$
$C(1) = C(2) - 5F_1 = 6 - 5 = 1$
$C(2) = C(3) - 5F_2 = 11 - 5 = 6$
$C(3) = C(4) - 5F_3 = 21 - 10 = 11$
$C(4) = C(5) - 5F_4 = 36 - 15 = 21$

Podemos fazer alguns testes na coluna C(n+1) para derivar a fórmula ou usar a fórmula anterior $C(n-1) = 5F_n - 4$, para n+1, donde teremos:

$C(n+1) = 5F_{n+2} - 4$. *Substituindo na fórmula para* $C(n)$ *acima*: $C(n) = 5F_{n+2} - 4 - 5F_n$, o que resulta em **$C(n) = 5(F_{n+2} - F_n) - 4$**. Testando:

$C(0) = 5(F_2 - F_0) = 5 - 4 = 1$
$C(1) = 5(F_3 - F_1) = 5 - 4 = 1$
$C(2) = 5(F_4 - F_2) = 10 - 4 = 6$
$C(3) = 5(F_5 - F_3) = 15 - 4 = 11$
$C(4) = 5(F_6 - F_4) = 25 - 4 = 21$

A formula agora vale para $n \geq 0$. Pelo menos vale até n = 4. Será que vale para qualquer n maior que zero?
Vamos fazer a prova por indução matemática e mostrar que a fórmula vale para qualquer $n \geq 0$.

Prova por indução sobre n:

$C(n) = 5(F_{n+2} - F_n) - 4, \ n \geq 0$

Base:

n = 0, C(0) = 1
$C(0) = 5(F_2 - F_0) - 4 = 1$

n = 1, C(1) = 1
$C(1) = 5(F_3 - F_1) - 4 = 1$

Passo: n > 1

Suponhamos, como hipótese de indução, que a fórmula vale para qualquer n' < n.

$C(n) = 4 + C(n-1) + C(n-2) = \qquad (Hipótese)$

$= 4 + \underbrace{5(F_{n+1} - F_{n-1}) - 4}_{C(n-1)} + \underbrace{5(F_n - F_{n-2} - 4)}_{C(n-2)} =$

$4 + 5F_{n+1} - 5F_{n-1} - 4 + 5F_n - 5F_{n-2} - 4 =$

$5(F_{n+1} + F_n - F_{n-1} - F_{n-2}) - 4 =$

$(mas, F_{n+1} + F_n = F_{n+2} \ e - (F_{n-1} + F_{n-2}) = -F_n)$

$= 5(F_{n+2} - F_n) - 4$

Pode-se obter uma formula fechada para C(n), através da fórmula

$$F_k = (\alpha^k - \beta^k)/\sqrt{5}, \ k \geq 0$$

ou seja, uma fórmula que calcula Fibonacci, onde:

$\alpha = (1+\sqrt{5})/2 \quad e \quad \beta = (1-\sqrt{5})/2$

α se aproxima do número irracional φ (phi), também aproximado pela razão (A+B)/A, onde (A+B)/A = A/B. Esta razão é mais conhecida como Razão de Ouro. β, por sua vez, equivale ao negativo do inverso de α (- 1/ α). Em outras palavras:

$$\frac{1-\sqrt{5}}{2} = -\frac{1}{\frac{1+\sqrt{5}}{2}}$$

Demonstração:

Base:

$$k = 0, \ F_0 = 0, \ \frac{\alpha^0 - \beta^0}{\sqrt{5}} = (1-1)/\sqrt{5} = 0$$

$$k = 1, \ F_1 = 1, \ \frac{\alpha^1 - \beta^1}{\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$$

Passo k > 1:

Admite-se como hipótese de indução que $F_L = \frac{\alpha^L - \beta^L}{\sqrt{5}}$ para todo L tal que $0 \leq L \leq k$.

Ora,

$F_k = F_{k-1} + F_{k-2} = \left[(\alpha^{k-1} - \beta^{k-1})/\sqrt{5}\right] + \left[(\alpha^{k-2} - \beta^{k-2})/\sqrt{5}\right] =$

$= \left[(\alpha^{k-1} + \alpha^{k-2}) - (\beta^{k-1} + \beta^{k-2})\right]/\sqrt{5} =$

$\left[\alpha^{k-2}(\alpha+1) - \beta^{k-2}(\beta+1)\right]/\sqrt{5} = (\alpha^k - \beta^k)/\sqrt{5}$

Usamos o fato de que $\alpha + 1 = (3+\sqrt{5})/2 = \alpha^2 \ e$

$\beta + 1 = (3-\sqrt{5})/2 = \beta^2$

Uma aproximação de C(n) é $C(n) \cong 2.96 \times (1.62)^{n+1}$

Um computador que realizasse um milhão de operações básicas por segundo levaria cerca de 1.2×10^8 anos para computar F_{100}! Por isso, o programa recursivo para Fib não é eficiente.

Muito mais eficiente é o programa que usa iteração, a seguir:

```
procedimento Fib(n):
        x, y, k ← 0, 1, 0;
        Enquanto k < n
                faça x, y, k ← y, x+k, k+1;
        Devolva x;
```

A atribuição (←) não é considerada uma operação básica.

Se n = 0, faz apenas a comparação: 1.
Se n > 0, faz a comparação e mais duas operações (somas) n vezes.
Então C(n) = 3n + 1. Este programa calcularia F_{100} em 3×10^{-4} segundos (301 operações).
Assim temos uma mesma função calculada por dois algoritmos de eficiências diferentes. Portanto, nem sempre recursão é melhor que iteração, e vice-versa.
No caso da função fatorial, o programa recursivo é mais eficiente do que o programa iterativo, porém, no caso da sequência de Fibonacci, isto é, no cálculo de um valor de Fibonacci, a versão iterativa é muito mais eficiente que a versão recursiva.

Programas Equivalentes

Dois programas são equivalentes se, e somente se, calculam a mesma função parcial, isto é, quando, partindo dos mesmos dados, ou as execuções de ambos não terminam ou então ambas terminam, sendo devolvidos os mesmos resultados. Aquele que parar primeiro, nas mesmas condições, é o mais eficiente.
Os dois programas anteriores para calcular um número de Fibonacci são equivalentes, porém, o segundo é o mais eficiente.
Adotada esta definição de equivalência, pode-se provar que um programa iterativo pode sempre ser transformado num programa recursivo equivalente.
Por exemplo, para eliminar um comando iterativo **enquanto E faça S**, devemos definir o procedimento auxiliar **g**:

```
procedimento g(m1,...,mn,y1,..yp):
inicio
        x1,...,xp ← y1,..,yp;
        Se E então
                inicio
                        S; devolva g(m1,...,mn,x1,..xp);
                fim
        senão devolva x1,..xp
fim
```

Vê-se que se E for verdadeiro, o comando S é executado e volta atestar E com novos $x_1,..x_p$.
Caso E seja falso, o último valor em $x_1,..x_p$ é devolvido.
No programa original, o comando ENQUANTO E FAÇA S deve ser substituído por $x_1,..x_p \leftarrow g(m_1,...,m_n,x_1,..x_p)$.
Igualmente, para o comando iterativo REPITA S ATÉ QUE E:

```
procedimento g(m1,...,mn,y1,..yp):
inicio
        x1,...,xp ← y1,..,yp;
        S;
        Se ¬E então
                devolva g(m1,...,mn,x1,..xp);
        devolva x1,..xp
fim
```

Isto é, enquanto E for falso, os comandos 1 e 2 são executados e o teste refeito.
Repetindo-se a aplicação desta transformação a todos os comandos iterativos do programa, chega-se a uma versão sem comandos iterativos.
É claro que a eficiência do programa também mudará, para mais ou para menos.

Exemplo.

```
procedimento Fib(n): {iterativo}
inicio.
        x, y, k ← 0, 1, 0;
        Enquanto k < n faça x, y, k ← y, x+k, k+1;
        Devolva x;
fim.
```

No lugar do comando Enquanto, colocamos 'x, y, k ← g(n, x, y, k)'. E o procecimento g será:

```
procedimento g(n,y1, y2, y3):
inicio
        x, y, k ← y1, y2, y3;
        Se k < n então
                inicio
                        x, y, k ← y, x+y, k+1;
                        devolva g(n,x,y,k);
                fim
        senão devolva x, y, k;
fim
```

Estas duas versões têm, praticamente, a mesma eficiência.
A transformação contrária, de recursão para iteração, pode ser bastante difícil, conforme o tipo dos objetos manipulados.
Por exemplo, fica bastante difícil construir um programa iterativo para percorrer uma estrutura em árvore.
Em certos casos, a recursão é mais poderosa do que a iteração.

Os programas a seguir mostram duas versões do programa Fibonacci, uma iterativa e outra recursiva. Os programas estão escritos em Pascal e calculam a sequência para n = 7.

```
program fibi;
var x,y,k,A : integer;

begin
  x := 0; y := 1; k := 0;
  while k < 7 do
  begin
    A : = x;
    x := y;   y := A + x;
    k := k + 1;
  end;
  message(x);
end.
```

```
program fibr;
var x,y,k,N : integer;

procedure G(N:integer; var y1,y2,y3:integer);
var A:integer;

begin
  x := y1; y := y2; k := y3;
  if k < N then
  begin
    A : = x;
    x := y;   y := A + x;
    k := k + 1;
  end;
end;

begin
  x:=0; y:=1; k:=0; N := 7;

  G(N,x,y,k);

  message(x);
end.
```

Estes dois programas gastaram, praticamente, o mesmo tempo para devolver o resultado x = 13. O tempo foi menor que 1s.
Já a versão abaixo, também recursiva, gasta mais que 1s para devolver o resultado anterior.

```
program fibrec;
var x : integer;

function fib(n:integer) : integer;
begin
  if n < 2 then fib := n
  else fib := fib(n-1) + fib(n-2)
end

begin
  x := fib(7);
  message(x)
end.
```

Recursão e Indução

Também no caso da recursão, podemos estabelecer uma regra de demonstração baseada no princípio da indução matemática. A indução é feita sobre o número de vezes que o procedimento é aplicado recursivamente. Normalmente, a parte mais difícil em indução é achar uma hipótese conveniente. A verificação da hipótese é simples.

Exemplo 1.

```
procedimento f(n): { n > 0 }
Se n = 1 então devolva 1
senão
  Se resto(n,2) = 0 então devolva n ÷ 2 {n é par}
  senão {n é ímpar}
    devolva f(f((3n+1) ÷ 2))
```

Veja que, quando n é ímpar, (2n + 1) também é ímpar (na verdade, 2n+1 é ímpar para qualquer n ímpar ou par). Se (2n+1) é ímpar e n é ímpar e sabemos que ímpar+ímpar é sempre par, então (2n+1)+(n) = 3n + 1 é par.

A proposição a ser demonstrada é:

Se n > 1, então, ou f(n) não está definido ou então f(n) ≤ n ÷ 2 (o programa só para passando por aqui). Essa será a hipótese de indução.

Seja n > 1.
Se o cálculo de f(n) não termina, então nada há para demonstrar. Se o cálculo termina, consideremos as duas possibilidades:

a) n é par: f(n) = n ÷ 2
b) n é ímpar: f(n) = f(a), a = f((3n+1) ÷ 2)

Como o cálculo de f(n) termina, então terminam também os cálculos de a e de f(a). Aplicando a hipótese de indução duas vezes, temos que a ≤ (3n+1) ÷ 4, ou seja,

[(3n+1)/2]/2 e f(a) ≤ a ÷ 2 ≤ (3n+1) ÷ 8

Para todo n > 1 tem-se que (3n+1) ÷ 8 ≤ n ÷ 2, portanto f(n) ≤ n ÷ 2.

Exemplo 2.

```
procedimento g(n): { n > 0}
Se n > 100 então devolva n-10
senão devolva g(g(n+11))
```

Demonstra-se que a função g calculada pelo procedimento é igual à função h, definida por:

$$h(n) = \begin{cases} n - 10, \; se \; n > 100 \\ 91, \; se \; n \leq 100 \end{cases}$$

Considerando os três casos possíveis:

a) n > 100. Neste caso, g(n) = n-10 = h(n)
b) $90 \leq n \leq 100$. Aqui, g(n) = g(g(n+11)), onde as duas ocorrências de g correspondem a chamadas recursivas do procedimento. Usando, então, a hipótese de indução ("h(n) = g(n)"):

   g(n) = g(g(n+11) = g(h(n+11)) = h(h(n+11))

   Ora, n+11 será, no mínimo, o número 101, ou seja, n+11 > 100 e, portanto:

   $h(\underbrace{h(n+11)}_{=n+11-10}) = h(\underbrace{n+1}_{\leq 101}) = 91 = h(n)$, por definição de h. Daí, g(n) = h(n). Hipótese verificada!
c) n < 90.
   Aplicando a hipótese duas vezes (pois teremos "g(g())"):
   $g(n) = g(g(n+11)) = g(h(n+11) = \{n+1 \leq 100 \Rightarrow n \leq 89\}$
   $= h(\underbrace{h(n+11)}_{91,\ por\ def\ de\ h}) = h(91) = 91 = h(n)$
   Daí, g(n) = h(n), como queríamos demonstrar.

O que foi demonstrado é que, se o procedimento g parar, então g(n) = h(n). Agora, g(n) para sempre?
Com pouco trabalho podemos descobrir que o número de chamadas C(n) é igual a 2(101-n) para $n \leq 101$. Claramente, para qualquer n > 101, não há nenhuma chamada recursiva, isto é

$$C(n) = \begin{cases} 0, \text{ se } n > 101 \\ 2(101 - n), \text{ se } n \leq 101 \end{cases}$$

Daí, o procedimento g sempre para.

Exemplo 3.

Seja M o menor conjunto que contém o elemento 1 e está fechado sob as operações
g(x) = 2x + 1 e f(x) = 3x + 1,
isto é, se o elemento **a** está em M, então 2a + 1 e 3a + 1 também estão em M.
Se n está em M, n é da forma 2y + 1 ou 3x + 1, onde x e y estão em M e x < y ou x = y ou x > y. Também $y = \frac{n-1}{2}$ *e* $x = \frac{n-1}{3}$. As duas expressões dão o mesmo resultado quando $\frac{x}{y} = \frac{2}{3}$.
Deseja-se escrever um programa que constrói a sequência dos n menores elementos de M, com $n \geq 1$, em ordem crescente.
Por exemplo, para n = 15: <1,3,4,7,9,10,13,15,19,21,22,27,28,31,39>.

Suponhamos que já conseguimos construir a sequência parcial <S_1, S_2, ...,S_m>, com m maior ou igual a 1 e menor ou igual a n. É fácil verificar que existem **p** e **q**, tais que: $1 \le p,q \le m$:

$$g(S_p) > S_m,\ g(S_i) \le S_m,\ \forall i \,|\, 1 \le i < p$$

$$f(S_q) > S_m,\ f(S_i) \le S_m,\ \forall i \,|\, 1 \le i < q$$

Em outras palavras, o menor entre $g(S_p)$ e $f(S_q)$ será o valor de Sm+1 e os novos valores de p e q serão p' e q':

$$p',q' = \begin{cases} p, q+1, \ se\ f(S_q) < g(S_p) \\ p+1, q, \ se\ f(S_q) > g(S_p) \\ p+1, q+1, \ se\ f(S_q) = g(S_p) \end{cases}$$

- p e q tem que ser os mínimos possíveis, e, nesse caso, q não pode ser menor que p.
- p é o primeiro elemento cujo dobro mais um não está na lista.
- m=p=q=1 e a sequência <1> constituem uma base conveniente para fazer a indução.
- O procedimento abaixo constrói a sequência para qualquer n:

```
procedimento h(n):
inicio.
  p,q,m,t ←1,1,1,<1>;
  Enquanto m < n faça
  inicio.
    a,b ← 2*t[p]+1,3*t[q]+1;
    m ← m+1;
    Se a < b então
        p,t ← p+1, t#a; {o símbolo # insere a na lista t}
    senão
        Se a > b então
            q, t ← q+1, t#b;
        senão
             p,q,t ← p+1, q+1, t#a;
  fim;
  devolva t;
fim.
```

O procedimento abaixo testa se $m \in M$.

```
procedimento pertence(m):
Se m ≤ 1 então devolva 1
senão devolva
          ((resto(m,2)=1) AND pertence(m÷2)) OR ((resto(m,3)=1) AND pertence(m÷3));
```

TEORIA DOS CONJUNTOS

Igualdade

$A = B \Leftrightarrow x \in A \wedge x \in B,\ para\ \forall x$

Lê-se: A é igual a B se, e somente se, x pertence a A e x pertence a B, para qualquer valor x.

Conjunto Vazio

É um conjunto que não contém elementos.

$\varnothing = \{\}$

Subconjunto

A é subconjunto de B se $x \in A \Rightarrow x \in B$. Em outras palavras, A está **contido** em B ou B **contém** A: $A \subset B\ ou\ B \supset A$.

Uma outra maneira é dizer que B é superconjunto de A.

Comparabilidade

A e B são comparáveis se $A \subset B\ ou\ B \supset A$

A e B são incomparáveis se $A \not\subset B\ e\ B \not\subset A$, isto é existe

$x \in A \mid x \notin B\ e\ \exists y \in B \mid y \notin A$

Conjunto de Conjuntos

Os elementos do conjunto são conjuntos. O conjunto principal é chamado também de *classe de conjuntos*, ou *família de conjuntos*.

Conjunto Potência

A família de todos os subconjuntos de qualquer conjunto S é chamada de *conjunto potência de S* e é designada por 2^S.

Se S é finito e tem n elementos, então 2^S tem 2^n elementos.

Conjunto Universal

Seja um conjunto A qualquer. O conjunto universal U é dado pela união (ver) do conjunto A com tudo que não pertence a A:

$U = A \cup \overline{A}$

Conjuntos Disjuntos

A e B são disjuntos se não tem pelo menos um elemento em comum, isto é, a intersecção (ver) de A com B resulta no conjunto vazio.

União

A união de um conjunto A com um conjunto B resulta num conjunto que contém elementos de A e elementos de B:

$$A \cup B = \{x \mid x \in A \vee x \in B$$

Lê-se: A unido B é um conjunto com elementos x tal que x pertence a A ou x pertence a B (os conjuntos originais, antes da união).

Propriedades:

$$A \cup B = B \cup A$$

$$A \subset (A \cup B) \;\; e \;\; B \subset (A \cup B)$$

Intersecção

A intersecção de um conjunto A com um conjunto B resulta num conjunto que contém elementos comuns a A e B, simultaneamente. Se A e B não tiverem elementos comuns, o conjunto resultante será o conjunto vazio.

$$A \cap B = \{x \mid x \in A \wedge x \in B$$

Propriedades:

$$A \cap B = B \cap A$$

$$(A \cap B) \subset A \;\; e \;\; (A \cap B) \subset B$$

$$Se\, A \; e \; B \; são\, disjuntos, então\, A \cap B = \varnothing$$

Diferença

$$A - B = \{x \mid x \in A \wedge x \notin B$$

Propriedades:

$$(A - B) \subset A$$

Complementação

O complemento de um conjunto $A \subset U$ é:

$$\overline{A} = (U - A) = \{x \mid x \in U \wedge x \notin A\} = \{x \mid x \notin A\}$$

Propriedades:

$A \cup \overline{A} = U$

$A \cap \overline{A} = \varnothing$

$\overline{U} = \varnothing \; e \; \overline{\varnothing} = U$

$\overline{\overline{A}} = A$

$A - B = A \cap \overline{B}$

Conjuntos Numéricos

Os Reais ℝ

- Podem ser representados por cada ponto de uma linha reta:

$-\pi \;\; -3 \;\; -2 \;\; -1 \;\; 0 \;\; \frac{1}{2} \;\; 1 \;\; \sqrt{2} \;\; 2 \; e \;\; 3$

---|-----|-----|----|---|---|--|---|---|-|---|--------

Os Inteiros ℤ

- São os reais ...,-3, -2, -1, 0, 1, 2, 3, ...

Os Racionais ℚ

- São os reais que podem ser representados como a relação de dois inteiros:

$$Q = \{x \mid x = \frac{p}{q}, \; com \; p, q \in Z$$

Os Naturais ℕ

- São os inteiros positivos mais o zero.

Os Irracionais I = ℝ - ℚ

Os Complexos ℂ

- O conjunto dos números complexos incluem número que não pertencem a ℝ. É definido assim:

$$C = \{a + bi \mid a, b \in R \wedge i^2 = -1$$

- Propriedades

- $\mathbb{N}$ é fechado nas operações de adição e multiplicação (qualquer operação de adição e multiplicação entre dois elementos de $\mathbb{N}$ gera um resultado que pertence a $\mathbb{N}$).
- $\mathbb{Q}$ é fechado nas operações de adição, subtração, multiplicação e divisão (exceto por zero).
- $\mathbb{Z}$ é fechado nas operações de adição, subtração e multiplicação.
- $\mathbb{R}$ é fechado nas operações de adição, subtração, multiplicação e divisão (exceto por zero).
- $N \subset Z \subset Q \cup R \subset C$

Conjunto Limitado e Ilimitado

O conjunto A é limitado se existir um número M positivo tal que $|x| \leq M$ para qualquer x pertencente a A.
|x| significa o valor absoluto (sem sinais) de x.

Um conjunto é ilimitado se não for limitado.
Se um conjunto A é finito, então A é limitado. Se A é infinito, pode ser limitado ou ilimitado. Por exemplo, o subconjunto {2,...,3} do conjunto $\mathbb{Q}$ é infinito e limitado.

Par Ordenado e Produto Cartesiano

O produto cartesiano de dois conjuntos A e B é dado por
$A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$
(a,b) é chamado de par ordenado.
Se $(a,b) = (a,c)$, então b=c.

Um par ordenado (a,b) pode ser definido rigorosamente por $(a,b) = \{a,\{a,b\}\}$.

Paradoxo de Russel

Um conjunto A não é um elemento de si mesmo ou então é, isto é:

$$A \in A \vee A \notin A$$

Seja R o conjunto de todos os conjuntos que não são membros de si mesmo.
Onde está R então?
- Se R pertence a R, então R não pode pertencer a R (porque R só pode conter conjuntos que não sejam membros de si mesmos).
- Se R não pertence a R, então R tem que pertencer a R, pois R não será um membro de si mesmo e pode ser colocado em R.

Daí o paradoxo: Não pode ser que R pertença a R e nem que R não pertença a R.

Assume-se que nenhum conjunto seja elemento de si mesmo. Na verdade, essa é uma condição essencial para a criação do paradoxo. Na verdade, também, qualquer conjunto é membro de si mesmo.

Funções

Função, transformação, máquina, mapeamento, são conceitos equivalentes, mas o nome mais usado é função.

Uma função é uma transformação, ou mapeamento, de um elemento de um conjunto A num único elemento do conjunto B:

- $f : A \to B$

  Domínio de *f (Domf)*: A
  Contradomínio de *f (Cdomf)*: B
  Imagem de *f (Imgf)*: C $\subset$ B

  - Para todo $a \in$ A, $f(a) \in$ B.
  - Se *Domf* = *Cdomf*, isto é, são o mesmo conjunto (*f*: A → A), então *f* é chamado **operador** ou **transformação** em A.
  - Sejam *f* e *g* duas funções, então, *f* = *g* se *Domf* = *Domg* e *f*(*a*) = *g*(*a*) para qualquer a pertencente ao domínio comum.
  - $f : A \to B$ é **injetora** (biunívoca, one-to-one) se, para todo $a \neq a' \Rightarrow f(a) \neq f(a')$, ou seja, cada elemento de B tem um, e só um, correspondente em A.
  - $f : A \to B$ é **sobrejetora** (sobre, onto) se *Imgf* = *Cdomf*, ou seja, para cada elemento $b \in$ B existe pelo menos um a $\in$ A tal que *f*(*a*) = *b*. A função *f* mapeia todo o conjunto B.
  - $f : A \to A$ é a transformação **identidade** em A, ou seja, para todo x $\in$ A, *f*(x) = x.
  - $f : A \to B$ é **função constante** se para todo x $\in$ A implicar que f(x) = b, isto é, o conjunto imagem C $\subset$ B é C = {b}.

Função Composta

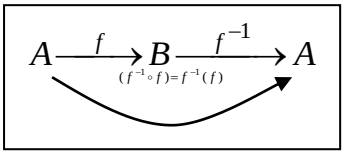
Sejam $f : A \to B$ e $g : B \to C$, $(g \circ f) : A \to C$ é a função composta de A em C e $(g \circ f)(a) \equiv g(f(a))$, *Dom*(*g*o*f*) = *Imgf*, *Img*(*g*o*f*) = {C}.

Função Inversa

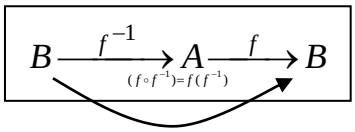
Seja $f : A \to B$. Sua inversa, denotada por f^{-1}, é $f^{-1}(b) = \{x \mid x \in A \wedge f(x) = b\}$, isto é: $f^{-1} : B \to A$

Propriedades:

- A imagem de f^{-1} são aqueles x $\in$ A que tem f(x) $\in$ B.
- f só terá uma inversa se f é injetora.
- Se f é injetora e sobrejetora, ou seja, bijetora, a inversa f^{-1} é imediata.
- Se f é injetora, f^{-1} é sobrejetora e injetora.
- Se f é sobrejetora apenas, podemos "cavar" uma f^{-1} injetora restringindo a imagem de f. Note que isso não prejudica a sobrejeção de f.
- Consideremos que a função $f : A \to B$ tem uma função inversa $f^{-1} : B \to A$. Pelo diagrama:



(f^{-1}of) representa A em A, é uma função de A em A e



é uma função de B em B.

- Se $f : A \to B$ é bijetora, então (f^{-1} o f): A → A é a função identidade em A, e (f o f^{-1}): B → B é a função identidade em B.

Gráfico f* de uma função f.

$f^* = \{(a,b) \mid a \in A \wedge b = f(a)\}$ e assim : $f^* \subset A \times B$

Funções Conjunto

Seja $f : A \to B$ e T $\subset$ A, então $f(T) = \{x \mid f(a) = x,\ a \in T,\ x \in B\}$ e $f(T) \subset B$

Função Característica

Seja A um subconjunto qualquer de U. Desse modo, a função característica real F_A:U → {0,1}, definida por

$$F_A(x) = \begin{cases} 1, \ se \ x \in A \\ 0, \ se \ x \notin A \end{cases}$$

é chamada **função característica do conjunto A**.

Nota:

A função é característica do conjunto A porque ela está ligada ao conjunto A do seguinte modo: Para todo x Є U, ela vale 1 (devolve 1) para todo e somente para aqueles x que pertencem a A.

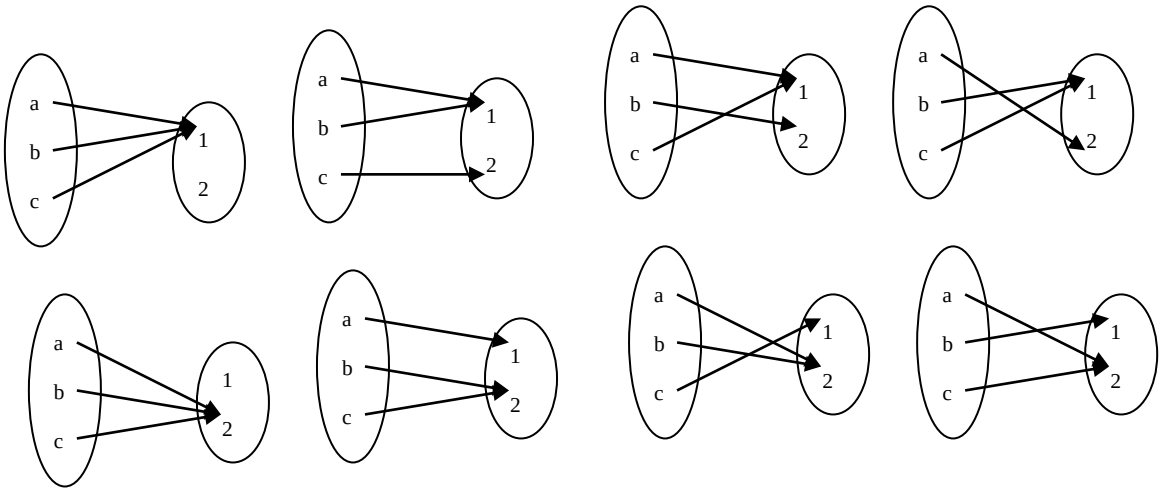
Além disso, qualquer função f : U → {0,1}, define um subconjunto $A_f = \{x \mid x \in U \ e \ f(x) = 1\}$ de U e que a função característica F_{A_f} do subconjunto A_f é a função original f.
Assim, existe uma injeção de todos os subconjuntos de U, denotado por 2^U, e o subconjunto de todas as funções de U em {0,1}. Isto é, cada subconjunto de U tem uma função que lhe é característica, e a função F_A aplicada ao conjunto B, quando B ≠ A, produzirá um resultado falso para um certo $x_0 \in B$, ou seja, $F_A(x_0) = 0$, implicando que $x_0 \notin B$.

Família de Funções

Sejam A e B dois conjuntos quaisquer. Então B^A designa o conjunto de todas as funções de A em B, ou B^A = {f : A → B}.

Exemplo:

Sejam A = {a,b,c} e B = {1,2}.
Podemos obter as seguintes funções (que são todas as possíveis) de A em B:



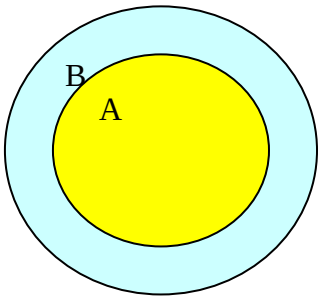
Daí B^A = { {(a,1),(b,1),(c,1)}, {(a,1),(b,1),(c,2)}, {(a,1),(b,2),(c,1)},
(a,1),(b,2),(c,2)}, {(a,2),(b,1),(c,1)}, {(a,2),(b,1),(c,2)},
(a,2),(b,2),(c,1)}, {(a,2),(b,2),(c,2)}
}

Pode-se ver que se B tem **m** elementos e A tem **n** elementos, B^A é composto exatamente de m^n funções.
Os valores *m* e *n* não chamados de *cardinalidade* de B e de A, respectivamente.
No exemplo acima, m=2 e n=3, daí $m^n = 2^3 = 8$, que é o número de funções de A em B.

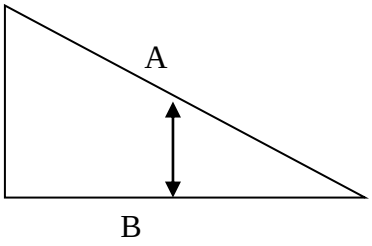
Se A = U e B = {0,1}, temos que B^A é o conjunto de todas as funções de U em {0,1}, ou seja, o conjunto das funções características de cada subconjunto de U.
O conjunto dos subconjuntos de U é 2^U e corresponde exatamente ao número de funções de U para {0,1}.

Conjuntos Equivalentes

Um conjunto A é equivalente a um conjunto B, designado por A ~ B se existir uma função f : A → B, bijetora.
Se A e B são finitos, A ~ B se, e somente se, A e B tem o mesmo número de elementos.



O conjunto A é equivalente ao conjunto B, pois tem (?!) a mesma quantidade de elementos (pontos).
E se abrirmos os dois círculos?



O conjunto A é equivalente ao conjunto B e B é um subconjunto próprio de A!

Claro está que um subconjunto próprio de um conjunto finito não é equivalente ao conjunto, mas, se o conjunto é infinito isso pode ser.

Um conjunto A é **infinito** sempre que for equivalente a um subconjunto próprio de si mesmo, caso contrário, o conjunto será finito.

Sabemos que o conjunto dos números pares é um subconjunto próprio de N e é equivalente a N, pois f(n) = 2n, com n pertencente a N, é uma função bijetora. Uma outra prova intuitiva é que você pode rotular todos os números naturais com valores pares. Daí, existem tantos números pares quantos são os números naturais.
Isso é muito esquisito. O que pode estar errado aqui? Nada. Se você pegar um número infinito de tijolos vermelhos e um número infinito de tijolos verdes, poderá por cada tijolo verde sobre cada vermelho ou vice-versa e, ainda assim, a quantidade de pilhas de dois tijolos será igual à quantidade de tijolos vermelhos e igual à quantidade de tijolos verdes. Ao mesmo tempo, a quantidade de pilhas vai ser igual à soma da quantidade de tijolos vermelhos com a quantidade de tijolos verdes.

Cardinalidade de um Conjunto

Seja A um conjunto qualquer. Todo conjunto equivalente a A tem o mesmo tamanho de A e, por isso, pertence à família de A. Todo conjunto pertence a uma dada família e todos os conjuntos desta família são equivalentes, e essas famílias são disjuntas, claro.
Cada uma destas famílias é numerada de acordo com o número de elementos de um de seus conjuntos. Assim, todos os conjuntos que tem um, e apenas um, elemento, pertencem à família número 1, e assim por diante.
Esse número que designa uma dada família de conjuntos equivalentes é chamado de **Número Cardinal**, e qualquer conjunto daquela família tem a mesma *cardinalidade*.

- A cardinalidade de N, #(N), é designada por #(N) = $\aleph_0$.
- A cardinalidade de R, #(R) = $\mathfrak{c}$.
- As cardinalidades de Z e Q são: #(Z) = #(Q) = #(N) = $\aleph_0$.
- A cardinalidade de [0,1] é #([0,1]) = #(R) = $\mathfrak{c}$
- Seja α e β números cardinais e sejam A e B conjuntos disjuntos tais que α = #(A) e β = #(B), então

$$\alpha + \beta = \#(A \cup B)$$
$$\alpha \bullet \beta = \#(A \times B)$$
$$\beta^{\alpha} = \#(B^{A})$$

Sejam **d** e **e** números cardinais não nulos. Sejam D e E conjuntos com #(D) = **d** e #(E) = **e**.
Definimos d^e como o número cardinal do conjunto de todas as funções de E para D, denotado por D^E.

Denote P(D) como o conjunto de todos os subconjuntos de D. 2^d é o cardinal de P(D). É, também, o cardinal do conjunto de funções de D para um conjunto de dois elementos.

Façamos T = {0,1}. Dado um subconjunto A de D, escrevemos para ele sua função característica, definida por $f_A(x) = 1 \; se \; x \in A \;\; e \;\; f_A(x) = 0 \; se \; x \notin A$.
O que associa A a f_A é uma função de P(D) em T^D, que é bijetora. Então P(D) e T^D tem o mesmo número cardinal, isto é, são equivalentes:

#(T^D) = 2^d por definição e #(P(D)) = 2^d.

Para qualquer conjunto *S*, #(*S*) < #(2^S). Este é o Teorema de Cantor.

Prova 1:

A função g : *S* → 2^S, que envia cada elemento s de *S* para o conjunto consistindo apenas de s, isto é, que é definida por g(s) = {s}, é injetora.
Assim #(*S*) ≤ #(2^S). Se demonstrarmos que *S* não é equivalente a 2^S, o teorema se seguirá.
Suponhamos o contrário, que exista uma função f : *S* → 2^S que é injetora e sobrejetora. Seja s Є *S* denominado de "mau" elemento. s não é um membro do conjunto que é sua imagem, isto é, $s \notin f(s)$, e seja *B* o conjunto dos "maus" elementos, ou seja: $B = \{x \mid x \in S, \; x \notin f(x)\}$.
Claramente, *B* é um subconjunto de *S*, isto é, B Є 2^S. Assim, como f : S → 2^S é definida sobre 2^S, então existe um elemento b Є *S* com a propriedade f(b) = *B*.
Agora, b é "mau" ou não?
Se *b* Є *B*, então $b \notin f(b) = B$, por definição de *B*. Também, se $b \notin B$, então $b \in f(b) = B$.
Temos duas contradições. Assim, a suposição original de que *S* ~ 2^S conduziu-nos a uma contradição. Daí, a suposição é falsa, e #(*S*) < #(2^S).

Prova 2:

Suponha, por absurdo, que *h* : *S* → 2^S é uma sobrejeção.
Seja $\int = \{s \in S \mid s \notin \int_s\}$, onde $\int_s \subseteq S$, $\int_s = h(s)$. Claramente $\int \subseteq S \therefore \int \in 2^s$.
Também $\int \neq \int_s$ para todo *s* Є *S*, pois $s \in \int \; sse \; s \notin \int_s$. Assim, h não pode ser uma sobrejeção.

Para quaisquer conjuntos A e B, pelo menos um dos seguintes precisa ser verdadeiro:

1. A ~ B, isto é #(A) = #(B).
2. A ~ D C B ou C C A ~ B, isto é, #(A) < #(B) **ou** #(B) < #(A).
3. A ~ D C B e B ~ C C A, isto é, #(A) ≤ #(B) **e** #(B) ≤ #(A).
4. A não é equivalente a um subconjunto de B e B não é equivalente a um subconjunto de A, isto é, #(A) ≥ #(B), #(A) ≠ #(B) e #(A) ≤ #(B).

No caso 3, A ~ B e o caso 4 é impossível.

Vimos que #(N) = $\aleph_0$ e que #(R) = $\mathfrak{c}$.

Temos que $\aleph_0 < \mathfrak{c}$. Para N, temos que $\aleph_0 < 2^{\aleph_0}$.

Teorema: $2^{\aleph_0} = \mathfrak{c}$.

Prova:

Seja a função f : R → $2^{\aleph_0}$, definida por $f(a) = \{x \mid x \in Q \wedge x < a\}$, isto é, f representa cada número real a no conjunto dos números racionais menores que a. Sejam $a, b \in R,\ a \neq b$, e digamos que $a < b$. Por uma propriedade dos números reais, existe um número racional r tal que a< r < b. Então $r \in f(b)\ e\ r \notin f(a)$, assim, $f(b) \neq f(a)$ e, desse modo, f é injetora. Assim |R| ≤ $|2^Q|$, ou, $\mathfrak{c} \leq 2^{\aleph_0}$.

Seja, agora, C(N) a família das funções características f : N → {0,1}, e C(N) ~ 2^N. Considere a função F : C(N) → [0,1], definida por F(f) = 0.f(1)f(2)f(3)... como um número decimal infinito composto de zeros e uns.

Se $f, g \in C(N)$ e $f \neq g$, então F(f) ≠ F(g). Então F é injetora. Desse modo, 2^N ~ C(N) ≤ [0,1] e então $2^{\aleph_0} \leq \mathfrak{c}$. Consequentemente, $\mathfrak{c} = 2^{\aleph_0}$.

Terorema: [0,1] ~ (0,1)

Prova:

$$f(x) = \begin{cases} 1/2, & x = 0 \\ 1/(n+2), & x = 1/n,\ n \in N \\ x, & x \neq 0, 1/n,\ n \in N \end{cases}$$ é bijetora.

Teorema: $\forall A, B \Rightarrow (A \times B) \sim (B \times A)$

Prova:

f : (A×B) → (B×A), definida por f(a,b)) = (b,a) é bijetora.

Conjuntos equivalentes tem a mesma cardinalidade e vice-versa (conjuntos de mesma cardinalidade são equivalentes).

Teorema: $\aleph_0 \mathfrak{c} = \mathfrak{c}$.

Prova:

Seja Z e A = [0,1). Além disso, seja f : Z × A → R, definida por f(i, a) = i + a. Em outras palavras, f({i} × [0,1)) é representada sobre [i, i+1). Assim, f é bijetora.

Assim, (Z × A) ~ R, daí, $\aleph_0 \mathfrak{c} = \mathfrak{c}$.

Seja A um alfabeto contendo m símbolos, onde $m \geq 1$.
Sempre que as palavras sobre A sejam interpretadas como números, devemos assumir que aos símbolos de A deu-se uma ordem fixa $a_1, a_2, a_3, ..., a_m$.
Além do mais, cada símbolo a_i de A será considerado como um dígito representando o inteiro i. Assim, podemos escrever: A = {1, 2, 3, ..., m}.
A função h : A* → N, tal que h(x) denota a interpretação numérica de uma palavra x pertencente a A* é definida como segue:

a) $h(\lambda) = 0$
b) Se $x = i_k i_{k-1} i_{k-2} ... i_3 i_2 i_1 i_0$, onde cada $i_j \in A$, então $h(x) = i_0 m^0 + i_1 m^1 + i_2 m^2 + i_3 m^3 + ... + i_k m^k = \sum_{j=0}^{k} i_j m^j$

Esta representação de números é chamada de Notação m-Ádica. Ela tem a vantagem de ter uma injeção entre A* e N, de tal modo que, a toda palavra sobre A*, corresponde um único inteiro em N, e cada inteiro em N é h(x) para exatamente uma única palavra x sobre A*.

Conjuntos Convexos

Um subconjunto X do espaço euclidiano n-dimensional R^n é chamado de **conjunto convexo** se, para quaisquer elementos x_1, x_2 em X e qualquer número α, tal que $0 \leq \alpha \leq 1$, o elemento $\alpha x_1 + (1-\alpha) x_2$ de R^n também pertence a X.
Um ponto x de um conjunto convexo X é chamado um **ponto extremo** de X se x não puder ser expresso com $(x_1 + x_2)/2$, em termos de um par de pontos distintos x_1, x_2 de X.

ENUMERABILIDADE

Conjuntos

- Se um conjunto D é equivalente a N, então D é *enumerável*, isto é, cada elemento de D pode receber um "rótulo" n de N.

- Diz-se que um conjunto é contável se ele é finito ou enumerável. Um conjunto é não-enumerável se ele é infinito e não equivalente a N.

- Um conjunto S é contável se, e somente se, S é vazio ou se existir uma sequência s0, s1, s2, s3, ..., que contem todos, e somente, os elementos de S.

- Cada conjunto infinito contém um subconjunto que é enumerável.

Prova:

Seja A o conjunto infinito e D seu subconjunto.
Selecione de A um elemento aleatório a_1. Desde que A é infinito, ele não será exaurido pela extração de a_1, e poderemos extrair um elemento a_2 do conjunto $\{A - \{ a_1\}\}$. Pela mesma razão, o conjunto $\{A - \{ a_1, a_2\}\}$não é vazio, e podemos extrair a_3 dele. Desde que A é infinito, podemos continuar este processo indefinidamente, obtendo, como resultado, uma sequência de elementos $a_1, a_2, a_3,..., a_n$, que forma o conjunto D requerido.

- Um subconjunto de um conjunto enumerável é finito ou enumerável.

Prova:

Seja $A = \{ a_1, a_2,...\}$ um conjunto enumerável qualquer e seja B um subconjunto de A. Se $B = Ø$, então B é finito. Se $B \neq Ø$, seja, então, a_{n_1}, o primeiro elemento na sequência tal que $a_{n_1} \in B$. Seja a_{n_2} o primeiro elemento que segue a_{n_1} na sequência tal que $a_{n_2} \in B$, etc.
Assim, $B = \{ a_{n_1}, a_{n_2},...\}$.
Se o conjunto dos inteiros $\{ n_1, n_2,...\}$ é limitado, B é finito. De outro modo, B é enumerável.

- Se A é infinito e enumerável e B é finito e $A \cap B = Ø$, então $A \cup B$ é enumerável.

- A união de uma família enumerável de conjuntos disjuntos enumeráveis é um conjunto enumerável.

Prova:

Seja A_k, k = 1, 2, 3, ... essa família. Escrevemos os conjuntos dessa família assim:

$$A_1 = \{a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, ...\}$$
$$A_2 = \{a_1^{(2)}, a_2^{(2)}, a_3^{(2)}, ...\}$$
$$A_3 = \{a_1^{(3)}, a_2^{(3)}, a_3^{(3)}, ...\}$$
$$...$$

Se escrevermos primeiro o elemento $a_1^{(1)}$, então os dois elementos $a_2^{(1)}$ e $a_1^{(2)}$, em que a soma dos índices superiores e inferiores seja igual a 3, em cada caso, depois os elementos para os quais a soma daqueles índices resulte em 4, depois em 5, e assim por diante, a soma $S = \bigcup_{k=1}^{\infty} A_k$ é arranjada numa sequência na seguinte forma:

$S = \{a_1^{(1)}, a_2^{(1)}, a_1^{(2)}, a_3^{(1)}, a_2^{(2)}, a_1^{(3)}, a_4^{(1)}, ...\}$, resulta daí que a enumerabilidade de S é óbvia.

- Se adicionarmos um conjunto A finito ou enumerável a um conjunto infinito M, a cardinalidade de M não se altera, isto é, M + A ~ M.

- $\mathfrak{a}$ – n = $\mathfrak{a}$, $\mathfrak{a}$ + n = $\mathfrak{a}$, n$\mathfrak{a}$ = $\mathfrak{a}$, $\mathfrak{a}$. $\mathfrak{a}$ = $\mathfrak{a}$

- O conjunto P dos polinômios de coeficientes inteiros é enumerável.

  Prova:

  Para cada par de números naturais (n, m), seja P(n,m) o conjunto dos polinômios em $p(x) = a_0 + a_1 x + ... + a_m x^m$ de grau m, nos quais $|a_0|+|a_1|+...+|a_m| = n$.
  Observe que Pn,m é finito. Desse modo, $P = U_{i \in N \times N} P_i$ é mensurável, pois é uma família mensurável de conjuntos mensuráveis. Porém, P não é finito. Daí, P é enumerável.

- Desde que cada polinômio tem somente um número finito de raízes (chamadas de números algébricos), segue que o conjunto de números algébricos é enumerável.
  Prova:

  $E = \{p_1(x) = 0,\ p_2(x) = 0, ...\}$ é enumerável pelo teorema anterior. Defina $A_i = \{x \mid x \text{ é solução de } p_i(x) = 0\}$

  Como vimos, um conjunto é enumerável se seus elementos podem ser enumerados, isto é, arranjados numa lista contendo um primeiro elemento, um segundo elemento, etc., de tal modo que cada membro do conjunto apareça mais cedo ou mais tarde na lista. E cada elemento da lista pode ser associado aos naturais 1, 2, 3, ..., para o primeiro elemento, segundo, terceiro, ..., respectivamente.

É perfeitamente possível que um dado elemento apareça mais que uma vez na listagem de um conjunto. O que é requerido é que cada elemento apareça pelo menos uma vez na lista. Não importa se a lista é redundante. Tudo o que se requer é que ela seja completa. Na verdade, uma lista redundante pode ser tornada não-redundante eliminando-se aqueles elementos, ou melhor, removendo os n-1 elementos iguais àquele elemento que se repete n vezes. Matematicamente falando, uma lista infinita determina uma função *f* que tem como argumentos inteiros positivos e os membros do conjunto (a lista) como valores. Assim, a lista 2, 4, 6, 8, ... de naturais pares determina a função *f* para a qual temos que $f(1) = 2$, $f(2) = 4$, $f(3) = 6$, $f(4) = 8$, ... e, reciprocamente, a função *f* determina a lista (dependendo da notação da função).

Podemos então falar de conjuntos enumeráveis por funções, bem como por listas. Se uma função enumera um conjunto não vazio, também infinitas outras funções enumeram o mesmo conjunto.

Assim, N pode ser enumerado por $f(n) = n \;\; ou \;\; g(n) = \begin{cases} n+1, \; se \; n \; é \; ímpar \\ n-1, \; se \; n \; é \; par \end{cases}$, e as listas correspondentes são:

l_f = 1, 2, 3, 4, ...
l_g = 2, 1, 4, 3, 6, 5, ...

Vimos que uma lista pode ser redundante ao enumerar um conjunto. Dizemos que também é perfeitamente possível que uma lista contenha "gaps" nela, pois podemos "passar" por eles. Assim, uma enumeração mais "fraca" de N é dada pela seguinte lista com "buracos":

1, -, 2, -, 3, -, 4, -, 5, -, 6, -, ...

A função h correspondente poderia ser $h = \begin{cases} (n+1)/2, \; se \; n \; é \; impar \\ indefinida, \; casocontrário \end{cases}$

A função h é uma **função parcial** de inteiros positivos.

As duas funções seguintes enumeram o conjunto de naturais pares:

$f(n) = \begin{cases} n, se \; n \; é \; par \\ indefinida, c.c. \end{cases}$ e $g(n) = 2n$

As listagens são -, 2, -, 4, -, 6, ... e 2, 4, 6, 8, ...

Qualquer conjunto S de inteiros positivos é enumerado bem simplesmente pela função parcial g, definida

$g(n) = \begin{cases} n, se \; n \in S \\ indefinida, c.c. \end{cases}$

- Uma função (parcial ou total) de naturais pode ter seu valor em qualquer conjunto. Seus argumentos devem ser todos naturais.

- O conjunto de cadeias finitas sobre o alfabeto é um exemplo de um conjunto enumerável infinito que não é um conjunto de naturais. Este conjunto é enumerável porque seus membros podem ser arranjados numa lista: os 26 primeiros elementos na lista são as 26 letras do alfabeto em sua ordem usual. Os próximos 676 (26^2) membros são as cadeias de duas letras. Os seguintes 17576 (26^3) são cadeias de 3 letras, e assim por diante.

- N é enumerável.

- Z é enumerável. A seguinte função enumera Z:
$$f(x) = \begin{cases} -(x+1)/2, \text{ se } x \text{ é ímpar} \\ x/2, \text{ se } x \text{ é par ou zero} \end{cases}$$

- Q é enumerável.

- R não é enumerável.

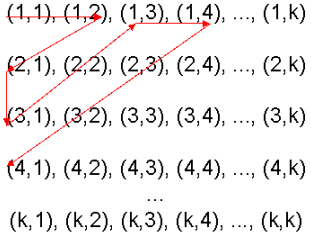
- I = R – Q ==> I não é enumerável.

O conjunto de todos os pares (x,y), com x, y Є N é enumerável.
Prova:

Seja a sequência a seguir, de todos os pares possíveis em N:

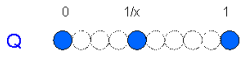
(1,1), (1,2), (1,3), (1,4), ..., (1,k)
(2,1), (2,2), (2,3), (2,4), ..., (2,k)
(3,1), (3,2), (3,3), (3,4), ..., (3,k)
(4,1), (4,2), (4,3), (4,4), ..., (4,k)
...
(k,1), (k,2), (k,3), (k,4), ..., (k,k)

onde k está no infinito.
Se tentarmos iniciar percorrendo as colunas da primeira linha, NUNCA chegaremos a (2,3), por exemplo, e o conjunto não poderá ser enumerado. Mas, percorrendo o conjunto de tal modo que i+j, em (i,j) gere uma sequência natural, percorreremos TODOS os pares, o que implica que enumeraremos o conjunto.
Podemos enumerar da maneira a seguir, começando em 2 e indo até k+k. Veja que alguns valores se repetirão, mas isso não é um problema.

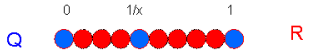


O Contínuo

N, Z e Q são conjuntos enumeráveis (e equivalentes!), sendo que seus membros são pontos discretos, isto é, entre dois membros consecutivos de um destes conjuntos não existe outro elemento que pertença ao conjunto. Por exemplo, em N não existe um número entre 5 e 6.



No conjunto R, porém, não existem "buracos". Seus membros são pontos contínuos e não há sentido em falar-se em um número entre dois reais consecutivos, porque não há "espaço".
A diferença entre N, Z, Q e R é melhor visualizada por gráficos de uma função definida num dos três primeiros conjuntos e outra definida em R. Vê-se que, no primeiro caso, o gráfico consistirá de pontos isolados, mas, no segundo caso, teremos pontos contínuos, formando uma linha.



Pontos discretos são até fáceis de visualizar quando estamos em N ou em Z, mas os pontos discretos em Q não são tão visíveis assim. Imagine um valor em Q que seja um número infinito, tipo 1/3. Como ele seria visualizado? Tudo bem que, no gráfico da função, basta apenas marcar um ponto ali, mas imagine esse valor dentro do conjunto Q, entre o 0 e o 1, já que o valor de 1/3 é 0.333333... continuando com 3 à direita até o infinito.
Nossa tendência natural para fazer esta visualização é a de imaginar o número como se ele estivesse sendo escrito numa página, da esquerda para a direita. Mas, se fizermos isso aqui dentro: Q = {...0, ..., 0.333333...,..., 1, ...}, expandindo o 0.333333 até o infinito, não conseguiríamos visualizar os demais números após o 0.333333 expandido, incluindo o 1 "logo" à direita!
Mas, há uma outra maneira de fazer esta visualização sem que deixemos de ver os demais valores à direita. Basta imaginar o número 0.333333 se expandindo perpendicularmente à página, indo em direção a seus olhos (saindo da frente da página) ou se afastando deles (fugindo pelo fundo da página para trás). Com isso, você pode conter o número dentro da estrutura como um ponto: Q = {...0, ..., 1/3...,..., 1, ...}. À direita dele você continua vendo os buracos (onde estão números que não pertencem a Q) e os demais valores. Disso, você nota que 1/3 não está infinitamente distante de zero, apesar de ser um número infinito. Assim, você pode marcar cada um dos membros de Q, enumerando-os.
Número infinito é uma coisa, intervalo infinito é outra coisa, diferente do primeiro.

Da mesma maneira, podemos visualizar os elementos de R ao preenchermos com eles os "buracos" em Q. Q, agora, foi transformado em R. Mas, o que são estes buracos, afinal? Veremos mais adiante.

Uma questão que surge em Q é: dados *a*, *b*, *c* ∈ Q, com *a* < *b* < *c*, sendo *a*, *b* e *c* números imediatamente consecutivos, o intervalo (buraco) entre *a* e *b* é igual ao intervalo entre *b* e *c*?
Se existirem **n** números racionais em [0, 1], então existem, exatamente, **n** números racionais em [1, 2], pois [0, 1] + 1 = [1, 2], para todo i em [0, 1].
Então, entre *a* e *b* existe a mesma quantidade de racionais que existem entre *b* e *c*.
Pelo que se discutiu acima sobre valores discretos, não existem racionais consecutivos sem um buraco entre eles (pelo menos é a essa conclusão que se chega logicamente).
Sendo *a* = 1/3, qual número seria *b*? 1/2? 1/4? A escolha rápida seria 1/4, mas, não é a correta. Nem 1/2 é correto. 1/2 é muito grande e 1/4 é muito pequeno (0.25 vem antes de 1/3).
A escolha mais adequada é 2/5 (0.4, que vem "logo" depois de 0.3). Mas, será que existe um racional entre 1/3 e 2/5? Vamos ver se existe um exatamente entre os dois: $(\frac{1}{3}+\frac{2}{5})/2=\frac{11}{30}$. Sim, existe. E entre 1/3 e 11/30? Sim: 21/60.
Continuando, teremos: 41/120, 81/240, 161/480... Sempre conseguiremos encontrar um número racional entre 1/3 e seu consecutivo "imediato", o que implica que não existe imediato!

Com um pouco de cálculo, descobrimos que a função $f(n)=\frac{1+10\times 2^n}{3\times 10\times 2^n}$, com n indo de zero até infinito, lista os racionais em $\left[\frac{1}{3},\left(\frac{1}{3}+\frac{2}{5}\right)/2\right]$.
Sendo *a* = 1/3, *b* = 11/30 e *c* = 2/5, sabemos agora que não existem dois racionais sem buracos entre eles.

O número decimal gerado por 1/3 não tem fim. Em sequência a ele deve vir um número com um 4 na última casa decimal de 1/3. Mas, esse novo número não pode ser gerado por 1/3, e nem será um imediato a 1/3. Portanto, ele é um buraco em Q. Infinitos outros podem vir após ele. Isso pode acontecer entre quaisquer intervalos em Q.
Então, os buracos entre *a*, *b* e entre *b*, *c* são iguais, mas infinitos. Se fossem diferentes, um dos intervalos seria finito, daí contável e enumerável, o que destruiria a discreção de Q naquele intervalo. A discreção precisa de buracos.

A função $f(x)=\frac{1}{x}$, com x ∈ N e x variando do infinito até 1 lista os racionais (não todos) no intervalo (0, 1]. Claramente, esta lista é um conjunto infinito. Se esse intervalo é infinito, muito mais infinito é o intervalo com os buracos.
Agora temos, além de um número infinito, números infinitos dentro de intervalos infinitos. Como visualizar isso?
Um número infinito dentro de um conjunto já sabemos visualizar. Visualizar intervalos infinitos tem que ser no sentimento. Se *a*, *b* e *c* são infinitos e pertencem a um conjunto S, e o intervalo entre *a* e *b* é infinito e também infinito entre *b* e *c*, visualizamos através da imaginação.

Se S é Q e quisermos caminhar de *a* até *b*, passando por todos os intermediários, como que rotulando (enumerando, contando) cada elemento, NUNCA chegaremos a *b*.
Esse forte sentimento que temos de que deveríamos chegar em *b* vem de nosso costume com coisas materiais, finitas e num intervalo finito (forçado por elas próprias).
Podemos ver a, b e c, mas, entre eles há a escuridão, o nada com alguma coisa. Ainda assim, em Q, podemos contar, enumerar.

Um número infinito é formado de um número infinito de partes (e partes infinitas, formam números infinitos?).
Uma quantidade infinita de partes (finitas ou infinitas) acumuladas formam um número infinito. Partes infinitas acumuladas finitas vezes podem formar um número infinito (1/3+1/3) ou um número finito (1/3+1/3+1/3).
Veja o 1/3. Ele pode ser gerado através da soma de $\frac{3}{10}+\frac{3}{100}+...+\frac{3}{\infty}$, ou $\sum_{x=1}^{\infty}\frac{3}{10^x}=\frac{1}{3}$; e 0.3, 0.03, 0.003, ... forma uma sequência infinita, entremeada de buracos que formam uma sequência infinita também.
Veja que se x começar em zero, teremos que $\sum_{x=1}^{\infty}\frac{3}{10^x}=\frac{10}{3}$, pois equivale a somar 3 a 1/3: $3+\frac{1}{3}=\frac{9}{3}+\frac{1}{3}=\frac{10}{3}$.

O que um número tem a ver com seus geradores, em termos de ser finito ou infinito?
Quaisquer números finitos acumulados finitas vezes geram um número finito. Um número infinito adicionado a números finitos, finitas vezes, gera um número infinito. Um número infinito adicionado finitas vezes a números infinitos gera um número infinito ou finito.
Um número inteiro pode ser dividido em infinitas partes, todas inteiras ou em partes inteiras e partes infinitas. Pode ser dividido só em partes infinitas. Todas essas partes são contáveis. Reacumulando estas partes, o número inteiro é recuperado.
E um número infinito? Pode ser dividido em infinitas partes também.

Enfim, como visualizar os números reais? O que eles são?
Se se for falar em termos de existência matemática, o conjunto R não existe de per si. Veja porque: temos os naturais N, os inteiros Z = {-N, 0, +N} e temos os racionais Q = $\left\{\frac{Z}{Z}\right\}$, sem o zero no denominador.
O próximo conjunto, R, deveria ser gerado em termos de N também.
É fácil construir uma função (ou máquina) para gerar todos os racionais:
$q(x,y)=\frac{x}{y}$, sendo x e y primos entre si, ou seja, eles não possuem um divisor comum diferente de 1.

Só que, ao rodarmos essa máquina, ela vai pular muitos números, como o número л, por exemplo. Pulará também $\sqrt{2}$.

Números como o л e $\sqrt{2}$ não podem ser gerados através da razão entre dois inteiros, simplesmente porque não é possível achar dois inteiros cuja razão resulte num daqueles números, ou seja, não л e $\sqrt{2}$ são racionalizáveis, sendo, assim, irracionalizáveis, ou, mais simplesmente *irracionais*. Temos, então, um conjunto à parte de Q (os racionais), que chamaremos de I.
Finalmente, temos a definição de R: $R = Q \cup I$.
Intuitivamente, podemos visualizar todo o conjunto R, com I fechando todos os buracos de Q. Você ainda pode caminha dentro de R, pisando em cada número até ao infinito, andando agora por uma estrada lisa, em vez da esburacada avenida Q!
O que você não consegue é construir uma função f(x) = y × i, em que x, y ∈ Z e i ∈ Z. Se você pudesse, então, para cada N, você poderia fazer corresponder um I. Em outras palavras, você poderia fazer uma contagem dos elementos de I.

Então, contagem (um conjunto é contável) tem a ver com uma função que mapeia o conjunto N em outro conjunto, e não com o botar o dedo em cada elemento do segundo conjunto.

O que é o conjunto I?

- Para começar, todo elemento de I tem que ser um número infinito, pois, para qualquer i finito é possível achar x, y ∈ Z, tal que i = x / y.
- Ser infinito não é suficiente, pois muitos elementos de Q são infinitos, como 1/3, por exemplo.
- Todo I tem casas decimais, caso contrário ele seria Z. Óbvio.
- O que diferencia um I de um Q infinito é que, no Q, as casas decimais entram num padrão de repetição, enquanto isso não acontece em I.
- Veja este grupo de racionais gerados pela função q(x,y) anterior. Note os grupamentos das casas decimais. São números infinitos, mas limitei a quantidade de casas decimais.

  329/33
  9.969696969696969696969696969696
  329/34
  9.6764705882352941176470588235294117647058823529411764705882352941176470588235294117

- O padrão de repetição começa a aparecer quando o resto da divisão passa a ser sempre o mesmo em cada período. Como o divisor é fixo, quando o resto se fixar, sendo sempre o novo divisor, o quociente também ficará fixo, sendo sempre o mesmo, criando um período infinito. Essa característica de Q nos permite afirmar, então, que numa divisão que gera um Q infinito, a partir de um certo ponto o resto será sempre o mesmo. Pelo mesmo motivo, e por causa de I, uma divisão inteira ou vai gerar um número finito (pertencente a Q) ou vai entrar num ponto em que o resto é sempre o mesmo, gerando um número infinito pertencente a Q, ou seja, **não existe uma divisão inteira que gera casas decimais aleatórias**!

O conjunto dos números irracionais é definido assim:

$$I = \{ \nexists\, m,n \in Z \mid x = m/n \}$$

Como m/n sempre vai gerar um padrão de repetição nas casas decimais e I não tem esse padrão, m/n não gera I.

Alguns irracionais são famosos, como л, *e* e $\sqrt{2}$. Veja na listagem a seguir onde, mais ou menos, uma divisão inteira os pulam.

$\sqrt{2}$ = 1,4142135623730950	*e* = 2.71828182845904523	Л = 3,14159265358979323
1.41418 = 379/268	2.71812 = 405/149	3.14141 = 311/99
1.41418 = 997/705	2.71815 = 704/259	3.14145 = 955/304
1.41419 = 618/437	2.71818 = 299/110	3.14146 = 644/205
1.41419 = 857/606	2.71821 = 791/291	3.14148 = 977/311
1.4142 = 239/169	2.71823 = 492/181	3.14151 = 333/106
1.41421 = 816/577	2.71825 = 685/252	3.14155 = 688/219
1.41422 = 577/408	2.71827 = 878/323	3.14159 = 355/113
1.41422 = 915/647	2.71831 = 193/71	3.14163 = 732/233
1.41423 = 338/239	2.71835 = 859/316	3.14167 = 377/120
1.41423 = 775/548	2.71837 = 666/245	3.1417 = 776/247

O número irracional está entre os dois valores pintados de vermelho.
A listagem foi gerada através de x/y, com cada um variando de 1 até 1000. Claro que 1000 é muito pouco para delimitar exatamente o buraco onde o pi se encontra (supondo que ele é delimitado por dois racionais). O mesmo se aplica ao cálculo da raiz de 2 e do número *e*.
Veja que no cálculo da raiz de 2, cálculos diferentes deram o mesmo resultado. Isso se deve à limitação da quantidade de casas decimais, porque, se considerarmos 10 casas decimais, por exemplo, teremos que:

379/268 = 1.4141791044
997/705 = 1.4141843971

Os valores só seriam exatamente iguais se uma fração pudesse ser reduzida à outra através de divisões/multiplicações inteiras. Apareceria um valor igual também (considerando toda a listagem) se o numerador e o denominador de cada fração tiverem um divisor comum, o que não é o caso das duas frações acima, também.

Os números л e *e* são chamados de irracionais *transcendentais*, enquanto que $\sqrt{2}$ é chamado de irracional *algébrico*. A diferença é que números algébricos são raízes de alguma equação polinomial de coeficientes racionais, enquanto os transcendentais não.

Prova de que $\sqrt{2}$ é um número irracional.

Antes, vamos relembrar que:

- Vimos que todos os racionais são gerados por *a*/*b*, onde *a* e *b* não tem fatores (divisores) comuns (o MDC entre eles é 1). Portanto, um deles **tem que** ser ímpar, isto é, ambos não podem ser par. Isso é óbvio (senão, 2 seria um fator comum entre eles).
- Todo par tem a forma 2n, onde n é natural. Todo ímpar tem a forma 2n+1.
- O quadrado de qualquer par é par também: $(2m)^2 = 4m^2 = 2(2m^2)$, que está na forma 2n, portanto, par.
- O quadrado de qualquer ímpar é ímpar também: $(2m+1)^2 = 4m^2 + 4m + 1 = 2(m^2 + 2m) + 1$, que está na forma 2n+1, portanto, ímpar.

A prova:

Se $\sqrt{2}$ é racional, então existem *a* e *b*, **sem fatores comuns**, tal que $\sqrt{2} = \frac{a}{b}$. Vamos ver:

$\sqrt{2} = \frac{a}{b} \Rightarrow 2 = \left(\frac{a}{b}\right)^2 \Rightarrow 2 = \frac{a^2}{b^2} \Rightarrow a^2 = 2b^2$, mas, isto implica que *a* é um número par. Até aí, tudo bem...

Se *a* é par, podemos fazer *a* = 2*k*, para um *k* qualquer. De $2 = \frac{a^2}{b^2}$, temos que $2 = \frac{(2k)^2}{b^2} \Rightarrow 2 = \frac{4k^2}{b^2} \Rightarrow 2b^2 = 4k^2 \Rightarrow b^2 = 2k^2$. Mas, isto implica que *b* também é par, como *a*. Chegamos a uma contradição, pois sendo *a* e *b* números pares, eles têm um divisor comum, o que contraria a nossa afirmação inicial de que eles **não têm fatores comuns**. Então, não existem a e b, tal que $\sqrt{2} = \frac{a}{b}$. Portanto, $\sqrt{2}$ é irracional (veja definição de I).

As provas de que л e *e* são irracionais são bem mais complicadas. Talvez por serem números transcendentais. As provas, normalmente, envolvem as fórmulas usadas para aproximar o número e a divisão de dois inteiros (como fizemos com raiz de 2). No caso do л, usa-se, por exemplo, a fórmula para o cálculo do comprimento (perímetro) de um círculo: C = лD, onde D é o diâmetro do círculo.

No caso do *e*, usa-se $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$.

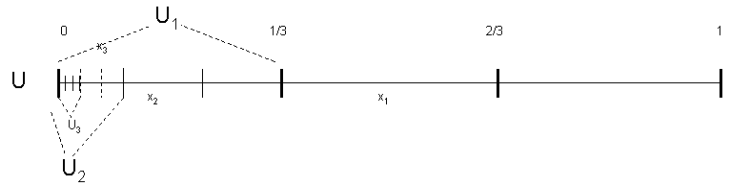
Bom, devido ao conjunto I não ser enumerável, decorre que R não é enumerável, o que implica que a cardinalidade de R é diferente da cardinalidade de Q.

Teorema: O intervalo fechado [0, 1] de números reais não é enumerável.
Prova 1:

Suponhamos que ele seja enumerável. Então, podemos ter uma sequência $x_1, x_2, x_3, ...$ que contém todos os pontos de U = [0, 1].
Dividamos U em três partes iguais de 1/3 cada uma, assim:

[0, 1/3][1/3, 2/3][2/3, 1]

É claro que o ponto x_1 não pode pertencer aos três subintervalos simultaneamente. Pelo menos um destes subintervalos falha em conter o ponto x_1.
Chamemos, agora, de U_1, o subintervalo que não contém x_1, e escolhamos $U_1 = [0, 1/3]$.
Dividamos U_1 do mesmo modo que U, em três subintervalos de igual tamanho de denotemos por U_2, o sub-subintervalo que não contém o ponto x_2.
Repitamos o processo com U_2, e assim por diante, obtendo algo assim:



Obtemos, como resultado, uma sequência infinita de intervalos aninhados: $U \supset U_1 \supset U_2 \supset U_3 \supset ...$, que possuem a propriedade $x_n \notin U_n$.
Como o tamanho do intervalo U_n é $\frac{1}{3^n}$ (primeiro 1/3, dando 3 intervalos de 1/3; depois 1/3, que o tamanho de U_1, dividido por 3, que dá 1/9; depois [1/9]/3, que dá 1/27, e por aí vai), este tamanho tende a zero quando n vai para o infinito. Eventualmente, vamos chegar num ponto б que pertence a todos os intervalos (como os intervalos estão aninhados, aquele lá no infinito está dentro de todos os demais, portanto, um ponto aqui pertence a todos os intervalos), isto é, б está em U e é único.
Sendo um ponto de U, б deve aparecer na sequência $x_1, x_2, x_3, ...$, mostrada no início, isto é б = x_i para um dado n = i. Mas, isso é claramente impossível, porque, para todo n temos que $x_n \notin U_n$, o que nos leva a concluir que $x_i \notin U_i$ ou que б$\notin U_б$! Daí, segue que б $\neq x_n$, para qualquer n, isto é, б não é um ponto da sequência $x_1, x_2, x_3, ...$ Isto é uma contradição, daí, U não é enumerável.

Prova 2:

Consideremos o oposto, assim, A = { x_1, x_2, x_3, ...}, isto é, os elementos de [0,1] podem ser escritos numa sequência. Cada elemento de A pode ser escrito sob a forma de um decimal infinito, como se segue:

$x_1 = 0.a_{11}a_{12}a_{13}...a_{1n}...$
$x_2 = 0.a_{21}a_{22}a_{23}...a_{2n}...$
$x_3 = 0.a_{31}a_{32}a_{33}...a_{3n}...$ (L)
...
$x_n = 0.a_{n1}a_{n2}a_{n3}...a_{nn}...$

onde $a_{ij} \in \{0, 1, 2, ..., 9\}$ e onde cada decimal contém um número infinito de elementos diferentes de zero (escreva 1 como 0.999 e todos, como ½ por exemplo, com 0.49999...).
Construa, agora, o número real $y = 0.b_1b_2b_3.....b_n...$, que pertencerá (por suposição) a A da seguinte maneira: escolha b_1 de tal modo que $b_1 \neq a_{11}$ e $b_1 \neq 0$; escolha b_2 de tal modo que $b_2 \neq a_{22}$ e $b_2 \neq 0$, e assim por diante, até b_n com a_{nn}.
Onde está y na listagem (L) acima? Observe que $y \neq x_1$, pois $b_1 \neq a_{11}$; $y \neq x_2$, pois $b_2 \neq a_{22}$; ... $y \neq x_n$, pois $b_n \neq a_{nn}$, isto é, $y \neq x_n$ para todo $n \in N$, assim, $y \notin A$, que contradiz a suposição de que $y \in A$. Desse modo, a suposição de que A é enumerável conduz a uma contradição, consequentemente, A não é enumerável.
Conseguimos um número real que não foi listado por um número natural, o que implica que existem mais reais do que naturais.
De novo, não há uma função em N que gere todos os elementos de R.
Note que "andamos" na diagonal da tabela formada pela lista (L), modificando cada dígito em cada linha, obtendo um novo número diferente de todos os que estão na tabela. Fizemos uma diagonalização.

- [0, 1] tem cardinalidade $\mathfrak{c}$, e assim também todos os conjuntos equivalentes a [0, 1].
- A função y = a + (b-a)x, com f:[0,1] → [a,b] é bijetora.
- A função f:(-л/2, л/2) → R, definida por f(x) = tgx é bijetora. Daí R ~ (-л/2, л/2).
- A união de conjuntos de cardinalidade $\mathfrak{c}$ tem cardinalidade $\mathfrak{c}$.
- O conjunto de números irracionais I tem cardinalidade $\mathfrak{c}$. Daí, números transcendentais (não-algébricos) devem existir.
- Seja $A \supset A_1 \supset A_2$. Se $A_2 \sim A$, então, também $A_1 \sim A$.
- Se o conjunto M tem cardinalidade μ e o conjunto T, de todos os subconjuntos de M, tem cardinalidade τ, dizemos que $\tau = 2^{\mu}$.

Teorema: O conjunto de todos os subconjuntos de N, 2^N(sua cardinalidade) não é enumerável.
Prova 1:

Suponhamos que 2^N é enumerável. Daí há uma sequência N_0, N_1, N_2,... que contém todos, **e somente todos**, os subconjuntos de N.
Construamos um novo conjunto como se segue:

$$M = \{n \mid n \notin N_n, n = 0,1,2,...\}$$

Assim, M contém 0 (zero) só se $0 \notin N_0$; contém 1 só se $1 \notin N_1$, e assim por diante. Claramente M é um subconjunto de N, e por isso M deve estar na lista N_0, N_1, N_2,... acima (por suposição de que a lista possa ser feita). Daí, devemos ter $M = N_k$ para algum k. Mas isso é impossível, pois k está M se, e só se, k não está em $N_k = M$ (pela definição de M). Assim, chegamos a uma contradição ao supor que o conjunto 2^N é enumerável, pois conseguimos um seu elemento que não está na lista, daí, 2^N não é enumerável.

Prova 2:

Podemos, em vez de pensar nos subconjuntos S_1, S_2, S_3,..., de N, pensar em suas funções características, f_1, f_2, f_3,.... Vimos que há bijeção entre S_i e f_i.
Pela suposição de que 2^N é enumerável, podemos ter uma lista das funções características. Podemos, então, construir a seguinte tabela:

	1	2	3	4	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	...
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	...
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	...
...	...	...	...	...	...

A linha n representa a função f_n, e assim o conjunto S_n. A linha $f_n(1)f_n(2)f_n(3)$... é uma sequência de uns e zeros dependendo se, para um dado p, $f_n(p)$ pertence ou não a S_n.
Os pontos da diagonal da tabela formam uma sequência de zeros e uns.

Esta sequência determina um conjunto de inteiros positivos que pode muito bem estar na tabela anterior. Em outras palavras, pode muito bem haver um natural *d* tal que o conjunto S_d não é outro senão o conjunto da diagonal. A sequência de zeros e uns da d-ésima linha concordaria com a sequência diagonal ponto por ponto, assim:

$$f_d(1) = f_1(1),\ f_d(2) = f_2(2),\ f_d(3) = f_3(3),\ ...$$

Construímos agora um conjunto que com certeza não está na tabela: o conjunto antidiagonal, que consiste de inteiros positivos que não estão no conjunto diagonal, isto é, esse conjunto diferirá dos outros (de todos os da tabela!) em pelo menos um elemento!
A sequência antidiagonal correspondente é obtida pela transformação de zeros em uns e vice-versa na diagonal. Podemos obter a sequência assim:

$$1- f_1(1),\ 1- f_2(2),\ 1- f_3(3),\ 1- f_4(4),\ ...$$

Esta sequência, com certeza, não está na tabela dada, pois se assim for, isto é, se ela aparece na m-ésima linha teríamos:

$$f_m(1) = 1- f_1(1),\ f_m(2) = 1- f_2(2),\ ...,\ f_m(m) = 1- f_m(m)$$

Mas, a última equação é falsa, pois, se $f_m(m) = 0$, então $0 = 1$; se $f_m(m) = 1$, então $1 = 0$. Daí, concluímos que 2^N não pode ser enumerado, ou seja, novamente, 2^N não é enumerável.

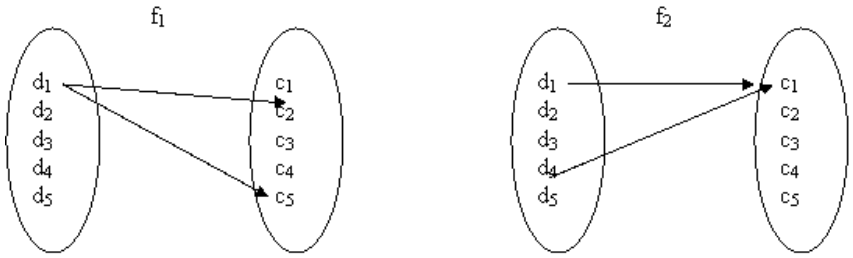
FUNÇÕES

Definições:

Uma **função** é uma transformação ou mapeamento dos elementos de um conjunto em outro. O conjunto-origem é chamado de *domínio* e o conjunto destino é chamado de *imagem* ou *codomínio*.

Uma definição formal de uma função *f* é:

Sejam os conjuntos D = (d_1, d_2, ...,dn) e C = { c_1, c_1, ...,c_n}, representando o *D*omínio e o *C*odomínio, por exemplo. Uma função f : D → C (a seta significa: mapeia os elementos de D em C) é definida assim:
Se $f(d_i) = c_j$ e $f(d_i) = c_k$ então $c_j = c_k$. Isso implica que, para cada elemento em D existe um, e apenas um, elemento em C. Um elemento em C, porém, pode corresponder a um ou mais elementos em D.
Na figura a seguir, f1 não é uma função, mas f2 é.



Se todos os elementos de D tiverem um correspondente em C, f é uma **função total.** Se pelo menos um elemento de D não tiver um correspondente em C, f é uma **função parcial**. Uma função *f* é parcial em *N* se $Domf \subset N$. Uma função *f* é total em *N* se *Domf = N*. Daqui se conclui que toda função total é, ao mesmo tempo, parcial.

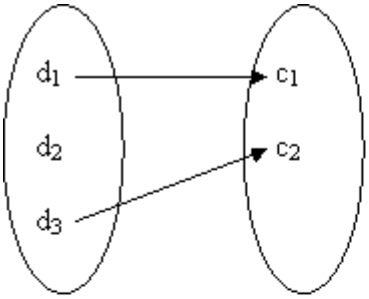
Seja $g : Z \rightarrow Z \mid g(n) = \sqrt{n}$. Para cada *n* em Z, a imagem tem que continuar em Z. Esta função, então, é indefinida para valores que não são quadrados perfeitos. Assim, g é uma função parcial, pois o domínio de g está em Z, mas não é todo o conjunto Z.

Se $f(d_i) = c_k$ e $f(d_j) = c_k$, então di = dj e a função f é **injetora**. Em outras palavras, para cada elemento em C existe um, e apenas um, elemento em D.

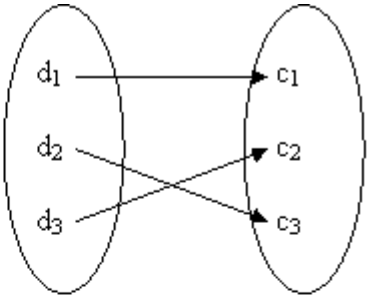
Se para qualquer c_j Є C existe d_i Є D, a função f é **sobrejetora**. Em outras palavras, a cada elemento de C corresponde pelo menos um elemento em D.

Se f é injetora e sobrejetora, f é **bijetora**.

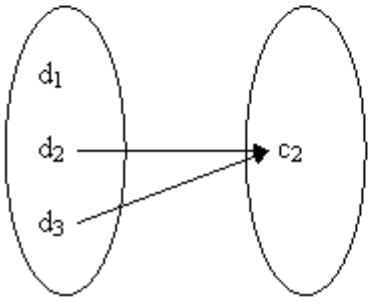
Veja os exemplos que se seguem.



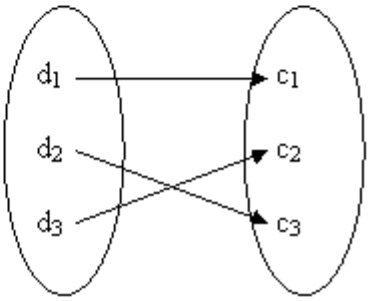
injetora parcial



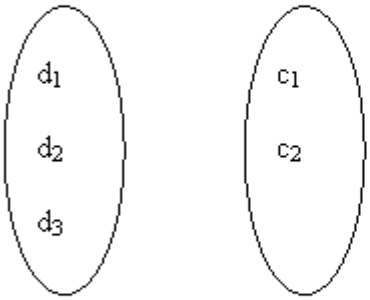
injetora total, e bijetora, portanto sobrejetora também



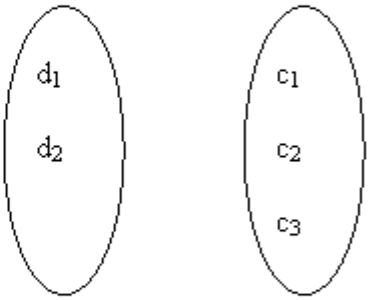
sobrejetora parcial



sobrejetora total, e bijetora, portanto injetora também.



Considerando uma função total, a relação aqui só pode ser de injeção ou sobrejeção, mas não ambas, assim não tem como termos uma bijeção.



Aqui não é possível injeção e nem sobrejeção.

Um conjunto A é **infinito** se existir f injetora f : A → A, tal que f(A) é subconjunto próprio de A (veja Conjuntos Equivalentes).
Se há injeção para um subconjunto de N, A é **contável**. Se há bijeção para N, A é **infinitamente contável**.
Uma função é **unária** quando aceita apenas um argumento, e.g., f(**x**). É **binária** quando aceita dois argumentos, e assim por diante.
Uma **função enumerativa** é uma função que, dado um argumento de N, reproduz todas as funções que são elementos de um conjunto de funções.

Sejam A, B dois conjuntos e seja S uma coleção de funções parciais f : A→ B. Dizemos que uma função binária u : N × A → B enumera S se, e somente se, sempre que g : A → B é uma função arbitrária, então g está em S se, e somente se, existe um número natural k tal que g(x) = u(k,x), para todo x Є A (lembre-se que N × A é o produto cartesiano de N por A, ou seja, a função u aceita dois argumentos, um pertencente a N e outro pertencente ao conjunto A).
Além disso, dizemos que S é contável se, e somente se, S tem uma função enumerativa, ou S é vazio. Se S é contável, então u(0,x), u(1,x), u(2,x), ... é uma sequência que contém todos e somente os elementos (funções) de S. Claramente, u é uma função total se cada função em S é total.

Exemplos

1) S consiste de todas as funções da forma 2x + 3, 4x + 3, 6x + 3, ... Uma função enumerativa é dada por u(k,x) = 2(k+1)x + 3.
2) S consiste das funções 0x – 0, x – 1, 2x – 2, 3x – 3, ... Note que todas são funções parciais em N (para x=0), exceto 0x – 0. Uma função enumerativa é u(k,x) = kx – k

O conjunto de todas as funções unárias totais de N para N é incontável (não é possível encontrar uma função que enumere estas funções unárias):

S_1 = { f : N → N | f(x) = z, com x, z Є N}

Prova 1:

Suponha que S_1 é contável com função enumerativa *u(k,x).*
Defina $g(x) = u(x,x) + 1 \ para \forall x \in N$. Claramente g é uma função unária total. Assim, g deve estar em S_1. Isso significa que, para algum k, *g(x) = u(k,x),* para qualquer x pertencente a N (a função u com o parâmetro k enumera – ou rotula – a função g dentro de S_1). Mas, isso é claramente impossível, pois, se pegarmos k = x em *g(x) = u(x,x)* + 1, obtemos *g(k)* = u*(k,k)* + 1. Mas, usando k em *g(x) = u(k,x),* obtemos *g(k) = u(k,k).* Isso implica que *g(k) = u(k,k)* + 1 = *u(k,k)*!!!
Se *u(k,k)* for zero, teremos que 1 = 0. Isso é uma contradição.
Assim, a suposição de que S_1 é contável deve ser falsa.

Note que a conclusão acima falharia se S_1 fosse o conjunto de todas as funções unárias parciais, pois u(k,k) + 1 = u(k,k) não seria uma contradição se u(k,k) for indefinido, ou seja, a imagem de u(k,k) não está em N (fosse -1, por exemplo). Contudo, a prova acima mostra que o conjunto de todas as funções unárias parciais contém um subconjunto incontável. Assim, o próprio conjunto de todas as funções parciais é incontável também (R é incontável porque contém I).

Prova 2:

Se S_1 for contável, podemos ter a sequência S = f_1, f_2, f_3,.... Mas, a função definida por diag(i) = f_i(i) + 1, que é uma função de N para N, não pode pertencer ao conjunto, pois diag seria f_j para algum j, mas, isso implicaria que diag(j) ≠ f_j(j) ou f_j(j) é a mesma coisa que f_j(j) + 1, o que não pode ser.

Funções Recursivas

Definições:

Símbolo: Um caracter, letra ou marca.
Alfabeto: Um conjunto finito de símbolos. Um alfabeto é, normalmente, denotado pela letra grega sigma (Σ).
Cadeia ou **Palavra**: Uma sequência finita de símbolos de um alfabeto.
Linguagem: Um conjunto de cadeias de um alfabeto. Este conjunto pode ser vazio, finito ou infinito.
Σ* : Todas as combinações possíveis de palavras do alfabeto Σ, formando uma linguagem.

Uma função é dita **recursiva** (computável, decidível) se existe um algoritmo (um programa, uma máquina) que a calcula, que a computa, isto é, mimica a função.
O termo *recursiva* é também usado para algumas funções (ou programas) que aceitam a si mesmos como argumento, como vimos em *Recursão*, mas, não é este o sentido usado aqui na teoria da computação.
Ser computável não necessita ser eficientemente computável. Pode ser que o algoritmo leve muito tempo para parar, se parar. Considera-se o tempo e o espaço (para os dados) ilimitados. O algoritmo deve ser finito (em número de passos) e, eventualmente, termina devolvendo um valor.

Uma função computável por algoritmo equivale a um mapeamento produzido pelo algoritmo, isto é, a cada função computável corresponde pelo menos um algoritmo que a calcula.
Uma mesma função pode ter vários algoritmos (diferentes entre si) que a computa, mas, um algoritmo não computa mais que uma função, ou seja, ele age de maneira única.
Sim, pois, se a função f produz o mesmo resultado que a função g, com os mesmos argumentos, então f = g. Por exemplo, não existe a função *f*, tal que $f(x) = x^2 = \sqrt{x}$. Veja isto: $f(x) = \sqrt{x}$. No conjunto imagem, existem dois elementos diferentes que satisfazem esta função. São eles: $\sqrt{x}$ *e* $-\sqrt{x}$, mas, claramente, são necessários dois algoritmos diferentes para computarem os resultados, ou seja, um para $f(x) = \sqrt{x}$ e outro para $g(x) = -\sqrt{x}$, e $f(x) \neq g(x)$.

Existem funções que são bem descritas (de maneira finita), mas que não são efetivamente computáveis (não existe um algoritmo para computá-la – ele teria que parar e produzir um resultado).

Exemplo: *Todo número par maior que 2 é soma de dois números primos* (Conjectura de Goldbach - Christian Goldbach - 1742).

Uma função equivalente seria: $h(x) = \begin{cases} 1, \text{se a conjectura é verdadeira} \\ 0, \text{se a conjectura é falsa} \end{cases}$

Claramente, h é uma função constante (se a conjectura for verdadeira, qualquer x será mapeado em 1, por exemplo) e é bem definida e bem descrita, portanto, deveria existir um algoritmo para representá-la, mas, como é ele?
Uma tentativa seria o procedimento:

EXAMINE CADA NÚMERO PAR ATÉ APARECER UM QUE NÃO É A SOMA DE DOIS PRIMOS. SE E QUANDO ELE APARECER, DEVOLVA ZERO.

Ora, se a conjectura for verdadeira, esse procedimento nunca terminará (o conjunto dos números pares é infinito e nenhum valor que satisfaça a condição será encontrado), daí, esse procedimento representa uma função parcial (é como se a função pegasse um argumento que não gera um mapeamento no conjunto imagem, que é o conjunto dos pares que não são soma de dois primos – a função fica indefinida para aquele argumento). A função é computável, mas não conseguimos escrever um algoritmo que a compute.

Um conjunto A de números naturais é chamado de **computável** (recursivo, decidível) se existir uma função computável total *f* tal que para qualquer n Є N, *f*(n) = 1 se n Є A e *f*(n) = 0 se n não pertence a A.

O conjunto A é **computavelmente enumerável** (recursivamente enumerável, semidecidível) se existir uma função computável *f* tal que, para cada n, *f*(n) é definida se, e somente se, n Є A.
Assim, um conjunto é computavelmente enumerável se, e somente se, ele é domínio de algumas funções enumeráveis.

Uma função unária parcial ou total *f* é dita **efetivamente computável** se existir um processo efetivo que, quando dados n argumentos $x_1, x_2, ...,x_n$, o processo

> Eventualmente páre, produzindo $f(x_n)$ se $f(x_n)$ é definida.
> Nunca páre, se $f(x_n)$ for indefinida.

O conjunto S_2 de todas as funções efetivamente computáveis em N é contável.
Prova:

> Cada função do conjunto possui um "algoritmo" que a computa. Mas, um algoritmo é formado de cadeias de símbolos do alfabeto Σ (pode ser o alfabeto {0, 1} usado pelos computadores), ou seja, de Σ*. Sabemos que Σ* é contável, daí, o conjunto de algoritmos é contável. Então, o conjunto das funções efetivamente computáveis é contável.

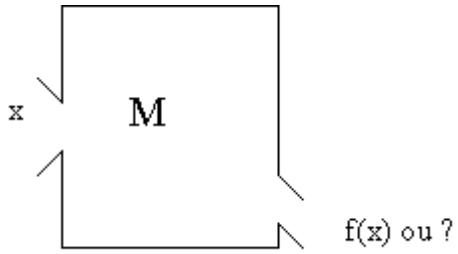
Todo programa é finito. Assim, o conjunto de todos os programas é contável. Daí, o conjunto de todas as funções computáveis de N para N é contável. Mas, o conjunto de todas as funções de N para N não é contável. Assim, muitas funções de N para N não são computáveis.
Um programa escrito em código binário pode ser visto com um único número binário, equivalente a um número natural. Com isso, o conjunto de todos os programas equivale a um conjunto de números naturais, sendo, por isso, contável.

Existem funções totais em N que não são efetivamente computáveis.
Prova:

Vimos que $S_1=\{f \mid f : N \rightarrow N\}$ não é contável.
Vimos que $S_2=\{f \mid f : N \rightarrow N$ efetivamente computável$\}$ é contável e $S_2 \subset S_1$.
Daí, existe um número incontável de funções não computáveis.

Existem mais funções de N para N do que programas que mapeiem um número natural para outro número natural.

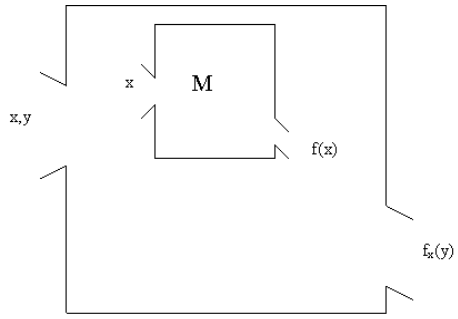
Se existe um procedimento (máquina) que calcula uma função *f*, então *f* é computável, e, para qualquer x pertencente a N, a máquina M devolve *f*(x) se x Є Dom*f*.



Se $x \notin Domf$, nada é obtido. É como se a máquina ficasse bloqueada, como se a computação nunca terminasse.

Uma função à qual corresponde uma máquina desse tipo é chamada de **função parcialmente computável**.

Para todo problema solúvel por um algoritmo (o algoritmo existe e é apresentado), é possível obter-se um procedimento para resolvê-lo, isto é, existe uma função (máquina) F tendo como entradas x, y Є N, em que x é o índice (rótulo) da função que se quer computar e y é o argumento para essa função: $F(x,y) = f_x(y)$.



Note que há uma sobrejeção de N no conjunto de funções, pois, a cada f_x corresponde pelo menos um x Є N.
Agora, a cada f(y) corresponde um y Є N? E se, para um y_0 Є N, $f_x(y_0)$ não estiver definida? Isto implica f_x parcial. Equivale a dizer (veja acima) que a máquina M_x não para nunca.

É possível, ou não, construir um procedimento (máquina) que resolve se uma dada função f_x é definida para um certo y?
Isto equivale a perguntar:

> Existe uma máquina **M** que decide se a máquina **m** para com um dado **n**?
>
> onde m calcula f_x com o dado y, ou seja, m = M_x.

Equivale, também, a perguntar: É possível construir um programa geral G que decide se um programa P qualquer para com um dado x?
Se G existe, ele teria a forma da seguinte função total h:

$$h(x,y) = \begin{cases} 1, \ se \ f_x(y) \ é \ definida \\ 0, \ se \ f_x(y) \ é \ indefinida \end{cases}$$

Com isso, podemos definir uma função *g* assim: $g(x) = \begin{cases} 1, \ se \ h(x,x) = 0 \\ indefinida, \ se \ h(x,x) = 1 \end{cases}$
isto é, g é definida se f_x for indefinida, ou, g(x) não dá resultado se $f_x(x)$ der um resultado.
A função *f* pode receber seu próprio índice (um número natural que a enumera) como argumento sim, e pode não estar definida nele.

A função *g*, claramente, é computável, pois, podemos computá-la pela suposição da existência da função *h*. E *g* é uma função parcial.
Ora, se o procedimento *h* decide sobre $f_x(y)$, também é capaz de decidir sobre *g*. Então, existe um x0 Є N tal que $f_{x0}(x) = g(x)$ correspondendo a uma máquina M_{x0}.
Podemos fazer x = x0 e perguntar: x0 Є *Dom*g = *Dom*f_{x0}?:

$$g(x0) = \begin{cases} 1, \ se \ h(x0,x0) = f_{x0}(x0) = g(x0) \ é \ indefinida \\ indefinida, \ se \ h(x0,x0) = g(x0) \ é \ definida \end{cases}$$

A conclusão que se tira é que: $g(x)$ *definida* $\Leftrightarrow g(x)$ *indefinida*. Isso é uma contradição, por isso, o programa G não pode existir.
Portanto, não se pode decidir se um dado x Є N pertence ou não ao domínio de uma função f_i qualquer. Pode-se saber que se x não pertence ao domínio da função, então a função é indefinida para aquele valor, mas não há como saber, não há como testar se x pertence ou não ao domínio da função. Daí, o procedimento ou máquina M, ou programa geral G, não existe, pois, a suposição de sua existência leva a uma contradição.

Este problema é conhecido como Problema da Parada (The Halting Problem). Vamos tentar deixá-lo mais claro com mais exemplos e discussões.

Analisando o Problema da Parada

Vamos repetir a argumentação anterior, rapidamente, através da construção de uma série de algoritmos.
Um programa de computador é uma série de dígitos 0 e 1. Assim, qualquer programa é, também, dado, e um programa pode ser construído para aceitar ele mesmo como dado.
Construamos um programa P que analisa um programa Q que aceita ele mesmo como argumento:

$$P(Q) = \begin{cases} Pare\ \ se\ Q(Q)\ \ pára \\ Não\ \ pare\ se\ Q(Q)\ não\ \ pára \end{cases}$$

Construamos, então, um programa D que aceite o programa Q como argumento:

$$D(Q) = \begin{cases} Não\ \ pare\ se\ Q(Q)\ \ parar \\ Pare\ se\ Q(Q)\ não\ \ parar \end{cases}$$

Como D é um analisador de programas, como ficaria D(D)? Vamos ver:

$$D(D) = \begin{cases} Não\ \ pare\ se\ D(D)\ \ parar \\ Pare\ se\ D(D)\ não\ \ parar \end{cases}$$

Mas, por que, aqui na nossa cabeça, essa contradição não parece clara, parece arranjada? Vamos analisar mais...

Vamos recolocar assim:
P é um programa que analisa se um programa qualquer vai parar ou não ao ser executado.

```
P(anyprogram)
{
        Se anyprogram para, devolva "SIM";
        Se anyprogram não para, devolva "NÃO";
}
```

Escrevamos o seguinte programa Q:

```
Q(anyprogram)
{
        Se P(anyprogram) == "SIM", entre em loop;
        Se P(anyprogram) == "NÃO", páre;
}
```

Aqui, o programa Q não vai parar se a subrotina P disser que *anyprogram* para, e vai parar se a subrotina P disser que anyprogram não para.
Como *anyprogram* pode ser qualquer programa, ele pode ser o próprio Q, certo? Então, vamos ver o que acontece quando fazemos *anyprogram* = Q, no programa acima.

```
Q(Q)
{
        Se P(Q) == “SIM”, entre em loop;
        Se P(Q) == “NÃO”, páre;
}
```

Se P, ao analisar Q, disser que ele para, vemos que, na verdade Q não para; se P disser que Q não para, nós forçamos Q parar. Assim, forçamos P a “mentir”, a apresentar uma contradição.
Como P analisaria Q(Q)? Vamos ver.

```
P(“Q(Q)”)
{
        Se [Se P(Q) == “SIM”, entre em loop; Se P(Q) == “NÃO”, páre;]
            para, devolva “SIM”;
        Se [Se P(Q) == “SIM”, entre em loop; Se P(Q) == “NÃO”, páre;]
             não para, devolva “NÃO”;
}
```

Quando P pegar Q, ele vai se ver em Q. Ele vai analisar uma análise que ele fará de Q. Haverá um aninhamento (recursão) sem fim, não é?
Isso ainda não está legal...

Vamos tentar construir um programa que analise qualquer outro programa e informa se esse programa para ou não com uma certa entrada. E se ele não tiver nenhuma entrada?

```
P(f,x)
{
        Se Tempo(f(x)) > 5h, devolva “timeout”
        senão devolva 1
}
```

Talvez f(x) parasse depois de 6 horas rodando... talvez nunca pararia.
Se nunca parar, como P decide isso? Ele não tem que decidir com base no resultado da execução de f, senão corre o risco de não poder decidir, caso f não páre.
O que deve ser feito é uma análise do argumento x em relação à transformação que ele sofre quando f executa. Essa transformação pode ser composta de um passo ou de vários passos. Cada passo tem um conjunto imagem que será domínio para o próximo passo. Para que se conclua que o procedimento vai parar, trazendo um resultado, cada passo tem que atingir seu conjunto imagem de tal modo que o próximo passo, ao pegar essa imagem como argumento, atinja, também o seu conjunto imagem. Quando o último passo fizer isso, o procedimento parou com sucesso.
Se a análise de um passo mostrar que ele não vai gerar um argumento válido para o próximo passo, o analisador pode concluir que o procedimento, como um todo, não vai parar para o argumento inicial x.

O passo i pode:

1) Gerar uma imagem aceitável.
2) Gerar uma imagem inaceitável.
3) Nunca gerar uma imagem.

O passo (i+1) vai:

1) Aceitar a imagem e transformá-la, podendo, por sua vez, cair num dos passos 1, 2 ou 3, acima.
2) Ficar bloqueado, aguardando um argumento aceitável. Se o passo i não sabe o que o passo (i+1) espera, este caso equivale ao 3. Porém, tratar o argumento inaceitável é torná-lo aceitável, então não consideraremos isso e (i+1) ficará bloqueado quando o passo i gerar um argumento inaceitável.
3) Ficar bloqueado, aguardando um argumento (aceitável).

Como saber, de antemão, se um argumento vai servir para o próximo passo? Será que analisar o programa de baixo para cima resolveria? Simulando uma execução reversa do passo i, ele daria as características de seu argumento para que o passo (i-1) o gere. O problema é que nem todas as transformações tem uma inversa. Então não podemos generalizar.

Ora, o Problema da Parada não especifica quantos passos o programa deve ter. Assim, vamos analisar o caso mais simples em que f(x) só tem 1 passo:

1) Ele vai parar.
2) Ele não vai conseguir gerar uma imagem (não vai parar)
3) Ele não vai parar.

Os casos 2 e 3 são a mesma coisa, o que se resume a:

1) Ele vai parar.
2) Ele não vai parar.

O analisar P ficará:

```
P(f,x)
{
        Se f(x) para, devolva 1
        se não, devolva 0
}
```

O programa f(x) para se x estiver no domínio de f. Mas, dado um x arbitrário, como saber se ele está no domínio de f sem ser através de um teste de f? Sem conhecer f não dá para saber. Conhecendo f, só dá para saber se f parar. Se f não parar, não dá para saber se vai parar algum dia ou que nunca vai parar com aquele x.
Um ser humano pode olhar f e x e decidir se f vai parar, mas, como uma máquina pode fazer isso? A máquina tem que executar f(x) para decidir.
O problema dos computadores é que eles incorporam o programa e são o programa enquanto este roda. Se o programa nunca parar, o computador nunca vai parar também.

Por outro lado, existem alguns problemas que um ser humano, e muito menos um computador, não é capaz de decidir, como a conjectura de Goldbach e o problema de Fermat (ver). No máximo, um ser humano poderia dizer: *desconfio que isso não para*, mas, uma máquina não pode nem isso.
Concluindo que o programa para ou não para para um dado x, não permite ao analisador generalizar e dizer que vai parar ou não vai parar para um dado y ≠ x. Seria necessário testar todo o domínio. Só isso iria fazer o analisar rodar por tanto tempo que poderíamos dizer que ele nunca iria parar. Então tal teste é inviável.

Dado um programa analisador, podemos confundi-lo, o que prova que ele pode ser confundido por um programa qualquer. Assim, não dá para construir um analisador que decide se um programa para ou não.
O truque consiste em construir um programa Q que pega um programa qualquer x e o usa como argumento para uma subrotina que chama o analisador P. Conforme a resposta de P sobre x, Q faz o oposto. Depois basta usar o próprio Q no lugar de x. Isso vai confundir P.

Todo programa tem uma "personalidade". Se ele não tiver nenhuma entrada (zero argumentos), sua personalidade é fixa. O que uma entrada (input) faz é alterar a personalidade do programa. Claro que o programa tem que reconhecer essa entrada (não ignorá-la). Não é a entrada e nem a saída que caracteriza a alteração da personalidade do programa, mas, sim, o tratamento que ele dá à entrada. Um programa pode pegar duas entradas diferentes e gerar a mesma saída, como se estivesse tratando apenas uma das entradas.
Como o analisador P se comportaria com um programa x que tem a personalidade fixa? Ele diria se para ou não. A questão é que é sempre possível construir um programa Q que aceite x como argumento, testá-lo com P, inverter e chamar Q com Q.
E se P mentir?

Voltando à linguagem matemática...
Suponha que alguém diga que construiu uma máquina F que resolve o problema da parada, isto é, F pega a instrução i e o dado x e diz se i é definida em x ou não:

$$F(i,x) = \begin{cases} 1 \; se \; i \; é \; definida \, em \; x \\ 0 \; se \; i \; é \; indefinida \, em \; x \end{cases}$$

Podemos ter no programa, então, uma instrução j definida assim:

$$j(x) = \begin{cases} 1 \; se \; F(x,x) = 0 \\ indefinida \, se \; F(x,x) = 1 \end{cases}$$

Isto é, a instrução *j* é definida se a instrução *x* for indefinida, e é indefinida se a instrução *x* for definida, para o mesmo dado. Nesse caso, o dado que define *j* não define *x*, e vice-versa.
Ora, se *j* é uma instrução no programa, *x* pode, também, ser uma instrução no mesmo programa (afinal de contas, uma instrução é um número binário – natural).

E, mais ainda: pode ser que exista um índice x_0, tal que x_0 é a instrução *j* (o conjunto de instruções de um programa é contável), ou $F(x_0,x) = j(x)$. Pode ser que $x = x_0$, o que resulta:

$$j(x_0) = \begin{cases} 1 \; se \; j(x_0) \; é \; indefinida \\ indefinida \, se \; j(x_0) \; é \; definida \end{cases}$$

Claramente uma contradição: $j(x_0)$ *definida* $\Leftrightarrow$ $j(x_0)$ *indefinida*

O que tivemos foi que a x-ésima instrução (i_x)pode ser a j-ésima instrução, e que ela estará definida no x-ésimo dado (d_x) se não for definida aí, e vice-versa.
A instrução j foi construída com base na máquina F. Então essa instrução é construída de tal modo que ela é indefinida quando F diz que ela é definida, e é definida quando F diz que ela é indefinida.
Se F existe, j pode ser construída. Basta olhar o que F diz e construir j dizendo o contrário. Isto fará F se confundir e se contradizer, pois ela é obrigada a falar "1" quando falou "0", e falar "0" quando falou "1". E isso pode ser feito, qualquer que seja F. Daí, não é possível construir uma tal máquina F.

Veja que se a máquina F parar (tem que, por suposição), ela irá se contradizer. Pode-se ver, porém, que quando ela for analisar j, encontrará a si mesma em j, e, para dizer que j é definida, tem que dizer que j é indefinida; e, para dizer que j é indefinida, tem que dizer que j é definida, e assim por diante, ficando se jogando pra lá e pra cá sem parar, mas, **por suposição**, ela para.
Ao se supor que ela para, então, agora, fica legal. Ela vai se contradizer.
No mundo real, ela não vai parar, e este é o melhor motivo para se afirmar que ainda não temos tecnologia para construir um tal programa analisador.

O processo usado para se chegar à contradição acima é chamado de **diagonalização**, porque os elementos i e x estão na diagonal da matriz formada pelas instruções e dados:

	I_0	**I_1**	**I_2**	**I_3**	**I_4**	**...**	**I_x**	**...**
d_0	i_0d_0	i_1d_0	i_2d_0	i_3d_0	i_4d_0	...	i_xd_0	...
d_1	i_0d_1	i_1d_1	i_2d_1	i_3d_1	i_4d_1	...	i_xd_1	...
d_2	i_0d_2	i_1d_2	i_2d_2	i_3d_2	i_4d_2	...	i_xd_2	...
d_3	i_0d_3	i_1d_3	i_2d_3	i_3d_3	i_4d_3	...	i_xd_3	...
d_4	i_0d_4	i_1d_4	i_2d_4	i_3d_4	i_4d_4	...	i_xd_4	...
...	...	...	...	...	...	...	...	...
d_x	i_0d_x	i_1d_x	i_2d_x	i_3d_x	i_4d_x	...	i_xd_x	...

Onde cada i_pd_q tem valor 0 ou 1, conforme a instrução esteja definida ou não no dado d.

Seja U(k,x) uma função que enumera o conjunto T = {f: N → N | f é total e computável}, isto é, U é uma máquina que enumera funções (outras máquinas!). Então U não pode ser uma função efetivamente computável (calculada por um algoritmo).

Prova:
Claramente U deve ser total. Pela definição de U, uma função f : N → N está em T se, e somente se, para algum k fixo tenhamos f(x) = U(k,x), para qualquer x pertencente a N. Veja que k rotula a função e x é argumento para essa função (veja a máquina M_F anterior!).
Agora suponha que U é computável por algoritmo. Seja a função g : N → N definida como segue, para cada x Є N:

$$g(x) = U(x,x) + 1$$

Como U é computável, também g o é. Além do mais, g é total (definida para todo x). Então, como g é total e computável, g Є T. Consequentemente, deve existir k_0 tal que $g(x) = U(k_0,x)$, para todo x (isto é, U lista g). Mas, isto implica que $U(k_0, k_0) = g(k_0)$ [lembre-se que g é total, então $g(k_0)$ é definido] por substituição e $g(k_0) = U(k_0, k_0)+1$ ==> 0 = 1 !
Assim, a suposição de que U é efetivamente computável é falsa.

Existe uma função U, parcial e computável, que enumera o conjunto de todas as funções computáveis parciais unárias.

Argumentação:
Toda função efetivamente computável deve possuir um algoritmo que pode ser usado para computá-la. Esse algoritmo pertence ao conjunto Σ*, que possui um procedimento que o enumera. Daí, podemos obter uma lista $l_0, l_1, l_2, ...$ de procedimentos. Basta determinar, depois, se algum l_n é um algoritmo.
Definimos a função U por:

$$U(k,x) = \begin{cases} g \text{ se o k - ésimo procedimento } l_k \textit{ eventualmente parar quando tem x como entrada} \\ \textit{Indefinido se } l_k \textit{ não parar quando aplicado a x} \end{cases}$$

Essa função pode ser efetivamente computável por:

Q: READ k,x → CONSTRUA l_k → $l_k(x)$ → DEVOLVA y SE l_k PARAR → HALT

Q para <==> U(k,x) é definido

Agora, devemos mostrar que essa máquina, de fato, enumera o conjunto T de funções unárias parciais efetivamente computáveis.

Seja T’ o conjunto de todas as funções enumeradas por U. Primeiro, U é uma função com dois argumentos efetivamente computável. Assim, para um $k_0 \in N$ fixo, a função $f(x) = U(k_0,x)$ é, também, efetivamente computável, tendo apenas um argumento, ou seja, é uma função unária efetivamente computável. Daí $T' \subseteq T$.

Agora, seja f uma função qualquer em T. Assim, f é efetivamente computável. Daí, f deve possuir um procedimento lk_0 que a computa. Assim, para cada x, f(x) é o resultado de lk_0 (x) se f(x) é definido aí, e lk_0 (x) não para se f(x) for indefinida aí. Mas, isso implica, pela nossa construção de U, que $f(x) = U(k_0,x)$, para qualquer x pertencente a N. Assim, $f \in T$. Daí, $T \subseteq T' \Rightarrow T = T'$

As duas proposições anteriores mostram que existem conjuntos que podem ser enumerados por uma função enquanto um seu subconjunto não possui uma tal função enumerativa!
Por exemplo, o conjunto P dos números primos maiores do que 2 é um subconjunto do conjunto Q dos números ímpares. Contudo, é muito mais fácil enumerar Q do que P.
Na primeira proposição, a função usada é total, de onde obtemos uma contradição. Na segunda, U é uma função parcial e a equação $U(k_0, k_0) = g(k_0) = U(k_0, k_0) + 1$ da primeira pode ser aqui satisfeita pelo caso em que $U(k_0, k_0)$ é indefinido.

Problemas Indecidíveis

Um problema é indecidível se não pudermos dizer *sim* ou *não* para a caracterização desse problema.
Por exemplo, o número 45 é par?
Este problema é decidível porque podemos provar isso:

a = 45/2; b = 2 × a. Se b = 45, então *sim*, senão não.

Um problema que não tem solução não é, necessariamente, indecidível (se soubermos que não há solução, podemos decidi-lo).

Indecidível se refere a uma classe. Para elementos específicos desta classe, o problema pode ser perfeitamente decidível, como o Décimo Problema de Hilbert: Para algumas equações diofantinas, pode-se obter um algoritmo de decisão, mas, para o conjunto geral, não. Este é um problema indecidível.

Equações Diofantinas:

É uma equação polinomial indeterminada (tem conjunto infinito de soluções) em que os valores (a serem descobertos) das variáveis só podem ser inteiros:
Forma geral: ax+by=1, x**n + y**n = z**n

Décimo Problema de Hilbert:

Encontre um algoritmo para determinar se uma dada equação diofantina com coeficientes inteiros tem uma solução inteira. Resolvido (Wey Sun - 1992): Não existe tal algoritmo. Isto implica que o problema agora é decidível, porém, era indecidível em 1982. Mas, sigamos com o resumo.

Para um conjunto de três equações, poderíamos obter um algoritmo (geral) que diria: SSS; outro que diria SSN; outro que diria..., isto é, teríamos 2^3 algoritmos diferentes que teriam como entrada o conjunto de equações, e como saída *sim* ou *não*, dependendo de ele encontrar ou não a solução da equação analisada. Desses 2^3 algoritmos, só um é correto. Qual? Ora, se as três equações não têm solução, o algoritmo correto é o que diz NNN. Quer dizer, se ele diz NNN, ele é correto, mas, para dizer NNN, ele tem que ser o correto!
Se soubermos que a terceira equação é a correta, então estaremos usando o conjunto para testar qual é o algoritmo correto; se soubermos qual é o algoritmo correto, podemos descobrir qual é a equação correta? Desde que possamos provar que ele realmente calcula a equação, sim.
Seria possível construir um único algoritmo que testasse todas as equações?

Estes problemas do tipo sim/não, são chamados de Problemas Recursivamente (por meio de algoritmos) Indecidíveis.

Exemplo:
$x^n + y^n = z^n, (x, y, z > 0 \wedge n > 2) \in N$

Pode-se construir um algoritmo que calcule esta equação (encontre uma solução para ela)?
Este problema é indecidível porque não se sabe se esta equação tem solução.

Complexidade

A análise de um algoritmo nos fornece a *limite superior* para a quantidade de recursos que é suficiente para resolver uma dada tarefa. Para melhorar o algoritmo, o *limite inferior* deveria ser encontrado. O ideal é atingido quando o LS for o menor possível e o LI for o maior possível (e coincidentes), e qualquer outro algoritmo seria menos eficiente.
O limite inferior é o ponto onde o algoritmo para de funcionar se o mínimo possível de recurso for removido. Qualquer recurso adicionado acima do limite inferior se torna um desperdício.
Existem funções que, por mais recursos que acrescentemos nela, não podem ser computadas, isto é, não se consegue obter um algoritmo para calcular uma tal função. Isto seria um caso em que o limite inferior é altíssimo, inatingível. Claro fica, que, abaixo do limite inferior, não se consegue um algoritmo.
Por exemplo, a seguinte função é computável, porém, não se consegue construir o algoritmo que a computa:

$$\pi(n) = \begin{cases} 1, \text{se a expansão decimal de } \pi \text{ tem, no mínimo, n dígitos 5 consecutivos} \\ 0, \text{em caso contrário} \end{cases}$$

i) Se л(n) = 1, então л(n-1) =1, n > 0
ii) Se л= 0, então л(n+1) = 0

Das duas, uma:
ou f_π é a função constante $f_\pi(n) = 1$,
ou existe um k fixo tal que

$$f_\pi(n) = \begin{cases} 1, \ se \ k \le k \\ 0, \ se \ k > k \end{cases}$$

n	1	2	3	...	k	k+1	k+2	...
$f_л$(n)	1	1	1	...	1	0	0	

Qual é k? Não se sabe se há um tal k, ainda.

Predicados

Um predicado θ sobre um conjunto X é uma função θ:X → {0, 1}. Se θ(x) = 1 para x Є X, então, dizemos que x satisfaz θ, ou que θ é verdadeiro em x. Caso contrário, isto é, se θ(x) = 0, então dizemos que x não satisfaz θ, ou que θ é falso em x.

Exemplos:

X = N

$$\theta(x) = \begin{cases} 1, \text{ se } x \text{ é primo} \\ 0, c.c \end{cases}$$

X = {conjunto das equações diofantinas}

$$\theta(x) = \begin{cases} 1, \text{ se } x \text{ tem soluções inteiras} \\ 0, c.c \end{cases}$$

Um predicado θ sobre X é **decidível** se existir um algoritmo F que calcula θ, isto é, se F, com argumento x Є X para e F(x) = θ(x).
F nos dirá se θ(x) é verdadeiro ou falso. Neste caso, dizemos que F é um Processo Efetivo de Decisão para θ.
θ é indecidível, se não for decidível.

Teorema → Seja θ o seguinte predicado sobre o conjunto r* × r* de pares de palavras de r*: (θ: r*× r* → {0, 1}).

i) $\theta(p_t, d_t) = 1$ se p_t for um programa que, executado com teto de dados d_t, para.
ii) $\theta(p_t, d_t) = 0$, se p_t com d_t não para.

Nestas condições, não existe um algoritmo F que calcula o predicado θ (não existe F tal que $F(p_t,d_t) = \theta(p_t,d_t)$. Em outras palavras, θ é indecidível.

Note que, para (p_t,d_t) particulares, podemos obter um F, mas, para a classe toda de (p_t,d_t), isto é, para o caso geral, tal F não é possível de se obter.

Prova:

Suponhamos que F exista, isto é, F calcula θ, ou $F(p_t,d_t) = \theta$. Então, para qualquer par p_t,d_t, $F(p_t,d_t) = 1$ se p_t com d_t para e $F(p_t,d_t) = 0$ se p_t com d_t não para. F pode, então, ser usado para definirmos um novo procedimento C, como se segue:

```
procedimento C(A,B):
    Se F(A,B) = 1, então
        repita x ← 1 até que x = 0
    senão {F(A,B) = 0} devolva 1
```

ou seja, C é uma função definida por

$$C(P,d) = \begin{cases} 1, \text{ se P com d não pára} \\ \text{indefinida (não pára) se P com d pára} \end{cases}$$

Como é suposto que F existe, a função C é computável. Note que F pode listar todos os programas, inclusive C, acima.

Definimos, agora, o seguinte procedimento:

```
procedimento D(A):
        devolva C(A,A)
```

Por extensão, D também é computável e aparecerá na lista dada por F. Também, D é uma função parcial definida por

$$D(P) = \begin{cases} 1, \text{se P não pára} \\ \text{indefinido , se P pára} \end{cases}$$

Pela suposição de que F existe, dizemos que D está na lista.
Que acontece se dermos a D seu próprio texto (isto é possível, pois, D recebe um programa A como dado)?

$$D(D) = \begin{cases} 1 \text{se D não pára} \\ \text{Não Pára,se D pára} \end{cases}$$

Se D para, então D não para; se D não para, então D para, ou seja, D para <==> D não para. Contradição.

Quando F diz que D não para, D para (por definição).

Se D para, então F diz que D não para e, por definição de D, D não para: contradição.
Se D não para, então F diz que D para e, por definição de D, seria o caso em que D para. Outra contradição.

Portanto, a suposição de que F existe implica que D é computável, o que leva a uma situação impossível. Por isso, somos forçados a admitir que F não existe.

Existem dois métodos para se demonstrar a insolvabilidade de um problema:

1) Diagonalização: Quando puder ser feita uma listagem dos objetos através de um algoritmo, o método produz um objeto que não está na lista.
2) Redução: Pegar um problema que se sabe ser insolúvel e mostrar que a solubilidade do problema em questão implicaria na solução do problema, que se saber ser insolúvel.

A prova acima foi por diagonalização.

Redutibilidade

Um predicado θ_1 é redutível a um predicado θ_2 se tivermos um método efetivo para decidir (resolver) θ_1 através de θ_2, desde que nos seja fornecido o valor de θ_2 sempre que for necessário. Ou, um problema é redutível a outro se um método para resolver o segundo produz um método para resolver o primeiro.

- Um predicado $\theta_1: X_1 \rightarrow \{0, 1\}$ é m-redutível a um predicado $\theta_2: X_2 \rightarrow \{0, 1\}$ se, e somente se, existir um algoritmo f que, para qualquer entrada $x_1 \in X_1$, produz uma saída $x_2 \in X_2$, tal que, $\theta_1(x_1) = 1$ se, e somente se, $\theta_2(x_2) = 1$.

(e $\theta_1(x_1) = 0 <==> \theta_2(x_2) = 0$)
$(\forall x \mid x \in \theta_1 \Leftrightarrow f(x) \in \theta_2)$

- Se θ_1 é m-redutível a θ_2, então escrevemos: $\theta_1 \leq m\theta_2$.

Teorema: Se θ_1 for <u>indecidível</u> e $\theta_1 \leq m\theta_2$, então θ_2 é <u>indecidível</u>.

Prova:

Suponha que θ_2 é decidível e $\theta_1 \leq m\theta_2$ através de f. Seja g o algoritmo que calcula θ_2 e seja h o procedimento

procedimento h(x):
devolva g(f(x))

f(x) sempre para. Como θ_2 é decidível, também g sempre para. Daí, h sempre para. Mas, por hipótese, $\theta_1(x) = 1$ se, e somente se, $\theta_2(f(x)) = 1$ e como g calcula θ_2, $\theta_2(f(x)) = 1$ se, e somente se, $g(f(x)) = 1$. Então h calcula θ_1, o que contradiz a hipótese de que θ_1 é indecidível.

- Se $\theta_1 \leq m\theta_2$ e θ_2 é decidível, então θ_1é decidível.

Prova:

Seja $\theta_1 \leq m\theta_2$ via f. Seja g o algoritmo que calcula θ_2.
Defina $h(x) = g(f(x))$. Daí, h(x) calcula θ_1, portanto, θ_1 é decidível.

- θ_1 *decidivel* $\Leftrightarrow \overline{\theta}_1$ *decidível*
- θ_1 *indecidivel* $\Leftrightarrow \overline{\theta}_1$ *indecidível*

$$\theta_1 : X \rightarrow \{0,1\} \qquad \overline{\theta}_1 = \begin{cases} 1, se\, \theta_1(x) = 0 \\ 0, se\, \theta_1(x) = 1 \end{cases}$$

Às vezes é mais fácil fazer a redução com $\overline{\theta}_1$.

Exemplos

- O problema de se determinar se um procedimento é ou não é um algoritmo é indecidível (Problema da Parada Uniforme).
  Vamos reduzir o Problema da Parada (indecidível) a este problema.
  Temos, então, que achar um algoritmo F que, tendo por entrada um programa P_t e seu dado d_t, dê por saída um programa Q_t, de nome Q, tal que Q pare com quaisquer dados se, e somente se, P para com d.
  A saída F será então o texto

  procedimento Q():
  P(d);

  Como se vê, Q() para se, e somente, se, P(d) para.

- Problema da Equivalência – Não existe um algoritmo E_q que decide se dois procedimentos dados, P_1 e P_2, são equivalentes ou não.
  Prova:
  Pode-se ver que

  Eq(Pt, procedimento D(): repita x←1 até que x=0) = 1

  se e somente se P não para com nenhuma entrada. Como se vê no próximo problema abaixo, isso é indecidível, assim este segundo problema é indecidível.

- O problema de determinar se um procedimento não para com nenhuma entrada (antialgoritmo) é indecidível:

$$\theta_1(P_t) = \begin{cases} 1 se \forall x, P(x)\ não\ pára \\ 0 se \exists x_0, P(x_0)\ \ pára \end{cases} \qquad \bar{\theta}_1(P_t) = \begin{cases} 1, se \exists x_0, P(x_o)\ \ pára \\ 0, se \forall x, P(x) não\ pára \end{cases}$$

  procedimento Q(z): P(x) ← Redução de P(d) a $\bar{\theta}_1$

  Então, P(x) para <==> Q para com qualquer entrada.

- O problema de determinar se um programa P, com dado d, executa um dado comando S do programa é indecidível:

  procedimento Q():
  P(d);
  S;

  Vê-se que Q executa S se, e somente se, P(d) para.

O Problema da Parada Revisitado

O Problema da Parada não é efetivamente decidível, isto é, não existe algoritmo F que decide se um programa P parará ou não com dado x, ou seja

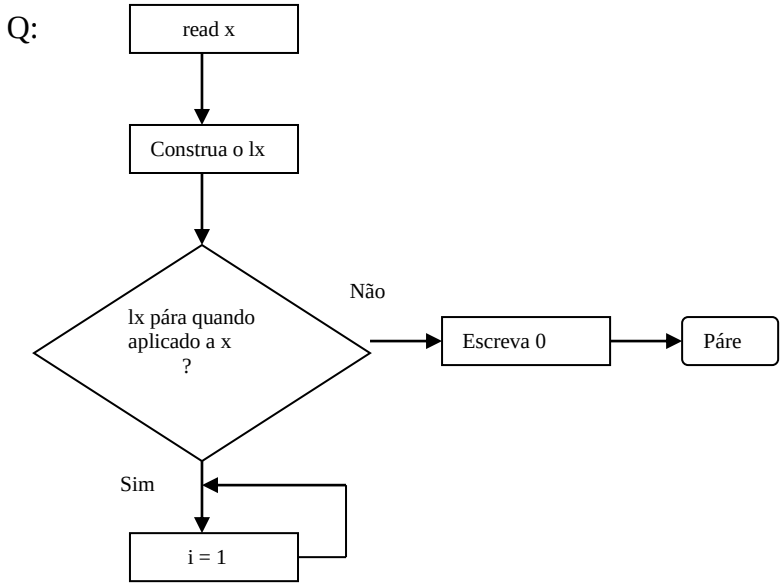
$$F(P,x) = \begin{cases} 1, \ se \ P(x) \ pára \\ 0, \ se \ P(x) \ não \, pára \end{cases}$$

Prova:

Suponha que F exista. Então F pode, certamente, ser aplicado a qualquer procedimento l_k (k = 0, 1, 2, ...).
Agora, defina uma função g, tal que, para todo x Є N,

$$g(x) = \begin{cases} 0, \ se \ l_x \ não termína, com \, x \\ indefinido, \ se \ l_x(x) termína \end{cases}$$

Considere o diagrama abaixo:



Este processo, claramente computa a função g. Como é suposto que F existe, este processo é efetivo e, assim, g é efetivamente computável (existe um procedimento que calcula g, e Q é um tal procedimento). Então, como g é computável por Q, Q deve aparecer na lista $l_0, l_1, l_2, ...$, digamos, na posição k0 (Q = l_{k0}). Considere agora o valor de g(k0), isto é, o resultado de aplicar o procedimento que computa g ao seu próprio índice na lista $l_0, l_1, l_2, ...$ (o que vai acontecer é que Q vai ser aplicado a ele próprio).

Somente dois casos podem ocorrer: ou g(k0) é definido ou é indefinido. Se g(k0) é definido, então Q deve parar (senão não estaria definido!). Mas, isso implica que a resposta a l_{k0} *para quando aplicado a k0* é *Não* (g definido implica g não definido [a resposta de F]). Consequentemente, pela definição de g, isso ocorre só quando g é indefinida. Contradição!

Então, a resposta é NÃO só quando g(x0) é indefinido, e isso concorda com a definição de g. Mas, se g(k0) é indefinido, Q deve entrar no loop i = 1. Mas, esse loop só ocorre se g(k0) é definido (resposta de F). Outra contradição.

Assim, a suposição que o Problema da Parada é efetivamente decidível implica que g é efetivamente computável, o que, por sua vez, leva a uma situação impossível:

g(k0) é definido <==> g(k0) é indefinido

Daí, o Problema da Parada não é efetivamente decidível.

Considere os dois conjuntos A = {x | W_x é infinito} e B = { x | f_x é total}.
A é o conjunto daqueles x que rotulam conjuntos infinitos. B é o conjunto dos x que rotulam funções totais (funções definidas para qualquer elemento de seu domínio).

De uma maneira razoável, cada um destes conjuntos é redutível ao outro, pois, primeiro, assuma que pudéssemos testar f_x total para qualquer x. Então, para testar Wx_0 infinito para um dado x_0:

Use x_0 para obter um x_1 tal que fx_1 liste os membros de Wx_0 sem repetição e tal que fx_1 é total se Wx_0 é infinito, e ver se fx_1 é total.

Similarmente, segundo, assuma que pudéssemos testar Wx infinito para qualquer x. Então, para testar fy_0 total para um dado y_0:

Use y_0 para obter um y_1 tal que $f_{y1}(z) = \begin{cases} 1, \ se \ \ f_{y0}(w) \ converge \ para \ \forall w \le z \\ diverge \ em \ caso \ contrário \end{cases}$ e

ver se Wy_1 é infinito.

Considere, agora, K = {x | f_x(x) converge} e L = {x | Wx é finito}

K é redutível a L.
Para testar x_0 em K: use x_0 para obter x_1, tal que

$f_{x1}(z) = \begin{cases} 1, \ se \ P_{x0}(x_0) \ pára \ em \ z \ passos \\ diverge \ em \ caso \ contrário \end{cases}$ e ver se Wx_1 é finito.

O problema de determinar se o valor da função inteira calculada por um programa P é maior do que 5 para alguma entrada é indecidível.

$$\theta_3 = \begin{cases} 1 se \exists x \mid f(x) > 5 \\ 0, c.c. \end{cases}$$

procedimento P(n):
y, p ← INTERPR(Qt, x, n);
Se p, então devolva 6
senão, devolva 0;

INTERPR executa Q com x durante n segundos (n suficiente!). Se Q para dentro dos n segundos, então p = TRUE e P devolve valor >5, senão, P devolve 0.

O procedimento INTERPReta recebe uma tripla (P, x, n), onde P é o programa, x é o dado para P e n é o número de passos (tempo) dado a P. Como temos uma tripla (P,x,n), para o mesmo par (P,x) teremos n's diferentes. Isto significa que se o programa P, com dado x, não parou em n segundos, ele terá uma nova chance com m > n. Isto é semelhante a um processo num sistema operacional, em que é dado um tempo maior a um processo caso ele não termine no tempo que lhe foi dado anteriormente. O processo é recolocado na fila de execução (round-robin).

Resumo de Redutibilidade

A função total F que calcula um predicado θ_1 não é a mesma *f* que reduz θ_1 a θ_2:

F diz sim ou não para θ_1.
f transforma θ_1 em θ_2.

O predicado θ_1 é decidível se existir F. Se F não existir, θ_1 é indecidível.
Se *f* reduz θ_1 a θ_2 e não existir F para θ_1, então não existe F para θ_2. Se existir F para θ_2, então existe F para θ_1.

Sabendo que θ_1 é indecidível e que θ_2 é decidível por F2, não será possível achar *f* que reduza θ_1 a θ_2.

Se $\exists F_{\theta 1} \Leftrightarrow \exists F_{\bar{\theta}1} = \overline{F}_{\theta 1}$
Se $\nexists F_{\theta 1} \Rightarrow \nexists F_{\bar{\theta}1}$

Se $\theta_1 \leq \theta_2$ por *f* e existe $F\theta_2$, então $F\theta_2$ ($f(x_1)$) = $F\theta_1$, isto é, $F\theta_1$ é a composição de $F\theta_2$ com *f*, $F\theta_2$ (*f*).

Os seguintes problemas são mostrados serem insolúveis pela redução do problema da parada a eles. O problema de

1. Decidir para qualquer x se f_x é uma função constante.
2. Decidir para quaisquer x e y se y ∈ Imgf_x.
3. Decidir para quaisquer x, y e z se $f_x(y) = z$.
4. Decidir para quaisquer x e y se $f_x = f_y$.
5. Decidir para qualquer x se Imgf_x é infinito.

Conjuntos Recursivos e Recursivamente Enumeráveis

Um conjunto S é *recursivo* (efetivamente decidível) se, e somente se, existe um processo efetivo que, quando dado um objeto x, eventualmente responderá *sim* se x Є S e, eventualmente, responderá *não* se x não pertencer a S. Assim, o processo, eventualmente, parará para qualquer entrada x.

Exemplo: O conjunto P = {x | x é par} é recursivo, pois o algoritmo abaixo decide se o valor n pertence a P.

```
procedimento F(n):
        x ← n ÷ 2;
        Se n = 2 × x, então devolva 1
        senão devolva 0;
```

A função enumerativa correspondente de S é $f_S(x) = \begin{cases} 1, \ se \ x \in S \\ 0, \ se \ x \notin S \end{cases}$

Existem $\aleph_0$ conjuntos recursivos.

Um conjunto S é *recursivamente enumerável* (efetivamente enumerável) se $S = \varnothing$ ou existe um processo efetivo que listará todos e só os elementos de S (possivelmente fora de ordem ou com repetições). Assim, se x é um objeto,

a) Se x Є S, o processo, eventualmente, listará x
b) Se $x \notin S$, então x nunca será listado.

Em termos matemáticos: S é recursivamente enumerável se $S = \varnothing$ ou existe *f* computável (existe uma máquina que calcula *f*) tal que S = imagem de *f*, ou seja, se *f* é computável e sobrejetora.

Note que um conjunto A pode ser enumerável, mas não ser recursivamente enumerável, pois pode não existir uma função uma função (máquina) para listar os elementos de A. Recursivamente Enumerável é mais forte que Enumerável.

Se A é recursivo, $\overline{A}$ é recursivo.

$$f_A(x) = \begin{cases} 1, se \ x \in A \\ 0, se \ x \notin A \end{cases} \qquad f_{\overline{A}}(x) = \bar{f}_A(x) = \begin{cases} 0, \ se \ x \in A \\ 1, \ se \ x \notin A \end{cases}$$

Seja U o conjunto universo (N ou Σ*).
Um conjunto $S \subset U$ é recursivamente se, e somente se, $S = \varnothing$ ou *S* é a imagem de uma função efetivamente computável total f : N → U.

Prova:

Se $S = \varnothing$, a prova é trivial (f é indefinida para qualquer valor).

Se $S \neq \varnothing$:

Seja *S* a imagem de *f*. Então, o procedimento abaixo enumera *S*.

```
procedimento p:
        x ← 0;
        1: y = f(x);
        write(y);
        x ← x + 1;
        goto 1;
```

Suponha que o procedimento q listará os elementos de S. Assumimos que q produzirá uma série infinita de saídas, mesmo que S seja finito (por repetição de um mesmo elemento).

Seja f : N → U tal que, para cada x Є N, f(x) seja igual ao (x+1)-ésimo elemento produzido por q. Claramente, f é total e sua imagem é S. Se f for computável, a proposição estará provada:

```
procedimento p(x):
        i ← 0;
        A: Se x = q(i) então {escreva f(x) = i; pare}
        senão {i ← i+1; goto A}
```

O procedimento p, acima, é um processo efetivo e computa f.

Proposição:

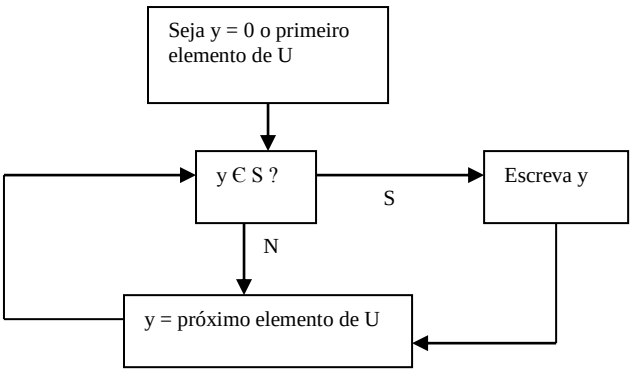
Se S é recursivo, então S é recursivamente enumerável.

Prova:

Se $S = \varnothing$, então *S* é recursivamente enumerável, por definição.

Se $S \neq \varnothing$ e *S* é recursivo, então existe P que decide pertinência em *S*.

procedimento Q



Proposição:

A é recursivo se, e somente se, *A* e $\overline{A}$ são, ambos, recursivamente enumeráveis.

Prova:

Ida: Como A recursivo implica A recursivamente enumerável (provado acima) e como A recursivo implica $\overline{A}$ recursivo, então $\overline{A}$ é recursivamente enumerável.

Volta: Existe P que enumera A e existe Q que enumera $\overline{A}$, o processo abaixo decide se x ∈ A.

procedimento T(x):
n ← 0;
1: Se x = P(n) então escreva 1
senão
se x = Q(n) escreva 0
senão [n ← n + 1; goto 1]

Como A ∪ $\overline{A}$ = U, x aparece em A XOU $\overline{A}$, daí T é um algoritmo (sempre para, porque x acaba aparecendo).

Existem $\aleph_0$ conjuntos recursivamente enumeráveis.

A é recursivamente enumerável não decrescente se f_A é não-decrescente.
A é recursivamente enumerável em ordem crescente se f_A é crescente.

Proposição:

Todo conjunto infinito recursivamente enumerável tem um subconjunto infinito recursivo.

Prova:
Seja A infinito e RE. Seja *f* computável e seja Im*f* = A.
Defina g computável por g(0) = f(0) e g(x+1) = f(y) | f(y) > g(x).

Seja B = Im*g*. Então g enumera B em ordem crescente. Então B é infinito e recursivo.

Desde que B ⊂ A, o teorema está provado.

Se A é RE e A não é recursivo, então $\overline{A}$ não é recursivamente enumerável:

$A = \{(P_t, x) \mid P(x) \textit{ não pára}\}$

$\overline{A} = \{(P_t, x) \mid P(x) \textit{ pára}$

$S = \{(P_t, x) \mid P(x) \textit{ pára}\}$

S é RE se, e somente se, existe P tal que P(x) para se, e somente se, x Є S.

Prova: Seja a_0 Є S.

Ida:

procedimento P(x):

n ← 0;

enquanto x ≠ P(n), faça n ← n + 1;

devolva 1;

Volta:

Seja $h_1 : N \rightarrow N^2$ computável e sobrejetora e $h_2 : N \rightarrow \Sigma^*$ computável e sobrejetora.

procedimento h(n):

p,q ← h_1(n);

x ← h_2(p);

u ← I(P_t,x,q);

Se u, então devolva x, senão devolva a_0;

Aqui não pode ser n, pois, um certo x_0 nunca mais teria outra chance com um tempo maior, e poderia ser que ele parasse para um $n_0 > n$.

Caracterização Alternativa de Recursivamente Enumerável

Proposição:

Se S é recursivamente enumerável, então S é o domínio de uma função computável parcial.

Prova:

Temos um procedimento efetivo P que enumera todos e somente os elementos de S: devemos produzir uma função computável f, tal que f(x) é definida se, e somente se, x Є S.

Seja C um elemento arbitrário de U. Defina f(x) como segue, para cada x Є U.

$$f(x) = \begin{cases} c, se\ x \in S \\ indefinida, se\ x \notin S \end{cases}$$

O diagrama abaixo é um procedimento efetivo para f.

read x;
y = primeiro elemento de S;
1: x = y?
Se SIM, write f(x) = c e pare;
Se Não, y = próximo elemento de S e vá para 1;

Proposição:

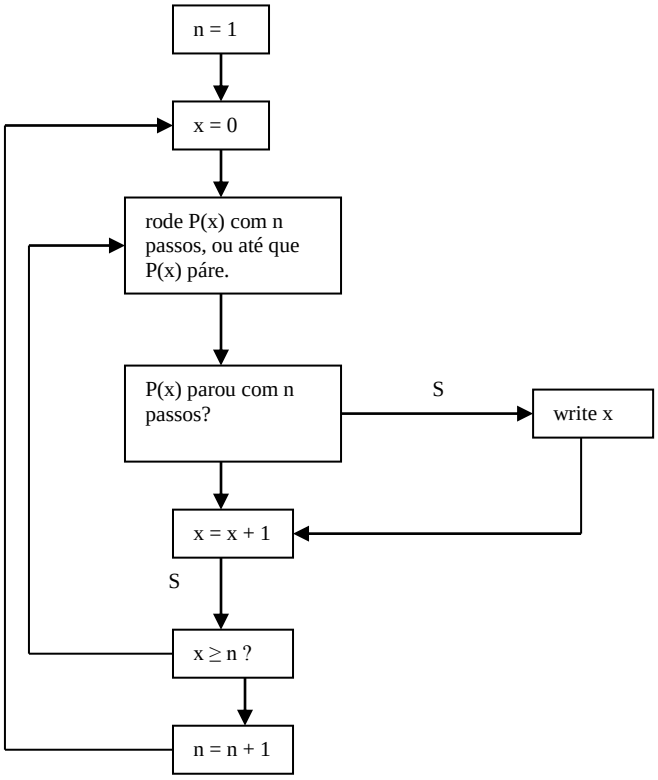
Se S é o domínio de uma função computável parcial, então S é recursivamente enumerável.

Prova:

Suponha que o procedimento P computa a função parcial f e que $S \neq \varnothing$ é o domínio de f. Devemos, então, produzir um procedimento efetivo para enumerar S. Uma tentativa inicial seria:

1. Seja x = primeiro elemento de U.
2. Calcule f(x) usando P.
3. Se f(x) estava definida (x Є S) então escreva x.
4. Em qualquer caso, faça x = próximo elemento de U e vá para o passo 2.

Porém, este procedimento não é efetivo, pois o passo 3 requer o fim de um processo possivelmente infinito, pois P não para se f(x) não está definida. Para sanar esta dificuldade, veja que P deve operar de uma maneira passo a passo. Daí, para computar f(0), tente com 1 passo. Se não der em 1 passo, tente com 2 passos, e assim por diante. O processo é ilustrado pelo fluxograma abaixo, com U = N.



Este procedimento é claramente efetivo, pois cada "box" é individualmente efetivo, e enumera todo o domínio de f. Este procedimento é chamado de *dovetailing*, em que todos os programas P(x) tem pelo menos um passo executado.

A é RE <==> A = Im*f*, *f* computável parcial.
A = Dom*f* <==> Im*g*, *f*, *g* parciais e computáveis.

Proposição: Se A é recursivo, então A e $\overline{A}$ são recursivamente enumeráveis.

De fato, as funções f e $\bar{f}$, definidas por

$$f(x) = \begin{cases} 1, se\ x \in A \\ indef, se\ x \notin A \end{cases} \qquad \bar{f}(x) = \begin{cases} indef, se\ x \in A \\ 1, se\ x \notin A \end{cases}$$

são parcialmente computáveis, e Dom$f$ = A e Dom $\bar{f}$ = $\overline{A}$.

A recíproca é verdadeira: se A e $\overline{A}$ são recursivamente enumeráveis, então A é recursivo.
Então existem duas máquinas M_A e $M_{\overline{A}}$, distintas, tais que M_A lista os elementos de A e $M_{\overline{A}}$ lista os elementos de $\overline{A}$. Assim, para um dado x, colocamos M_A e $M_{\overline{A}}$ para trabalhar e testamos suas saídas com x:

> Se M_A solta x, então x Є A; se $M_{\overline{A}}$ solta x, então X Є A.
> O valor x aparece por M_A ou por $M_{\overline{A}}$ porque $A \cup \overline{A} = U$.

Existem conjunto recursivamente enumeráveis que não são recursivos. Isso implica que se A é recursivamente enumerável, então $\overline{A}$ não é recursivamente enumerável, ou que, se $\overline{A}$ é recursivamente enumerável, A não é recursivamente enumerável.

> Seja K = { i | i Є A_i}, onde A_i = Domf_i
> K é o conjunto dos números naturais que pertencem ao domínio das funções das quais eles são índices. Então f_i está definida para todo x Є A_i e existe x_0 e A_i tal que x_0 = i. A cada A_i corresponde um f_i tal que A_i = Dom f_i e K é o conjunto dos índices i.
>
> Seja x um índice de uma função f computável e y o argumento de f. Daí, existe F, tendo como entrada x, y Є N, tal que F(x,y) = f_x(y), ou seja, F computa f.
> A função g(i) = F(i,i) é parcialmente computável, e
>
> K = { i | i Є A_i} = i | i Є Domf_i } = DomF = Dom$g$
>
> então K é recursivamente enumerável.
> E $\overline{K}$?
>
> Suponhamos que $\overline{K}$ é RE. Então $\overline{K} = \{i \mid i \notin A_i\} = A_p$ para algum p.
> Então i Є $\overline{K}$ se, e somente se, i Є A_p.
> Pela definição de K, i e $\overline{K}$ se, e somente se, i ∉ A_i. Ora, se $\overline{K}$ é RE, existe h_p tal que A_p = Domh_p. Tomando p = i, teremos p Є $\overline{K}$ <==> p ∉ A_p = $\overline{K}$, o que é absurdo. Daí, $\overline{K}$ não é RE e K não é recursivo.

Seja K = {x | $f_x(x)$ converge} = { x | x Є W_x}

W_x = Domínio de f_x, onde f é parcial e computável, sendo W_x, então, recursivamente enumerável.

Prova:

Defina $g(x) = \begin{cases} 1, \text{ se } f_x(x) \text{ converge} \\ \text{diverge, se } f_x(x) \text{ diverge} \end{cases}$

g, claramente, é parcialmente computável e K = Dom*g*. Daí, K é RE. E $\overline{K}$? Assuma que K é recursivo. Então $\overline{K}$ = W_m para algum m ($\overline{K}$ é RE). Daí,

m Є K <= => m Є W_m, por definição de K, mas
m Є $\overline{K}$ <= => m Є W_m, pela escolha de m

Isto é uma contradição, daí, K não pode ser recursivo. Assim, enquanto K é RE, $\overline{K}$ não é.

Teorema 7: Se f é recursiva parcial e A é recursivamente enumerável, então $f^{-1}(A)$ é recursivamente enumerável.

Prova Informal

Liste os membros de A. Ao mesmo tempo, liste, por *dovetailing*, computações de f para todas as entradas possíveis. Escolhas aquelas entradas que tem correspondentes em A.

EXEMPLOS

1. Se A é RE e B é RE, então $A \cap B$ é RE.
Prova:

procedimento $P_{A \cap B}(x)$*:*
$P_A(x)$; $P_B(x)$;

$P_{A \cap B}(x)$ para se, e somente se, x Є A e x Є B.

2. Seja $A \subset N$ infinito e RE em ordem não decrescente, isto é, para i ≤ j => f(i) ≤ f(j), então A é recursivo.
Prova:

procedimento P(x):
m ← -1;
repita n ← n+1 até que x ≤ f(n);
Se x = f(n), então devolva 1
senão devolva 0;

3. A = {P_t | DomP é infinito}, P para para qualquer a Є A.
Prova:
Seja o algoritmo f que, dado P_t, devolve Q_t:

procedimento Q(x):
inicio:
w ← x_0; {x_0 é uma palavra de Σ}*
Enquanto w ≠ x faça
inicio:
P(w);
w ← sucessor(w);
fim
devolva 1;
fim

T ≤ A via f: P_t Є T ==> Q_t Є A
T não RE ==> A não RE ==> A não recursivo.
$\overline{T} \leq \overline{A} \Rightarrow \overline{A}$ não RE.

4. $A \subseteq X$ é RE se, e somente se, existe um programa P que, com entrada x Є X, para se, e somente se, x Є A.
Prova:
"==>"
Como A é RE, ou $A = \varnothing$ *ou* $\exists f : N \rightarrow A$ sobrejetora:
Caso 1: $A = \varnothing$
procedimento P(x):
Enquanto 1 ≠ 0, faça nada;

Caso 2: $A \neq \varnothing$, então f existe

procedimento P(x):
inicio
n ← -1;
repita n ← n + 1 até que f(n) = x;
devolva 0;
fim

P(x) para se, e somente se, x Є A.

"<=="
Suponha P um procedimento tal que P(x) para se, e somente se, x Є A. Suponhamos que $A \neq \varnothing$, ou seja, P(x) para pelo menos para uma entrada. Se $A = \varnothing$, trivial. Com $A \neq \varnothing$ tomamos a_0 Є A (onde a_0 é um elemento qualquer escolhido). Geramos o par (x,y), deixando P(x) ser executado por y segundos. Ou P(x) para e x Є A ou P(x) excede limite de tempo, então devolvemos a_0.

Redutibilidade e Recursividade

$A \leq B \; se \; \exists f \; tal \, que \forall x \in A \Leftrightarrow f(x) \in B$

(Equivalentemente:

$A \leq B se \; A = f^{-1b}(B) ou \; f(A) \subset B \wedge f(\overline{A}) \subset \overline{B} \; ou \; C_A = C_B(f)$

onde *C* é a função característica).

Teorema:

a) $A \leq B \Rightarrow \overline{A} \leq \overline{B}$

b) $A \leq B \wedge B recursivo \Rightarrow A recursivo$

É o mesmo que: se A ≤ B e B é decidível, então A é decidível.

c) $A \leq B \wedge B$ *é recursivamente enumerável* $\Rightarrow A$ *é recursivamente enumerável*

Prova:

a) Imediato.

b) B recursivo implica que existe um algoritmo P_B:

$$P_B(x) = \begin{cases} 1, \; x \in B \\ \\ 0, \; x \notin B \end{cases} = C_B$$

Como f reduz A a B, temos:

procedimento $P_A(x)$*:*

devolva $P_A(f(x))$ = C_B o *f*

c) A ≤ B por *f*, daí A = f^{-1}(B). Então, pelo teorema 7, A é recursivamente enumerável.

Se A é recursivamente enumerável e $\overline{A}$ não é recursivamente enumerável, então $\overline{A} \not\leq A \; e \; A \not\leq \overline{A}$.

Então $K \not\leq \overline{K} \; e \; \overline{K} \not\leq K$

$A \leq B$ não implica geralmente que $A \leq \overline{B}$.

Definição

$A \equiv B \; se \; A \leq B \; e \; B \leq A$

A é equivalente a B se B ≤ A e A ≤ B.

Outra prova de c:

$A \leq B \wedge B$ *recursivamente enumerável* $\Rightarrow A$ *recursivamente enumerável.*

Se B é recursivamente enumerável $\exists P \mid P(x)$ *pára* $\Leftrightarrow x \in B$.

$P(f(x))$ *pára* $\Rightarrow f(x) \in B \Rightarrow x \in A$ ($pois \; A \leq B, x \in A \Leftrightarrow f(x) \in B$!)

$P(f(x))$ *não pára* $\Rightarrow f(x) \notin B \Rightarrow x \notin A$

Corolário: Se A ≤ mB e A não é recursivamente enumerável, então B não é recursivamente enumerável.

Seja K = {P_t Є Σ* | P_t é um programa e P(P_t) para}, K é recursivamente enumerável.
Tentemos reduzi-lo ao Problema da Parada, que é recursivamente enumerável.
Seja B = { P_t | P(d) páre}. "Cavamos" uma função f redutora que faz f(P_t) = (P_t, P_t).

Caso 1:

P_t é um programa $\begin{cases} P(P_t)\ pára \Rightarrow P_t \in K\ e\ (P_t, P_t) \in B \\ P(P_t)\ não\ pára \Rightarrow P_t \notin K\ e\ (P_t, P_t) \notin B \end{cases}$

Caso 2:

P_t não é um programa $\Rightarrow P_t \notin K\ e\ (P_t, P_t) \notin B$

Como houve concordância, é redução e, como B é recursivamente enumerável, também K é recursivamente enumerável.

Qualquer conjunto A recursivamente enumerável é redutível a K.

Seja A recursivamente enumerável. Então existe procedimento P tal que P(x) para se, e somente se, x pertence a A. Portanto:

$$\left.\begin{array}{l} proce\,\text{dim}\,ento\ Q(y): \\ devolva\ \ P(x) \end{array}\right\} Se\ x \in A \Rightarrow P(x)\ pára \therefore Q(y)\ pára\ para\ \forall y \in \Sigma^* \therefore Q(Q_t)\ pára \therefore Q_t \in K$$

$$Se\ x \notin A \Rightarrow P(x)\ não\ pára \Rightarrow Q(y)\ não\ pára\ \forall y \in \Sigma^* \Rightarrow Q(Q_t)\ não\ pára. \therefore Q_t \notin K$$

Como houve concordância dos "==>", é redução. Daí, podemos concluir que A é recursivamente enumerável se, e somente se, A ≤ mK (esta é outra caracterização).

Mais uma Caracterização de Res:

$$A = \{x \in \Sigma^* \mid \exists y \in \Sigma^* \wedge (x, y) \in B\}\ e\ B\ recursivo, então\ A\ é\ RE$$

A é formado por aqueles x que, juntamente com y, ambos em Σ*, formam um par (x,y) que pertence a B.

Prova:

Caso 1: Se $A = \varnothing$, implica que A é RE (por definição)
Caso 2: Se $A \neq \varnothing$ (então deve existir f, ..., computável, sobrejetora).

Seja x_0 Є A e seja h_1 : N → Σ* × Σ* uma função sobrejetora e computável que, dado n Є N, devolve um par (x,y) Є (Σ* × Σ*).

```
procedimento h(n):
        x,y ← h1(n);
        Se P(x,y),  então devolva x      {P decide (x,y) Є B?
        senão, devolva x0
```

h(n) para para qualquer n, devolve x ou devolve x_0.
Se devolve x_0, implica que x_0 pertence a A.
Se devolve x, neste caso P(x,y) é verdadeiro, portanto (x,y) Є B e, portanto, existe y tal que (x,y) Є B.
Portanto, x Є A, portanto h : N → A é **computável**.

h é sobrejetora, pois, seja x Є A, então existe y pertencente a Σ* tal que (x,y) pertence a B.
h_1 é sobrejetora, portanto existe n tal que $h_1(n) = (x,y)$, então h(n) devolve x.
Portanto, A é RE (pela definição).

Prova 2 (mais simples):

```
procedimento busca(x):
inicio:
    y ← λ;
    Enquanto (x,y) ∉ B, faça y ← proximo(y);
    devolva 1
```

busca(x) para se, e somente se, x Є A, daí A é RE.

Recíproca da Caracterização Antesequente

Para $\forall A \subseteq \Sigma^*$ recursivamente enumerável, existe B recursivo, com $B \subseteq \Sigma^* \times \Sigma^*$.
Então A pode ser definido por $A = \{x \in \Sigma^* \mid \exists y \in \Sigma^* \wedge (x,y) \in B\}$.

Prova:

Seja $A \subseteq \Sigma^*$ e seja P tal que P(x) para se, e somente se, x pertence a A (existe, pois A é recursivamente enumerável). Defina B por

$B = \{(x, x_n) \mid P(x) \text{ para em n passos}\}$

onde $x_n = h(n)$ e onde h : N → Σ* bijetora-computável, resulta B recursivo e $A = \{x \in \Sigma^* \mid \exists y \in \Sigma^* \wedge (x,y) \in B\}$.

5. A recursivo implica que existe algoritmo Q tal que $Q(x) = \begin{cases} 1 \ se \ x \in A \\ 0 \ se \ x \notin A \end{cases}$.

Construímos

```
procedimento P(x):
    Se Q(x) = 1, então devolva 1,
    senão repita nada até que 1 = 0;
```

P(x) para se, e somente se, x pertence a A. Daí, A é recursivamente enumerável.

6. T = {Pt | Pt é algoritmo} não é recursivamente enumerável.

7. $B = \{P_t \mid P \equiv Q,\ Q\ dado\}$ = conjunto dos programas equivalentes a um programa Q dado.

Hipóteses:

B não é RE ==> B não é recursivo.

$$\overline{B}: \begin{cases} não\ é\ RE,\ Dom(Q) \neq \varnothing \\ é\ RE,\ Dom(Q) = \varnothing \end{cases}$$

Análise:

Caso 1: Q é algoritmo.

Pode-se mostrar que T ≤ mB por f:

$f(R_t) = P_t \rightarrow$

```
procedimento P(x):
inicio
        R(x);
        devolva Q(x) // Rode Q com x //
fim
```

$Se\ R_t \in T \Rightarrow P \equiv Q \Rightarrow P \in B$

$Se\ R_t \notin T \Rightarrow P \not\equiv Q \Rightarrow P \notin B$

(pois Q para sempre e P não!)

Então, B não é RE, $\overline{B}$ não é RE e B não é recursivo.

Caso 2: $Dom(Q) = \varnothing$

$\overline{S} = \{(R_t, d_t) \mid R(d)\ é\ RE\},\ \exists f : N \rightarrow \overline{S}\ a\lg oritmo\ sobrejetor.$

$Q\ não\ a\lg oritmo \Rightarrow \exists x' \mid Q(x')\ não\ pára.$

S = {(Rt,dt) | R(d) não para}.

Podemos fazer S ≤ mB, por *f(Rt,dt)* = Pt:

```
procedimento P(x):
        Se x = x', então devolva R(d)
        senão devolva Q(x);
```

$Se\ (R_t, d_t) \in S \Rightarrow R(d)\ não\ pára \Rightarrow P(x)\ não\ pára \therefore P \equiv Q$

$Se\ (R_t d_t) \notin S \Rightarrow R(d)\ \ pára \Rightarrow P(x')\ pára \therefore P \not\equiv Q$

$\therefore B\ não é\ RE \Rightarrow B\ não\ é\ recursivo.$

Análise de $\overline{B}$

Caso 1: $Dom(Q) = \emptyset, \exists x' \mid Q(x')\ pára$

$S = \{(R_t, d_t)\ não\ pára\ e\ S \leq m\overline{B}$

f(Rt,dt) = Pt:

procedimento P(x):
inicio.
R(d);
Q(x);
fim.

Se $(R_t,d_t) \notin S \Longrightarrow$ R(d) pára $\Longrightarrow P \equiv Q$ (pois o P dependerá do Q!).

Se $(R_t,d_t) \in S \Longrightarrow$ R(d) não pára $\Longrightarrow$ P(x') não pára (mas deveria) $\Longrightarrow P \not\equiv Q$ (já que Q(x') para).

Portanto, $\bar{B}$ não é recursivamente enumerável.

Caso 2: $Dom(Q) = \emptyset$

procedimento g(n):
$R_t, d_t \leftarrow f(n)$;
Devolva R_t;

g enumera $\bar{S}$ e é sobrejetora de N → $\bar{B}$, portanto, $\bar{B}$ é recursivamente enumerável.

8.

$C = \{P_t \mid P$ calcula função $\equiv 0\}$
$C = \{P_t \mid Dom(P) = \sum^* e\ \forall x, P(x) = 0)$

C não é recursivamente enumerável. $\bar{C}$ não é recursivamente enumerável. C não é recursivo.
Pode-se mostrar que $T \leq mC$:

Seja $f : \sum^* \rightarrow \sum^*$ o algoritmo que, dado Q_t, devolve P_t:

procedimento P(x):
Q(x);
Devolva 0;

Supondo C recursivo:
Se $Q_t \in T \Longrightarrow P_t \in C$ e se $Q_t \notin T \Longrightarrow P_t \notin C$. Portanto, $T \leq mC$.

9.

$D = \{P_t \mid Dom(P)$ é recursivo$\}$

Mostra-se que $T \leq m\bar{D}$, $f(R_t) = P_t$.

Se $R_t \notin \overline{T} \Longrightarrow$ Dom(P) é finito, portanto recursivo ($P_t \notin \overline{D}$).
Se $R_t \in T \Longrightarrow$ Dom(P) = K (o conjunto K visto anteriormente) $\Longrightarrow P_t \in \overline{D}$.

```
Procedimento P(x):
Inicio.
        w ← x0;
        enquanto w ≠ x faça
        inicio
                R(w);
                w ← succ(w);
        fim
        R(w); X(x); devolva 0
Fim.
```

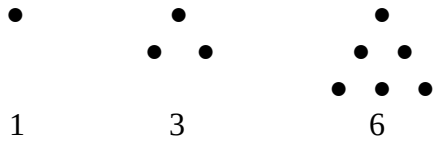
ADENDO I – TEORIA DOS NÚMEROS

A teoria dos números estuda as propriedades das séries de números naturais.
Segue, abaixo, um resumo rápido e restrito de teoria dos números.

Números Figurados

Números figurados são aqueles cujas quantidades de pontos que ele representa podem formar uma figura. Podemos ter:

Números retangulares: a.b
Números quadrados: a.b, com a = b.
Números triangulares:



Todo número triangular é da forma $T_n = \frac{1}{2}n(n+1)$, ou seja, essa fórmula dá o n-ésimo número triangular.

Números pentagonais:

O n-ésimo pentagonal é dado por $P_n = \frac{1}{2}(3n^2 - n)$.

Números k-agonais:

O n-ésimo número k-agonal é dado por $\frac{1}{2}k(n^2 - n) - n^2 + 2n$.

Números Gerados

Todo número *c* é produto de dois fatores, ou seja, $c = a.b$, com $c \neq 0$ e *b* e *a* são os divisores de *c*. Ambos, *a* e *b* não podem exceder $\sqrt{c}$ ao mesmo tempo. Qualquer divisor de *a* ou de *b* também dividirá *c*.

O maior resto *r* de uma divisão de *a* por *b* é igual a $b - 1$.

$a = qb + r \Longrightarrow a/b = q + r/b$ (dividindo ambos os lados por b). Daí, concluir-se que $r/b < 1$ e $q = [a/b]$.

Todo número n pode ser escrito em uma das formas:

$$3k, 3k \pm 1; 5k, 5k \pm 1, 5k \pm 2; 4k, 4k + 2, 4k \pm 1$$

Qualquer número é igual a 2q ou 2q+1. Quadrando, temos $4q^2$ e $4q^2+4q+1$. Daí, todo número quadrado é divisível por 4 ou deixa resto 1.

O quadrado de um número ímpar é da forma 8q+1. Prove com 4k±1, acima.

Qualquer número n pode ser escrito na forma $n = 10a + b$, com $0 \leq b \leq 9$.

Os números da forma y^x só são divisíveis por números da forma $y^{x'}$, onde $x' \leq x$.

Qualquer potência de 2, 2^n, tem n+1 divisores.

Qualquer potência y^x tem x + 1 divisores.

Números Primos

- Os únicos divisores de um número primo são o próprio número e o 1.
- Os números primos não são números retangulares. Sim, pois, a = 1 ou a = p, ou b = 1 ou b = p, sendo que a.b não gera um retângulo e, sim, uma linha.
- O menor divisor de um número c > 1 é primo.
- Qualquer inteiro maior que 1 é primo ou é produto de primos:
  $60 = 2^2 * 3^1 * 5^1$
  60 tem (2+1)(1+1)(1+1) = 12 divisores. Pode-se generalizar isso.

Primos de Marsènne

Um primo de Marsènne tem a forma $M_p = 2^p - 1$, onde p é primo.

[Note que nem todo número dessa forma é primo e que nem todo número primo é dessa forma].

3, 7, 31, 127 são primos de Marsènne.

Primos de Fermat

Tem a forma $2^{2^n} + 1$

Conjectura de Goldbach

Todo número par ≥ 6 é soma de dois primos ímpares.

(A conjectura nunca foi provada!)

Máximo Divisor Comum e Mínimo Múltiplo Comum

MDC(x,y) é o maior valor que divide x e y simultaneamente (divisão inteira). Nota-se, facilmente, que o mdc nunca é maior que o menor de x e y.

MMC(x,y) é o menor valor entre a.x e b.y, sendo que a.x = b.y, e a e b são valores quaisquer.

Propriedades:

MDC(a,b)*MMC(a,b) = a*b
MDC(a,b,c) = MDC(MDC(a,b),c)
MMC(a,b,c) = MMC(MMC(a,b),c)

Pode-se generalizar para qualquer número de argumentos.

Números Perfeitos

São aqueles cuja soma de seus divisores, não incluindo o próprio número (mas, incluindo o 1), dá o próprio número como resultado.
6, 28, 496, 8128 são perfeitos. Não se sabe se o conjunto dos números perfeitos é infinito ou não.
Todo número perfeito é da forma $2^{p-1}(2^p - 1)$, onde p é primo. Note que $2^p - 1$ é um primo de Marsènne:

$6 = 2(2^2 - 1)$
$28 = 2^2(2^3 - 1)$
$496 = 2^4(2^5 - 1)$
$8128 = 2^6(2^7 - 1)$

Veja que todo número perfeito é da forma $2^{p-1}(2^p - 1)$, mas, nem todo número da forma $2^{p-1}(2^p - 1)$ é perfeito, portanto, ela não é uma fórmula fechada para gerar todos os números perfeitos. É por isso que não se sabe se o conjunto de perfeitos é infinito ou não.

Seja S_d a soma dos divisores do número n:

Números Abundantes: $S_d > n$.
Números Perfeitos: $S_d = n$.
Números Deficientes: $S_d < n$.

O primeiro número para abundante é o 12.
O primeiro número perfeito é o 6.
O primeiro ímpar abundante é 945.
Claramente, todo primo é deficiente.

$2^{n-1} \times (2^n - 1)$

O primeiro fator tem n divisores e a soma de seus (n – 1) divisores (excluiu-se o próprio número) é igual a $2^{n-1} - 1$.
Se o segundo fator for um número primo, a soma de seus divisores (excluso o próprio número) será igual a 1. Daí, esses dois fatores produzirão um número m cuja soma de seus divisores [note que todos os divisores do primeiro fator e todos os do segundo são divisores do produto também], excluindo m, será igual a (m – 1) + 1 = m, e m será um número perfeito.

Números Amigos

Seja $S_d(m)$ e $S_d(n)$ as somas dos divisores de m e n, respectivamente. Os números m e n são *amigos* se $S_d(m) = n$ e $S_d(n) = m$.

220 e 284 são os primeiros números amigos.

Se, para algum n, dois termos sucessivos P_{n-1} e P_n são ambos primos, examina-se o número $q_n = 9*^{2n-1} - 1$. Se q_n for primo, então $M = 2^n.P_{n-1}.P_n$ e $N = 2^n.q_n$ são amigos.

Conjectura de Fermat

Não existem inteiros x, y, z > 0 e n > 2 tal que $x^n + y^n = z^n$

Assim como a Conjectura de Goldbach, a de Fermat também não foi provada. É verdadeira para valores específicos de n. Não se conseguiu ainda a prova para n igual 3.

Vamos, nós aqui, conjecturar sobre esta conjectura para n = 3 (cubos).

Primeiro, uma definição particular:

c é um cubo real se c = m × n, sendo m e n cubos quaisquer.

Se $m = p^3$, sendo p primo e n um cubo qualquer, não há como m ser gerado pela multiplicação de dois cubos (pelo menos um deles maior que 1).

O menor cubo maior que 1 que existe é 8. Então 8 é o gerador inicial. O próximo cubo que existe é 27. O 27 é gerado por um primo ao cubo (3 ao cubo). O 27 não pode ser gerado por 8 × n, onde n é um cubo cuja raiz cúbica é inteira. Se pudesse, teríamos que $8 \times n = 27 \Rightarrow 2^3 \times r^3 = 3^3 \Rightarrow 2 \times r = 3 \Rightarrow r = \frac{3}{2}$. Como $n = \frac{27}{8}$ não é inteiro, ele não existe. Então, 27 não é um cubo real. Outros cubos não reais: 125, 343.
Então, todo cubo não real é gerado por um primo. A raiz cúbica desse cubo nunca será inteira.
Um cubo real é um cubo que pode ser gerado pelo acúmulo multiplicativo de outro cubo: CuboX × CuboY = CuboZ.
A adição de dois cubos não gera um cubo, pois, a adição mínima de cubos para gerar um cubo, necessita de 8 cubos.

Se a adição de n cubos (n>7) gera um cubo, é possível criar 2 grupos com os n cubos de tal maneira que cada grupo seja um cubo? Respondendo isso, você decide sobre Fermat.

Fermat diz: A + B = C, onde A, B e C são cubos inteiros.

A + B é o agrupamento que queremos, mas, não sabemos quantos termos tem em A e nem quantos termos tem em B. Sabemos, porém, que qualquer elemento de A é igual a qualquer elemento de B. É o mesmo elemento X que aparece em A e em B, n vezes nos dois. Assim, A + B = nX. X não é um cubo, mas é a n-ésima parte de um cubo.

Se X é um cubo e Y é um cubo, então XY é um cubo, mas, X/Y pode não ser um cubo. Então, dado um cubo X > Y, X/Y será um cubo apenas se, ao se retirar Y de X n vezes, X fica nulo. Pela linha acima, então, implica que n é um cubo, e X = nY. Acumulando-se Y n vezes, teremos um cubo. Separando-se Y em dois grupos, o maior grupo possível terá n/2 Y's. Sendo n um cubo, n/2 não é um cubo. Assim, o primeiro grupo deverá ter menos que n/2 elementos Y e o segundo grupo mais que n/2 elementos Y, ou seja, os grupos têm que ser complementares em relação a Y.

O que é A + B e o que é A × B?

A é uma quantidade; B é outra quantidade; A+B é a quantidade A continuada na quantidade B (ao se contar), como se B fosse um outro A e B + A é a quantidade B continuada na quantidade A, como se A fosse B. Nesse último sentido, podemos escrever A + (A) e B + (B).

"A × B é A + (B), onde B se acumula (A-1) vezes e o primeiro A é igual a B: A + (B) = B + (B+B+B+...+B) × (A-1).
B × A = B + (A) = A + (A+A+A+...+A) × (B-1)"

m^{3n} é cubo para qualquer n > 0.
m^{n} não é cubo se m não é cubo e n não é múltiplo de 3

Abaixo, C significa “cubo” e NC significa “não cubo”.

C/C = C ou NC ==> C = C × C ou C = C × NC
C/NC = NC ou ? ==> C = NC × NC ou C = NC × C (então, NC)
NC/C = NC ou ? ==> NC = C × NC ou NC = C × C (então, C)
NC/NC = NC ou C ==> NC =NC × NC ou NC = NC × C
C × C = C
C × NC = NC
NC × NC= cubo (2 x 32) e não cubo (2 x 15)

Se A é cubo e B é cubo, (A-1) não é cubo e B menos uma parte dele não é cubo. Como (B+B+B+...+B)(A-1) não pode ser um cubo, A + B não gera um cubo!!!

Como Fermat sentiu preguiça, também estou sentindo. Esta deve ser mais uma das milhares de provas falsas.

Números Congruentes

O número *a* é congruente ao número *b* se $a - b$ é divisível por um número *m* dado e diz-se que *a* é congruente a *b* módulo *m*.

Notação: $a \equiv b(mod\ m)$

O número *a* será congruente a *b* se *b* puder ser escrito assim $b = a +$ km, para um k qualquer.
O número *a* pode ser uma expressão: expressão ≡ n (mod m).

Números Computáveis

O número real r, $0 \leq r \leq 1$, é computável se, e somente se, existe uma função recursiva total f_r cujo valor com argumento i é o i-ésimo dígito após o ponto decimal na representação de r.
Isto é, se $r = .d_0d_1d_2...$, então, $f_r(0) = d_0$, $f_r(1) = d_1$, etc. A função f_r é chamada de *gerador de r*.

- Os racionais em [0, 1) são computáveis.
- Existem reais não computáveis.
  Prova:
  O número cujo n-ésimo dígito é 1 se fn é definido em n, e 0 em caso contrário, possivelmente pode não ser computável, pois, se fosse, qualquer gerador seria uma função característica para o conjunto P abaixo.
  P = { x | fx é definida em x}

  Mas, sabemos que esse conjunto não possui uma função característica, isto é, não é recursivo. Teríamos uma contradição.

- Existem irracionais computáveis
  Exemplo:
  $\sqrt{2} - 1\ está\ em\ [0,1)$ e é irracional.
  A função recursiva abaixo calcula a expressão acima.

```
Integer procedure oxen(d); value d; integer d;
begin
        integer temp, i;
        if d = 0 then oxen := 4
        else
         begin
                temp := 1*10^(d+1);
                for i := 0 step 1 until d – 1
                do temp := temp + oxen(i)*10^(d-i);
                for i:=0 step 1 until 10
                do if(temp+i)^2 > 2*10^(2*(d+1))
                    then goto exit
                exit: oxen := i – 1;
        end
end
```

Multiplicação de Matrizes

Algoritmo de Strassen para matrizes 2 × 2:

```
function prodmat2 (x, y): { X × Y = Z }
begin.
        P1 := (x11 + x22) * (y11 + y22);  /* Notação Pascal */
        P2 := (x21 + x22) * y11;
        P3 := x11 * (y12 - y22);
        P4 := x22 * (y21 - y11);
        P5 := (x11 + x12) * y22;
        P6 := (x21 - x11) * (y11 + y12);
        P7 := (x12 - x22) * (y21 + y22);

        z11 := P1 + P3 – P2 + P7;
        z12 := P3 + P5;
        z21 := P2 + P4;
        z22 := P1 + P3 – P2 + P6;

        return z;
End.
```

Para qualquer matriz de dimensão 2^k, basta aplicar recursivamente, dividindo-a em blocos. Para matrizes de dimensão $\neq 2^k$, basta completá-las com zeros para a potência de 2 mais próxima.

Multiplicação Russa

87 × 59:

87	59
43	118
21	236
10	472
5	944
2	1888
1	3776

O primeiro número é divido por 2 (divisão inteira), até chegar a 1, enquanto o segundo número é multiplicado por 2.
Elimine da segunda coluna aquele número que corresponda a um número par na primeira coluna. Adicione o que restou da segunda coluna para obter o resultado.

Bases

Um número a numa base b qualquer é representado na base 10 por:

$$a = a_n b^n + a_{n-1} b^{n-1} + \ldots + a_2 b^2 + a_1 b^1 + a_0 b^0$$

onde *a* são os dígitos que compõem o número na base b.
Veja isso:

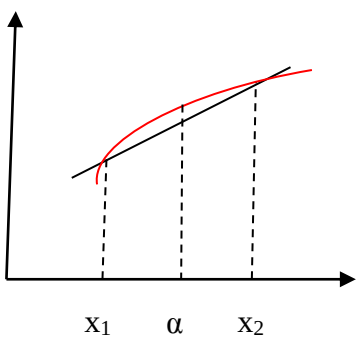
$$\left(\frac{1}{3}\right)_{10} = 0.3333\ldots, = \left(\frac{1}{3}\right)_{6} = 0.2, \left(\frac{1}{3}\right)_{12} = 0.4$$

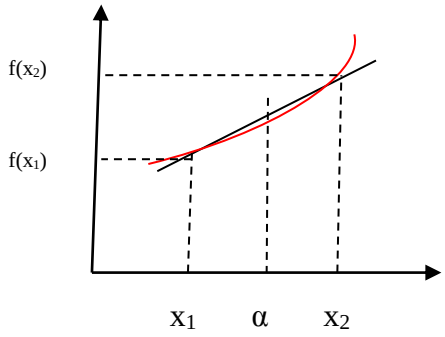
Uma divisão que é infinita numa base pode se tornar finita em outra base. Claro que não se trata da divisão do mesmo número!!!

Uma do dia a dia da Matemática: $\log_b x = \frac{\log_{10} x}{\log_{10} b}$.

ADENDO II – FUNÇÕES CONVEXAS

Uma função real f(x) definida num conjunto convexo (vide) D de um espaço linear de R é chamada de *função convexa* se, para todo x, y de D e $0 \leq \alpha \leq 1$, temos $f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y)$.





A função f(x) é chamada *estritamente convexa* se, em vez de ≤ tivermos <, exceto quando $\alpha = 0$ ou $\alpha = 1$.
Se –g(x) é convexa (estritamente convexa), a função +g(x) é chamada de *função côncava* (estritamente côncava).
Algumas vezes, a condição para uma função ser convez é relaxada tal que a desigualdade acima é assumida somente para $\alpha = \frac{1}{2}$.
Contudo, se D é espaço linear topológico e f é contínua, então a condição relaxada implica a condição original. Consideramos, principalmente, o caso onde D é um intervalo na linha real. Neste caso, uma função convexa f(x), no sentido mais fraco (condição relaxada), é contínua no interior do intervalo se f(x) é mensurável ou limitada superiormente num conjunto positivamente mensurável.

Passagem do conteúdo manuscrito para conteúdo digital concluído em 2 de fevereiro de 2012, às 17:05.

Este resumo foi obtido a partir das seguintes obras:

1. Aspectos Teóricos da Computação – C. Lucchesi, Thomasz Kowaltowski, Imre Simon, Istvan Simon, Janos Simon.
2. Fundamentals of Computing Science – Kurt Maly/Allen R. Hanson
3. Set Theory – Seymour Lipschutz
4. Theory of Functions of Real Variables – I. Natanson
5. Computability Theory – Neil D. Jones
6. Computability and Logic – George Boulos/Richard Jeffrey
7. Theory of Recursive Functions and Effective Computability – Hartley Rogers Jr.
8. Introdução à Teoria da Computação – Roberto Lins de Carvalho
9. Number Theory and Its History – Oystein Ore
10. Notas de Aula no curso de Ciência da Computação da UNICAMP, turma 1977

Os dois melhores livros da lista são o do Seymour Lipschutz (não se se por causa dele ou por causa do esquema da Coleção Schaum, que sempre publicou livros bastante didáticos) e o do Hartley Rogers Jr.
O que achei mais difícil de absorver foi o primeiro da lista, que era o adotado no curso da Unicamp. Seus autores realmente sabiam do que estavam falando, só não ficou bem explicado. Dá parece sentir “buracos” no livro que, se preenchidos, o tornariam mais compreensível.

Campinas, SP, Julho de 1982 (versão escrita manualmente e estendida agora).