1. Fundamentos

1.1. Como raciocinamos? Lógica proposicional

A lógica é a base de todo o raciocínio. Portanto se quisermos estudar e fazer matemática precisamos de dominar os princípios básicos da lógica. Citando *Language, Proof and Logic* (de Jon Barwise e John Etchemendy):

> "(...) all rational inquiry depends on logic, on the ability of people to reason correctly most of the time."

> "(...) there is an overwhelming intuition that the laws of logic are somehow more irrefutable than the laws of the land, or even the laws of physics."

Porque deve um estudante de Informática estudar lógica? Porque precisa de dominar ferramentas lógicas que lhe permitam argumentar se um problema pode ou não ser resolvido num computador, traduzir proposições lógicas da linguagem comum em diversas linguagens computacionais, argumentar se um programa está correcto e se é eficiente. Os computadores baseiam-se em mecanismos lógicos e são programados de modo lógico. Os informáticos devem ser capazes de compreender e aplicar novas ideias e técnicas de programação, muitas das quais requerem conhecimento dos aspectos formais da lógica.

Todos nós raciocinamos enunciando factos e tirando conclusões baseadas nesses factos. O início de uma conclusão é habitualmente indicada por uma palavra como

Então, Logo, Portanto, Consequentemente, ...

Para chegarmos a uma conclusão aplicamos uma *regra de inferência* (ou *regra de dedução*). A mais comum é a chamada regra *modus ponens* (modo que afirma): sendo A e B afirmações, se A e "se A então B" são ambas verdadeiras, então podemos concluir que B é verdadeira.

`[Como aprendeu a regra modus ponens em criança?]`

Outra regra muito comum é a *modus tollens* (modo que nega): sendo A e B afirmações, se "se A então B" é verdadeira e B é falsa, então podemos concluir que A é falsa.

Por exemplo:

- Se ele foi a Coimbra então visitou a Universidade de Coimbra.
- Ele não visitou a Universidade de Coimbra.
- Logo não foi a Coimbra.

`[Como aprendeu a regra modus tollens em criança?]`

Quando tiramos uma conclusão que não decorre dos factos estabelecidos previamente, o raciocínio diz-se *non sequitur* (que não segue).

`[Pense num exemplo de non sequitur que já tenha observado.]`

Neste primeiro capítulo começaremos por estudar um pouco de lógica. Algumas definições de *lógica* que podemos encontrar nos dicionários:

- Estudo dos princípios do raciocínio, particularmente da estrutura das afirmações e proposições e dos métodos de determinação da sua validade.
- Sistema de raciocínio.
- Raciocínio válido.

Um *cálculo* é uma linguagem de expressões, onde cada expressão tem um valor lógico e há regras para transformar uma expressão noutra com o mesmo valor. Aqui estudaremos um pouco do cálculo proposicional. O *cálculo proposicional* é a linguagem das *proposições*. Uma *proposição* é uma expressão da qual faz sentido dizer que é verdadeira ou que é falsa. Cada proposição tem um e um só valor lógico, entre dois possíveis: V (verdadeiro) ou F (falso).

Exemplo. "Coimbra é uma cidade portuguesa" é uma proposição com valor lógico verdadeiro. Mas atribuir um valor lógico à afirmação "Hoje está um belo dia!" já não faz sentido, pois trata-se duma expressão subjectiva que exprime um sentimento de alguém, não de uma afirmação objectiva.

Lógica e operações-bit: Os computadores representam informação por meio de bits. Um bit tem dois valores possíveis, 0 e 1. Um bit pode ser usado para representar os valores de verdade F e V, 0 representa F e 1 representa V. Há assim uma relação evidente entre a lógica e o sistema de funcionamento dos computadores.

O cálculo proposicional (tal como outros tipos de lógica) que vamos estudar pressupõe os seguintes princípios:

Princípio da não contradição: *Uma proposição não pode ser verdadeira e falsa ao mesmo tempo.*

Princípio do terceiro excluído: *Uma proposição é verdadeira ou falsa.*

A afirmação "Os alunos de Estruturas Discretas são de Engenharia Informática ou de Comunicações e Multimédia" pode decompor-se em duas afirmações: "Os alunos de Estruturas Discretas são de Engenharia Informática" e "Os alunos de Estruturas Discretas são de Comunicações e Multimédia". Estas duas últimas afirmações já não se podem decompor mais. Dizemos então que a proposição "Os alunos de Estruturas Discretas são de Engenharia Informática ou de Comunicações e Multimédia" é *composta* e as afirmações "Os alunos de Estruturas Discretas são de Engenharia Informática" e "Os alunos de Estruturas Discretas são de Comunicações e Multimédia" são atómicas. As proposições compostas são construídas a partir de proposições atómicas ligando-as por *conectivos* (ou *operadores*). No exemplo anterior, esse conectivo é "ou". Se denotarmos por p a proposição "Os alunos de Estruturas Discretas são de Engenharia Informática" e por q a proposição "Os alunos de Estruturas Discretas são de Comunicações e Multimédia" e usarmos o símbolo $\vee$ para representar "ou", a afirmação "Os alunos de Estruturas Discretas são de Engenharia Informática ou de Comunicações e Multimédia" escreve-se simplesmente $p \vee q$. Temos assim a operação $\vee$ de *disjunção*.

A afirmação "Ele não visitou a Universidade de Coimbra" é a negação de "Ele visitou a Universidade de Coimbra". Se denotarmos por p esta última proposição e usarmos o símbolo $\neg$ para representar a operação de negação, a afirmação "Ele não visitou a Universidade de Coimbra" escreve-se $\neg p$.

A seguinte tabela contém uma lista destes e doutros conectivos lógicos[1]:

negação	$\neg p$ (não p)
conjunção	$p \wedge q$ (p e q)
disjunção	$p \vee q$ (p ou q)
implicação	$p \rightarrow q$ (se p então q; p só se q; p é condição suficiente para que q; q é condição necessária para que p)
equivalência (formal)	$p \leftrightarrow q$ (p é equivalente a q)
disjunção exclusiva	$p \dot{\vee} q$ (ou p ou q)

As proposições são representadas por fórmulas chamadas *fórmulas bem formadas* que são construídas a partir de um alfabeto constituído por:

- Símbolos de verdade: V e F.
- Variáveis proposicionais: letras do alfabeto $p, q, r, \ldots$
- Conectivos (operadores):

  $\neg$ ("não", negação)

  $\wedge$ ("e", conjunção)

  $\vee$ ("ou", disjunção)

  $\rightarrow$ ("implica", implicação).
- Símbolos de parênteses: (,).

Uma *fórmula bem formada* (abreviadamente, fbf) fica definida da seguinte forma:

- V e F são fbf's; toda a variável proposicional é uma fbf.
- Se A e B são fbf's, as seguintes são também fbf's: $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$, (A).
- Toda a fbf é construída por aplicação sucessiva das regras anteriores.

Exemplo. A expressão $p \neg q$ não é uma fbf. Mas cada uma das seguintes expressões é uma fbf: $p \wedge q \rightarrow r$, $(p \wedge q) \rightarrow r$, $p \wedge (q \rightarrow r)$.

Os parênteses funcionam como símbolos auxiliares que indicam como é formada a fbf. Para evitar um uso excessivo de parênteses e simplificar a escrita das expressões lógicas convenciona-

[1] Em lógica é habitual designar estes conectivos por *conectivos booleanos*, em homenagem ao lógico britânico George Boole (1815-1864), que estudou as leis do pensamento usando métodos matemáticos em [*An investigation into the Laws of Thought*, 1854].

-se que as operações lógicas são consideradas pela seguinte ordem de prioridade: $\neg$, $\wedge$, $\vee$, $\rightarrow$. Convenciona-se ainda que na presença de uma só das três últimas operações, na ausência de parênteses as operações são realizadas da esquerda para a direita.

Exemplos.

$$\begin{array}{rcl} \neg p \wedge q & \text{significa} & (\neg p) \wedge q \\ p \vee q \wedge r & \text{significa} & p \vee (q \wedge r) \\ p \wedge q \rightarrow r & \text{significa} & (p \wedge q) \rightarrow r \\ p \rightarrow q \rightarrow r & \text{significa} & (p \rightarrow q) \rightarrow r. \end{array}$$

Relativamente a uma dada linguagem lógica podemos sempre estudar dois aspectos: a sintaxe e a semântica. A sintaxe diz respeito às regras de formação das expressões lógicas a utilizar, ou seja, as fórmulas bem formadas. Em cima, acabámos de descrever a sintaxe do cálculo proposicional.

A semântica estuda o significado das expressões.

$$\text{Linguagem (conjunto de símbolos)} \begin{cases} \text{sintaxe (fórmulas bem formadas)} \\ \\ \text{semântica (significado).} \end{cases}$$

Quanto à semântica, dada uma fbf, interpretando cada uma das suas variáveis proposicionais com os valores lógicos V ou F, é possível dar um significado à fórmula através da interpretação dos conectivos lógicos dada pelas respectivas *tabelas de verdade*. Cada conectivo tem uma tabela de verdade (que vai ao encontro da forma corrente do significado das operações "não", "e", "ou", etc.). A tabela de verdade faz corresponder aos possíveis valores lógicos das variáveis o correspondente valor lógico da operação[2]:

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$
V	V	F	V	V	V
V	F	F	F	V	F
F	V	V	F	V	V
F	F	V	F	F	V

Em conclusão:

- O significado de V é verdade e o de F é falso.
- O significado de qualquer outra fbf é dado pela sua tabela de verdade.

Exemplo. $\neg p \wedge q$ corresponde a $(\neg p) \wedge q$, cuja tabela de verdade é:

[2]Nas aulas teórico-práticas usaremos o software `Boole` para nos ajudar a escrever tabelas de verdade. Consulte o apêndice Usando Boole.

p	q	$\neg p$	$(\neg p) \wedge q$
V	V	F	F
V	F	F	F
F	V	V	V
F	F	V	F

É claro que a cada fbf corresponde uma e uma só tabela de verdade.

Uma fbf diz-se uma *tautologia* se for verdadeira para todos os possíveis valores lógicos das suas variáveis proposicionais. Uma fbf diz-se uma *contradição* se for falsa para todos os possíveis valores lógicos das suas variáveis proposicionais. Uma fbf diz-se uma *contingência* se não for uma tautologia nem uma contradição.

Exemplos. Suponhamos que queremos averiguar se $p \rightarrow p \vee q$ é ou não uma tautologia. Para isso basta construir a respectiva tabela de verdade:

p	q	$p \vee q$	$p \rightarrow p \vee q$
V	V	V	V
V	F	V	V
F	V	V	V
F	F	F	V

Como para quaisquer valores de p e q toma sempre o valor de verdade, concluímos que é uma tautologia.

Mais exemplos: $p \vee \neg p$ é uma tautologia, $p \wedge \neg p$ é uma contradição e $p \rightarrow q$ é uma contingência.

Duas fbf's dizem-se *(logicamente) equivalentes* se tiverem o mesmo significado, isto é, a mesma tabela de verdade. Para indicar que duas fbf's A e B são equivalentes, escrevemos $A \equiv B$. Em vez do símbolo $\equiv$ também se costuma usar $\Leftrightarrow$. Note que dizer que A e B são logicamente equivalentes é o mesmo que dizer que as fórmulas $(A \rightarrow B)$ e $(B \rightarrow A)$ são tautologias (Prova: $A \equiv B$ sse A e B têm os mesmos valores de verdade sse $(A \rightarrow B)$ e $(B \rightarrow A)$ são tautologias).

Exemplos. As seguintes equivalências básicas são de fácil verificação e são fundamentais no cálculo proposicional:

$p \vee \neg p \equiv \mathsf{V}$	Lei do terceiro excluído
$p \wedge \neg p \equiv \mathsf{F}$	Lei da contradição
$p \wedge \mathsf{V} \equiv p$	Leis da identidade
$p \vee \mathsf{F} \equiv p$	
$p \vee \mathsf{V} \equiv \mathsf{V}$	Leis do elemento dominante
$p \wedge \mathsf{F} \equiv \mathsf{F}$	
$p \vee p \equiv p$	Leis da idempotência
$p \wedge p \equiv p$	
$\neg(\neg p) \equiv p$	Lei da dupla negação
$p \vee q \equiv q \vee p$	Leis da comutatividade
$p \wedge q \equiv q \wedge p$	

$p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$	Leis da absorção
$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	Leis da associatividade
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	Leis da distributividade
$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$	Leis de De Morgan
$p \rightarrow q \equiv \neg p \vee q$	

Por exemplo, construindo as tabelas de verdade de $\neg(p \wedge q)$ e $\neg p \vee \neg q$

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
V	V	V	F	F	F	F
V	F	F	V	F	V	V
F	V	F	V	V	F	V
F	F	F	V	V	V	V

concluímos que ambas as fbf's têm o mesmo valor lógico para os mesmos valores das variáveis proposicionais, pelo que são logicamente equivalentes.

É possível provar uma equivalência sem construir as tabelas de verdade por causa dos seguintes factos:

1. Se $A \equiv B$ e $B \equiv C$, então $A \equiv C$.
2. Se $A \equiv B$, então qualquer fbf C que contenha A é equivalente à fbf obtida de C substituindo uma ocorrência de A por B.

Exemplo. Use as equivalências básicas na tabela anterior para provar que $p \vee q \rightarrow p \equiv q \rightarrow p$.

Prova:

$$\begin{aligned} p \vee q \rightarrow p &\equiv \neg(p \vee q) \vee p \\ &\equiv (\neg p \wedge \neg q) \vee p \\ &\equiv (\neg p \vee p) \wedge (\neg q \vee p) \\ &\equiv \mathsf{V} \wedge (\neg q \vee p) \\ &\equiv \neg q \vee p \\ &\equiv q \rightarrow p. \qquad QED. \end{aligned}$$

Teste (1 minuto cada). Use equivalências conhecidas para provar:

1. $p \vee q \rightarrow r \equiv (p \rightarrow r) \wedge (q \rightarrow r)$.
2. $(p \rightarrow q) \vee (\neg p \rightarrow q) \equiv \mathsf{V}$ (isto é, $(p \rightarrow q) \vee (\neg p \rightarrow q)$ é uma tautologia).

3. $p \to q \equiv (p \wedge \neg q) \to \mathsf{F}$.

Teste (1 minuto cada). Use as leis da absorção para simplificar:

1. $(p \wedge q \wedge r) \vee (p \wedge r) \vee r$.
2. $(s \to t) \wedge (u \vee t \vee \neg s)$.

Se p é uma variável proposicional numa fbf A, denotemos por $A(p/\mathsf{V})$ a fbf que se obtém de A substituindo todas as ocorrências de p por V . De modo análogo podemos também definir a fórmula $A(p/\mathsf{F})$. As seguintes propriedades verificam-se:

- A é uma tautologia se e só se $A(p/\mathsf{V})$ e $A(p/\mathsf{F})$ são tautologias.
- A é uma contradição se e só se $A(p/\mathsf{V})$ e $A(p/\mathsf{F})$ são contradições.

O *Método de Quine* usa estas propriedades, conjuntamente com as equivalências básicas, para determinar se uma fbf é uma tautologia, uma contradição ou uma contingência. (Trata-se de um método alternativo à construção das tabelas de verdade.)

Exemplo. Seja A a fórmula $(p \wedge q \to r) \wedge (p \to q) \to (p \to r)$. Então:

$$\begin{aligned} A(p/\mathsf{F}) &= (\mathsf{F} \wedge q \to r) \wedge (\mathsf{F} \to q) \to (\mathsf{F} \to r) \\ &\equiv (\mathsf{F} \to r) \wedge \mathsf{V} \to \mathsf{V} \\ &\equiv \mathsf{V}. \end{aligned}$$

Portanto $A(p/\mathsf{F})$ é uma tautologia. A seguir olhemos para

$$\begin{aligned} A(p/\mathsf{V}) &= (\mathsf{V} \wedge q \to r) \wedge (\mathsf{V} \to q) \to (\mathsf{V} \to r) \\ &\equiv (q \to r) \wedge q \to r. \end{aligned}$$

Seja $B = (q \to r) \wedge q \to r$. Então

$$B(q/\mathsf{V}) = (\mathsf{V} \to r) \wedge \mathsf{V} \to r \equiv r \wedge \mathsf{V} \to r \equiv r \to r \equiv \mathsf{V}$$

e

$$B(q/\mathsf{F}) = (\mathsf{F} \to r) \wedge \mathsf{F} \to r \equiv \mathsf{F} \to r \equiv \mathsf{V},$$

o que mostra que B é uma tautologia. Portanto, A é uma tautologia.

Teste (2 minutos cada). Use o método de Quine em cada caso:

1. Mostre que $(p \vee q \to r) \vee p \to (r \to q)$ NÃO é uma tautologia.
2. Mostre que $(p \to q) \to r$ NÃO é equivalente a $p \to (q \to r)$.

No nosso dia a dia raciocinamos e tiramos conclusões usando determinadas regras. A lógica ajuda a compreender essas regras permitindo distinguir entre argumentos correctos e argumentos não correctos. Seguem-se alguns argumentos lógicos, cada um deles com um exemplo e a respectiva formalização.

(1)

1. Se o gato vê o peixe, então o gato apanha o peixe.
2. Se o gato apanha o peixe, então o gato come o peixe.

---

3. Se o gato vê o peixe, então o gato come o peixe.

1. $p \rightarrow q$
2. $q \rightarrow r$

---

3. $p \rightarrow r$

(2)

1. Se o João tem mais de 16 anos, então vai ao cinema.
2. O João tem mais de 16 anos.

---

3. O João vai ao cinema.

1. $p \rightarrow q$
2. p

---

3. q

(3)

1. A Maria vai aos testes ou faz o exame.
2. A Maria não faz o exame.

---

3. A Maria vai aos testes.

1. $p \vee q$
2. $\neg q$

---

3. p

Um argumento da forma "De $A_1, A_2, \ldots,$ e A_n deduz-se B", esquematicamente,

$$\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_n \\ \hline B \end{array}$$

diz-se um *argumento correcto* se $A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B$ for uma tautologia. Neste caso escreve-se

$$A_1, A_2, \ldots, A_n \models B.$$

Para indicar que $A \models B$ também se usa $A \Rightarrow B$, nomenclatura que faz parte do dia a dia da escrita matemática.

Um literal é uma variável proposicional ou a sua negação; por exemplo, p e $\neg p$ são literais (ditos *literais complementares*).

Uma fbf diz-se uma *forma normal disjuntiva* (FND) se for da forma $C_1 \vee C_2 \vee \cdots \vee C_n$, onde cada C_i é uma conjunção de literais (chamada *conjunção fundamental*).

Analogamente, uma fbf diz-se uma *forma normal conjuntiva* (FNC) se for da forma $D_1 \wedge D_2 \wedge \cdots \wedge D_n$, onde cada D_i é uma disjunção de literais (chamada *disjunção fundamental*).

Exemplos de formas normais disjuntivas:

$$p$$

$$\neg p$$

$$p \vee \neg q$$

$$p \wedge \neg q$$

$$(p \wedge q) \vee (p \wedge \neg q)$$

$$p \vee (p \wedge r)$$

Exemplos de formas normais conjuntivas:

$$p$$

$$\neg p$$

$$p \wedge \neg q$$

$$\neg p \vee q$$

$$p \wedge (q \vee r)$$

Como já observámos, para qualquer variável proposicional p temos $\mathsf{V} \equiv p \vee \neg p$ e $\mathsf{F} \equiv p \wedge \neg p$. Ambas as formas são FND e FNC. Qualquer fbf tem uma FND e uma FNC. De facto, em qualquer fbf podemos usar equivalências básicas para obter uma forma normal conjuntiva:

1. "Removem-se" todas as $\rightarrow$.
2. Se a expressão contém negações de conjunções ou negações de disjunções, fazem-se desaparecer usando as leis de De Morgan, e simplifica-se onde necessário.
3. Agora basta usar as duas propriedades distributivas

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$$

   e simplificar onde necessário.

Para obter uma forma normal disjuntiva procede-se de forma análoga, usando agora em 3 as propriedades

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

$$(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r).$$

Teste (2 minutos). Transforme $(p \wedge q) \vee \neg(r \rightarrow s)$ numa FND e numa FNC.

Exemplo. $(p \to q \vee r) \to (p \wedge s)$

$\equiv \neg(p \to q \vee r) \vee (p \wedge s)$ $(x \to y \equiv \neg x \vee y)$

$\equiv (p \wedge \neg(q \vee r)) \vee (p \wedge s)$ $(\neg(x \to y) \equiv x \wedge \neg y)$

$\equiv (p \wedge \neg q \wedge \neg r) \vee (p \wedge s)$ $(\neg(x \vee y) \equiv \neg x \wedge \neg y))$ (FND)

$\equiv ((p \wedge \neg q \wedge \neg r) \vee p) \wedge ((p \wedge \neg q \wedge \neg r) \vee s)$ ($\vee$ é distributiva relativamente a $\wedge$)

$\equiv p \wedge ((p \wedge \neg q \wedge \neg r) \vee s)$ (absorção)

$\equiv p \wedge (p \vee s) \wedge (\neg q \vee s) \wedge (\neg r \vee s)$ ($\vee$ é distributiva relativamente a $\wedge$) (FNC)

$\equiv p \wedge (\neg q \vee s) \wedge (\neg r \vee s)$ (absorção) (FNC).

Uma *função de verdade* (ou *função lógica*) é uma função que só pode tomar os valores lógicos V ou F e cujos argumentos também só podem tomar esses valores. Por exemplo,

$$f(p,q) = \begin{cases} \mathsf{V} \text{ se } p \text{ é } \mathsf{V} \\ \mathsf{V} \text{ se } p \text{ e } q \text{ são ambas } \mathsf{F} \\ \mathsf{F} \text{ se } p \text{ é } \mathsf{F} \text{ e } q \text{ é } \mathsf{V} \end{cases}$$

é uma função de verdade.

É claro que toda a função de verdade pode ser representada por uma tabela de verdade. No exemplo anterior:

p	q	$f(p,q)$
V	V	V
V	F	V
F	V	F
F	F	V

De seguida vamos ver que

Toda a função de verdade é equivalente a uma fbf.

A metodologia a seguir será encontrar uma fbf com a mesma tabela de verdade (podemos construir quer uma FND quer uma FNC).

Técnica. Para construir uma FND, tendo em conta que a disjunção de um número finito de fbf's é V se e só se uma delas o for, basta tomar cada linha da tabela que tenha valor V e construir uma conjunção fundamental que só seja verdadeira nessa linha. De modo análogo, para construir uma FNC, basta considerar cada linha que tenha valor F e construir uma disjunção fundamental que só seja falsa nessa linha.

Exemplo. Na função f acima,

p	q	$f(p,q)$	Partes FND	Partes FNC
V	V	V	$p \wedge q$	
V	F	V	$p \wedge \neg q$	
F	V	F		$p \vee \neg q$
F	F	V	$\neg p \wedge \neg q$	

Assim, $f(p,q)$ pode ser escrita nas formas:

$$f(p,q) \equiv (p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge \neg q) \ \text{(FND)}$$

$$f(p,q) \equiv p \vee \neg q \ \text{(FNC)}.$$

Uma FND para uma fbf A é uma *FND plena* (*forma normal disjuntiva plena*) se cada conjunção fundamental contém o mesmo número de literais, um por cada variável proposicional de A. Uma FNC para uma fbf A é uma *FNC plena* (*forma normal conjuntiva plena*) se cada disjunção fundamental contém o mesmo número de literais, um por cada variável proposicional de A.

Exemplo. No exemplo anterior obtivemos uma FND plena e uma FNC plena.

Podemos usar a técnica das funções de verdade para determinar uma FND plena ou uma FNC plena de qualquer fbf com a excepção das tautologias (não têm uma FNC plena) e das contradições (não têm uma FND). Por exemplo:

$\mathsf{V} \equiv p \vee \neg p$, que é uma FND plena e uma FNC, mas não é uma FNC plena,

$\mathsf{F} \equiv p \wedge \neg p$, que é uma FNC plena e uma FND, mas não é uma FND plena.

Os conectivos lógicos que usámos para definir as fbf's do cálculo proposicional são $\neg$, $\wedge$, $\vee$ e $\rightarrow$. É evidente que o símbolo $\rightarrow$ não é absolutamente necessário (pela última lei da tabela das equivalências básicas): qualquer fbf pode ser substituída por outra logicamente equivalente e onde não figura o símbolo $\rightarrow$.

Um conjunto de conectivos lógicos diz-se *completo* se toda a fbf do cálculo proposicional é equivalente a uma fbf onde figuram apenas conectivos desse conjunto. É claro que

$$\{\neg, \wedge, \vee, \rightarrow\}$$

é completo, por definição.

Exemplos. Cada um dos seguintes conjuntos é um conjunto completo de conectivos:

$$\{\neg, \wedge, \vee\}, \{\neg, \wedge\}, \{\neg, \vee\}, \{\neg, \rightarrow\}, \{\mathsf{F}, \rightarrow\}.$$

Teste (2 minutos cada).

1. Mostre que $\{\neg, \rightarrow\}$ é completo.
2. Mostre que $\{if\text{-}then\text{-}else, \mathsf{V}, \mathsf{F}\}$ é completo.

Observação final. Como vimos, as tabelas de verdade são suficientes para determinar quando uma fbf é uma tautologia. Contudo, quando uma proposição tem mais do que duas variáveis e contém vários conectivos, a tabela de verdade pode começar a ficar muito complicada. Nesses casos, o método alternativo que vimos de encontrar uma prova de equivalência usando as leis

de equivalência básicas ou ainda uma combinação dos dois (por exemplo, o método de Quine) pode ser mais prático.

Quando usamos uma prova de equivalência, em vez de uma tabela de verdade, para verificar se duas fbf's são equivalentes, isso parece de certo modo mais parecido com o modo como comunicamos habitualmente. Embora não seja necessário raciocinar formalmente desse modo no cálculo proposicional, há outros tipos de sistemas lógicos onde isso já é necessário para averiguar da validade das fbf's pois aí as tabelas de verdade não funcionam. Para esses casos existe uma ferramenta: os sistemas de raciocínio formal. Quais são as ideais básicas destes sistemas?

Um *sistema formal* consiste em três partes:

(1) Um conjunto (numerável) de símbolos.

(2) Um conjunto de sequências finitas destes símbolos que constituem as chamadas *fórmulas bem formadas*.

(3) Um determinado conjunto de fbf's, chamadas *axiomas*, que se assumem ser verdadeiras.

(4) Um conjunto finito de "regras de dedução" chamadas *regras de inferência* que permitem deduzir uma fbf como consequência directa de um conjunto finito de fbf's.

Um sistema formal requere algumas regras que ajudem à obtenção de novas fórmulas, as chamadas *regras de inferência*. Uma regra de inferência aplica uma ou mais fbf's, chamadas *premissas*, *hipóteses* ou *antecedentes*, numa só fórmula, chamada *conclusão* ou *consequente*. Algumas regras de inferência úteis:

MP (modus ponens)	MT (modus tollens)	Conj (conjunção)
$\dfrac{p \to q,\ p}{\therefore q}$	$\dfrac{p \to q,\ \neg q}{\therefore \neg p}$	$\dfrac{p,\ q}{\therefore p \wedge q}$
Ad (adição)	**SD (silogismo disjuntivo)**	**SH (silogismo hipotético)**
$\dfrac{p}{\therefore p \vee q}$	$\dfrac{p \vee q,\ \neg p}{\therefore q}$	$\dfrac{p \to q,\ q \to r}{\therefore p \to q}$

Aqui o conceito crucial é o de dedução. Uma dedução de uma certa conclusão — digamos S — a partir de premissas $P_1, P_2, \ldots, P_n$ é feita passo a passo. Numa dedução, estabelecem-se conclusões intermédias, cada uma delas conclusão imediata das premissas e conclusões intermédias anteriores. Podemos dizer que uma dedução consiste numa sucessão de afirmações, que são premissas ou conclusões intermédias, e que termina, ao fim de um número finito de passos, quando se obtém a conclusão S.

Cada passo de dedução é correcto, i.e., não oferece dúvidas quanto à validade de cada conclusão intermédia, em consequência da validade das premissas e das conclusões intermédias anteriores.

Uma dedução de uma afirmação S a partir de premissas $P_1, P_2, \ldots, P_n$ é uma demonstração passo a passo que permite verificar que S tem que ser verdadeira em todas as circunstâncias em

que as premissas sejam verdadeiras. Uma dedução formal assenta num conjunto fixo de regras de dedução e tem uma apresentação rígida — um pouco à semelhança dos programas escritos numa dada linguagem de programação.

Seja $\mathcal{C}$ um conjunto de fbf's e seja P uma fbf em S. Diz-se que P *é dedutível a partir de* C em S, e escreve-se $C \vdash_S P$ (ou apenas $C \vdash P$ se não houver dúvidas sobre o sistema S a que nos referimos) se existir uma sequência finita de fbf's, $P_1, P_2, \ldots, P_n$ tais que:

- $P_n = P$.
- Para cada $i \in \{1, \ldots, n\}$, P_i é um axioma de S ou uma fbf em C ou uma consequência dos P_i's anteriores através de aplicação das regras de inferência.

A sequência dos P_i's diz-se uma *prova formal* de P a partir de C. Se P é dedutível de um conjunto vazio escreve-se $\vdash_S P$. Neste caso, P diz-se um teorema de S.

Exemplo. $A \to (B \to C), A \to B, A \vdash C$.

Prova:	1. $A \to (B \to C)$	premissa	
	2. $A \to B$	premissa	
	3. A	premissa	
	4. B	MP(2,3)	
	5. $B \to C$	MP(1,3)	
	6. C	MP(4,5)	*QED.*

Leituras suplementares:

- James Hein, *Discrete Structures, Logic and Computability*, Secção 6.3.
- Jon Barwise e John Etchemendy, *Language, Proof and Logic*, Capítulo 6.

Apêndice: lógica de predicados; linguagens de primeira ordem

O cálculo proposicional providencia ferramentas adequadas para raciocinarmos sobre fbf's que são combinações de proposições. Mas uma proposição é uma sentença tomada como um todo o que faz com que o cálculo proposicional não sirva para todo o tipo de raciocínio que precisamos de fazer no dia a dia. Por exemplo, no argumento seguinte é impossível encontrar no cálculo proposicional um método formal para testar a correcção da dedução sem uma análise mais profunda de cada sentença:

- Todos os alunos de Engenharia Informática têm um computador portátil.
- Sócrates não tem um computador portátil.
- Então Sócrates não é um aluno de Engenharia Informática.

Para discutir e analisar este argumento precisamos de partir as sentenças em partes. As palavras "Todos", "têm" e "não" são relevantes para a compreensão do argumento. A afirmação "x tem um computador portátil" não é uma proposição porque o seu valor de verdade não é absoluto, depende de x. De um ponto de vista gramatical, a propriedade "ter um computador portátil" é um *predicado* (ou seja, a parte da frase que enuncia uma propriedade do sujeito). Do ponto de vista matemático, um predicado é uma relação (unária, binária, ternária, etc.). Por exemplo, "ter um computador portátil" é um predicado unário que podemos designar por CP; então $CP(x)$ significa que x tem um computador portátil (por exemplo, acima afirma-se que $\neg CP(Socrates)$).

Portanto, para analisarmos argumentos deste tipo (necessários nas aulas de Análise Matemática e de Álgebra Linear) precisamos de um cálculo de predicados, que inclua quantificadores (existencial e total). Por exemplo, se quisermos representar formalmente a afirmação

Existe um conjunto de números naturais que não contém o 4

precisamos de um quantificador existencial:

$\exists S(S$ é um subconjunto de $\mathbb{N}$ e $\neg S(4))$.

Podemos continuar com a formalização pois uma afirmação do tipo "A é um subconjunto de B" pode ser formalizada como $\forall x(A(x) \rightarrow B(x))$. Então podemos escrever a afirmação inicial na forma:

$\exists S(\forall x(S(x) \rightarrow \mathbb{N}(x)) \wedge \neg S(4))$.

Para mais exemplos e para testar os seus conhecimentos de utilização dos quantificadores e predicados utilize o `Tarski World`[3].

Costuma-se classificar o cálculo proposicional como uma *lógica de ordem zero*. Porquê? Porque não permite que os conjuntos sejam quantificados e que sejam elementos de outros

[3]Consulte o apêndice Usando Tarksi.

conjuntos. O cálculo de predicados já é uma lógica de ordem superior: permite que os conjuntos sejam quantificados e que sejam elementos de outros conjuntos. De que ordem?

Diz-se que um predicado tem *ordem* 1 se todos os seus argumentos são termos (isto é, constantes, variáveis individuais ou valores de funções). Caso contrário, diz-se que tem *ordem* $n+1$, onde n é a maior ordem entre os seus argumentos que não são termos. Por exemplo, em $S(x) \wedge T(S)$, o predicado S tem ordem 1 e o predicado T tem ordem 2. Em $p(f(x)) \wedge q(f)$, p tem ordem 1, f tem ordem 1 e q tem ordem 2.

Uma *lógica de ordem n* é uma lógica cujas fbf's têm ordem $\leq n$.

Após termos estudado um pouco de cálculo proposicional, o passo seguinte, que não efectuaremos, seria estudar uma lógica de primeira ordem, nomeadamente o chamado *cálculo de predicados de primeira ordem*. Em vez disso aprenderemos, praticando, uma linguagem simbólica simples de primeira ordem, trabalhando com o `Tarski World`.

Uma linguagem de primeira ordem serve para descrever mundos da seguinte maneira:

- Cada nome deve designar um objecto.
- Nenhum nome pode designar mais do que um objecto.
- Um objecto pode ter vários nomes, mas também pode não ter nome.

As constantes são símbolos referentes a objectos previamente fixados.

Língua Portuguesa	LPO
nome	constante (designação, termo)

Os *símbolos de predicado* ou *relacionais* são símbolos que designam propriedades dos objectos ou relações entre objectos.

Por exemplo, podemos usar o símbolo EEI, um símbolo de predicado unário (ou seja, de aridade um), para designar, no universo dos alunos deste curso, `ser Estudante de Engenharia Informática`. Outro exemplo: o símbolo $<$, um símbolo de predicado binário (ou seja, de aridade dois), representa, no universo dos números reais, `ser menor do que`.

Língua Portuguesa	LPO
O Pedro é estudante de Eng. Inf.	$EEI(Pedro)$
2 é menor que 1	$2 < 1$

Portanto, numa linguagem de primeira ordem, a cada símbolo de predicado está associado exactamente um número natural — o número de argumentos que ocorre no predicado, que se designa por *aridade*. Além disso, cada símbolo de predicado ou relacional é interpretado por uma propriedade bem determinada ou uma relação com a mesma aridade que o símbolo.

Uma *sentença atómica* é uma sequência finita de símbolos, escolhidos entre as constantes, os símbolos de predicados, os parênteses "(" e ")" e a vírgula, da forma

$$P(c_1), \quad T(c_1, c_2), \quad R(c_1, c_2, c_3)$$

onde c_1, c_2, c_3 são constantes e P, T, R são símbolos de predicados num vocabulário fixado.

Exemplos.

Língua Portuguesa	LPO
A Rita é estudante de Comunicações e Multimédia	$CM(Rita)$
O Nuno é mais velho do que o Pedro	$MaisVelho(Nuno, Pedro)$

A notação usual é a *prefixa*: o símbolo de predicado escreve-se à esquerda. Excepções: com o símbolo de igualdade $=$ utiliza-se a notação habitual $a = b$; com os símbolos $<, >$ também se utiliza a notação habitual: $1 < 2$, $2 > 1$.

Portanto, com algumas excepções (predicados $=, <, >$), numa linguagem de primeira ordem as sentenças atómicas são expressões que se obtêm escrevendo um símbolo de predicado de aridade n, seguido de n constantes, delimitadas por parênteses e separadas por vírgulas: $P(c_1, c_2, \ldots, c_n)$. As excepções $(=, <, >)$ podem ser estendidas a outros símbolos. Note que a ordem em que as constantes ocorrem é fundamental.

Especifica-se uma linguagem de primeira ordem fixando as constantes, os símbolos de predicado e os símbolos funcionais. Cada símbolo de predicado e cada símbolo funcional tem uma aridade bem determinada. Uma linguagem de primeira ordem pode não incluir símbolos funcionais, mas necessita sempre de símbolos relacionais. No entanto, em vários exemplos, o único símbolo relacional considerado é o da igualdade $=$.

As linguagens de primeira ordem podem assim distinguir-se entre si através das respectivas constantes, símbolos de predicado e símbolos funcionais. Partilham os conectivos $\neg, \wedge, \vee, \rightarrow$ e $\leftrightarrow$ e os quantificadores $\forall, \exists$.

Quando se traduz uma frase em Língua Portuguesa para uma sentença numa linguagem de primeira ordem, tem-se em geral uma linguagem previamente definida, em que se conhecem à partida as constantes, os símbolos relacionais e (caso existam) os símbolos funcionais. No entanto, há situações em que há que decidir quais as constantes, os símbolos relacionais e (caso existam) os símbolos funcionais adequados para expressar o que se pretende.

Exemplo. Consideremos a frase O Nuno explicou o Tarski World ao Pedro.

(1) Tomando o símbolo de predicado binário $ExplicouTarskiWorld$ podemos escrever

$$ExplicouTarskiWorld(Nuno, Pedro).$$

(2) Tomando o símbolo de predicado ternário $Explicou$ podemos escrever

$$Explicou(Nuno, TarskiWorld, Pedro).$$

O poder expressivo da linguagem (2) é maior do que o da linguagem (1). De facto, considerando a frase A Rita explicou o Boole ao Miguel, esta pode ser traduzida usando o símbolo de predicado ternário $Explicou$ — teríamos $Explicou(Rita, Boole, Miguel)$ — mas não pode ser traduzida usando o símbolo de predicado $ExplicouTarskiWorld$. O símbolo de predicado $Explicou$ é mais versátil do que os símbolos de predicado $ExplicouTarskiWorld$ ou $ExplicouBoole$.

Para considerar as frases A Rita explicou o Boole ao Miguel no sábado e No domingo, o Miguel explicou o Boole ao João podemos considerar um predicado quaternário $Explicou(x, y, z, w)$ — que se lê "x explicou y a z no w" — e traduzir as duas frases consideradas para LPO:

$$Explicou(Rita, Boole, Miguel, sábado)$$

$$Explicou(Miguel, Boole, João, domingo).$$

Os *símbolos funcionais* são símbolos que permitem obter outras designações para objectos.

Exemplos. (1) Jorge é pai do Nuno. Supondo que a afirmação é verdadeira, $Jorge$ e $pai(Nuno)$ são duas designações diferentes para o mesmo indivíduo; *pai* é um *símbolo funcional unário.*

(2) As expressões 3 e $((1 + 1) + 1)$ são duas designações diferentes do mesmo número natural; $+$ é um *símbolo funcional binário.*

As expressões 1, $(1 + 1)$ e $((1 + 1) + 1)$ são termos. A definição de *termo* é a seguinte:

- Constantes são termos.
- Se F é um símbolo funcional de aridade n e $t_1, t_2, \ldots, t_n$ são n termos, então a expressão $F(t_1, t_2, \ldots, t_n)$ é um termo.

Numa linguagem de primeira ordem com símbolos funcionais

- Termos complexos obtêm-se colocando um símbolo funcional n-ário antes de um n-tuplo de n termos. Excepção: certos símbolos funcionais binários escrevem-se entre termos, como por exemplo $+$ (por exemplo, $(1 + 1)$).
- Termos usam-se como nomes ou designações na formação de sentenças atómicas.

Exemplos de linguagens de primeira ordem.
(1) `A Linguagem de Primeira Ordem da Teoria de Conjuntos`. Na linguagem de primeira ordem da Teoria de Conjuntos tem-se apenas dois símbolos de predicados, ambos binários, $=$ e $\in$. As sentenças atómicas nesta linguagem são da forma $a = b$ (lê-se "a é igual a b") e $a \in b$ (lê-se "(o elemento) a pertence ao (conjunto) b"), sendo a e b constantes individuais.

Por exemplo, supondo que a designa 2, b designa o conjunto $\mathbb{N}$ dos números naturais e c designa o conjunto dos números ímpares, tem-se:

$a \in a$	sentença falsa
$a \in b$	sentença verdadeira
$a \in c$	sentença falsa
$b = c$	sentença falsa.

(2) `A Linguagem de Primeira Ordem da Aritmética`. A linguagem de primeira ordem da Aritmética contém duas constantes 0 e 1, dois símbolos relacionais binários $=$ e $<$ e dois símbolos funcionais binários $+$ e $\times$. São termos desta linguagem $0, 1, (1+1), ((1+1)+1), (0 \times (1+1)), \ldots$

Os termos na aritmética de primeira ordem formam-se segundo as regras:

- As constantes 0, 1 são termos.
- Se t_1 e t_2 são termos, também são termos as expressões $(t_1 + t_2)$ e $(t_1 \times t_2)$.
- São termos apenas as expressões que possam ser obtidas por aplicação sucessiva dos passos anteriores um número finito de vezes.

As sentenças atómicas na aritmética de primeira ordem são as expressões que se podem escrever usando os termos (no lugar das constantes) e os símbolos relacionais $=, <$:

- Se t_1 e t_2 são termos, são sentenças atómicas as expressões $t_1 = t_2$ e $t_1 < t_2$.

Com o `Tarski World` trabalharemos com linguagens de primeira ordem simples. Aí podemos construir mundos tridimensionais habitados por blocos geométricos de diversos tipos e tamanhos, e usar uma linguagem de primeira ordem para descrever esses mundos e testar o valor lógico (verdadeiro ou falso) de sentenças de primeira ordem elaboradas sobre esses mundos.

1.2. Raciocínio matemático, indução e recursão

Para entendermos um texto matemático temos que compreender o que faz com que um argumento esteja matematicamente correcto, isto é, seja uma prova. Uma *prova* é uma demonstração de que alguma afirmação é verdadeira. Normalmente apresentamos provas escrevendo frases em Português misturadas com equações e símbolos matemáticos.

Quando é que um argumento matemático está correcto?

Um *teorema* é uma afirmação que se pode demonstrar ser verdadeira. Demonstra-se que um teorema é verdadeiro com uma sequência de afirmações que formam um argumento, chamada *prova*. Para construir provas, precisamos de métodos que nos permitam deduzir novas afirmações a partir de afirmações já comprovadas. As afirmações usadas numa prova incluem os *axiomas* ou *postulados* da teoria, as hipóteses do teorema a provar e teoremas previamente provados. As *regras de inferência* são as ferramentas para deduzir novas conclusões e ligam os diversos passos da prova. Recorde de 1.1 que um argumento do tipo

$$\begin{array}{l} A_1 \\ A_2 \\ \vdots \\ \underline{A_n} \\ \therefore B \end{array}$$

é uma regra de inferência se $A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B$ for uma tautologia. Por exemplo, é a tautologia $(p \wedge (p \rightarrow q)) \rightarrow q$ que está por trás da regra *modus ponens*, como vimos na secção anterior. Vimos na altura, também, uma lista das regras de inferência mais usadas no raciocínio matemático:

Regras de inferência	**Tautologia**	**Nome**
$\begin{array}{l} p \\ \underline{p \rightarrow q} \\ \therefore q \end{array}$	$(p \wedge (p \rightarrow q)) \rightarrow q$	Modus ponens (MP)
$\begin{array}{l} \neg q \\ \underline{p \rightarrow q} \\ \therefore \neg p \end{array}$	$(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$	Modus tollens (MT)
$\begin{array}{l} \underline{p} \\ \therefore p \vee q \end{array}$	$p \rightarrow (p \vee q)$	Adição (Ad)
$\begin{array}{l} \underline{p \wedge q} \\ \therefore p \end{array}$	$(p \wedge q) \rightarrow p$	Simplificação (S)
$\begin{array}{l} p \vee q \\ \underline{\neg p} \\ \therefore q \end{array}$	$((p \vee q) \wedge \neg p) \rightarrow q$	Silogismo disjuntivo (SD)
$\begin{array}{l} p \rightarrow q \\ \underline{q \rightarrow r} \\ \therefore p \rightarrow r \end{array}$	$((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$	Silogismo hipotético (SH)

Qualquer argumento elaborado com regras de inferência diz-se *válido*. Quando todas as afirmações usadas num argumento válido são verdadeiras, podemos ter a certeza de chegar a uma conclusão correcta. No entanto, um argumento válido pode conduzir a conclusões incorrectas se uma ou mais proposições falsas são usadas no argumento. Por exemplo,

> "*Se* 101 *é divisível por* 3 *então* 101^2 *é divisível por* 9. 101 *é divisível por* 3. *Logo,* 101^2 *é divisível por* 9."

é um argumento válido (baseado na regra MP) mas a conclusão é falsa: 9 não divide $101^2 = 10\,201$.

Noutro tipo de falácias muito comum as conclusões estão incorrectas porque os argumentos não são válidos: apesar de aparentarem ser regras de inferência, na realidade não o são (baseiam-se em contingências e não em tautologias).

Exemplo 1. A proposição $((p \to q) \wedge q) \to p$ não é uma tautologia (é falsa quando p é falsa e q é verdadeira). No entanto, por vezes é usada como se fosse uma tautologia (este tipo de argumento incorrecto chama-se *falácia de afirmar a conclusão*):

> *Se resolver todos os exercícios destes apontamentos, então aprenderá matemática discreta. Aprendeu matemática discreta. Logo, resolveu todos os exercícios.*

(É claro que pode aprender matemática discreta sem precisar de resolver todos os exercícios destes apontamentos!)

Exemplo 2. A proposição $((p \to q \wedge \neg p) \to \neg q$ não é uma tautologia, pois é falsa quando p é falsa e q é verdadeira. É outro exemplo de proposição que por vezes é usada como regra de inferência em argumentos incorrectos (a chamada *falácia de negar a hipótese*):

> *Se resolver todos os exercícios destes apontamentos, então aprenderá matemática discreta. Não resolveu todos os exercícios. Logo, não aprendeu matemática discreta.*

(É claro que pode aprender matemática discreta sem ter resolvido todos os exercícios destes apontamentos!)

Que métodos podemos usar para elaborar argumentos matemáticos correctos?

Os matemáticos usam diversos métodos para provar a validade de uma proposição ou argumento. Façamos uma breve digressão, com exemplos, pelos mais comuns.

(1) Verificação exaustiva. Algumas proposições podem ser provadas por verificação exaustiva de um número finito de casos.

Exemplo 1. *Existe um número primo entre 890 e 910.*

Prova. Verificando exaustivamente descobrirá que o 907 é primo. □

Exemplo 2. *Cada um dos números 288, 198 e 387 é divisível por 9.*

Prova. Verifique que 9 divide cada um desses números. □

É claro que uma proposição enunciada para um número infinito de casos não poderá ser provada directamente por verificação exaustiva (por mais casos que consigamos verificar nunca conseguiremos verificar todos...). Por exemplo, se tentarmos comprovar, com a ajuda do `Maple`, a famosa **conjectura de Goldbach**, que afirma que

qualquer inteiro par maior do que 2 pode escrever-se como soma de dois primos

não encontraremos nenhum *contra-exemplo* (isto é, um exemplo que refute a conjectura). Mesmo assim, não poderemos garantir que a conjectura é verdadeira para qualquer inteiro par maior do que 2 (tal como ninguém o conseguiu fazer até hoje!).

Procedimento que calcula a decomposição de um inteiro par na soma de dois primos:

```
> Goldbach := proc(p::integer)
>   local i,j,terminou,next_i_valor;
>   terminou := false;
>   i := 0; j := 0;
>   while not terminou do
>      next_i_valor := false;
>      i := i+1; j :=i;
>      while not next_i_valor do
>         if ithprime(i) + ithprime (j) = p then
>            printf(‘%d pode ser expresso como %d+%d\n‘,p,ithprime(i),ithprime(j));
>            terminou := true;
>            next_i_valor := true;
>         fi;
>         j := j+1;
>         if ithprime(j) >= p then
>            next_i_valor := true
>         fi;

>      od;
>   od;
> end:

> Goldbach(156456);

156456 pode ser expresso como 19+156437
```

Procedimento mais automático, que realiza o cálculo para todos os inteiros pares entre n e m:

```
> ManyGoldbach := proc(startval::integer,finalval::integer)
>    local i;
>    for i from max(4,startval) to finalval do
```

```
>        if i mod 2 = 0 then Goldbach(i);
>        fi;
>    od;
> end:
```

Verificação da conjectura para os inteiros pares entre 4 e 50:

```
> ManyGoldbach(4,50);
```

```
4 pode ser expresso como 2+2
6 pode ser expresso como 3+3
8 pode ser expresso como 3+5
10 pode ser expresso como 3+7
12 pode ser expresso como 5+7
14 pode ser expresso como 3+11
16 pode ser expresso como 3+13
18 pode ser expresso como 5+13
20 pode ser expresso como 3+17
22 pode ser expresso como 3+19
24 pode ser expresso como 5+19
26 pode ser expresso como 3+23
28 pode ser expresso como 5+23
30 pode ser expresso como 7+23
32 pode ser expresso como 3+29
34 pode ser expresso como 3+31
36 pode ser expresso como 5+31
38 pode ser expresso como 7+31
40 pode ser expresso como 3+37
42 pode ser expresso como 5+37
44 pode ser expresso como 3+41
46 pode ser expresso como 3+43
48 pode ser expresso como 5+43
50 pode ser expresso como 3+47
```

(2) Prova de implicações (condicionais). A maioria dos teoremas que se provam em matemática são implicações (ou equivalências, que são conjunções de duas implicações). A implicação ("se p então q" ou "p implica q") é uma afirmação *condicional* com *hipótese* p e *conclusão* q. A sua *contraposta* é a afirmação "se não q então não p" e a sua *recíproca* é "se q então p".

Como a maioria dos teoremas utilizados na prática da matemática são implicações, as técnicas para provar implicações são muito importantes. Para provar que $p \Rightarrow q$, ou seja, que $p \to q$ é uma tautologia, mostra-se que se p é verdadeira também q o é: começamos por assumir que a hipótese p é verdadeira; depois tentamos encontrar uma proposição que resulte da hipótese e/ou factos conhecidos; continuamos deste modo até chegarmos à conclusão q.

Exemplo 3. *Se m é ímpar e n é par então $m - n$ é ímpar.*

Prova. Suponhamos que m é ímpar e n é par. Então $m = 2k + 1$ e $n = 2l$ para alguns inteiros k e l. Portanto, $m - n = 2k + 1 - 2l = 2(k - l) + 1$, que é um inteiro ímpar. □

Exemplo 4. *Se n é ímpar então n^2 é ímpar.*

Prova. Suponhamos que n é ímpar. Então $n = 2k + 1$ para algum inteiro k. Portanto, $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$, que é um inteiro ímpar pois $2k^2 + 2k$ é um inteiro. □

(2a) Prova indirecta. A seguinte tabela de verdade mostra que uma condicional e a sua contraposta são equivalentes:

p	q	$\neg q$	$\neg p$	$p \to q$	$\neg q \to \neg p$
V	V	F	F	V	V
V	F	V	F	F	F
F	V	F	V	V	V
F	F	V	V	V	V

Esta equivalência proporciona um método alternativo para provar uma implicação (o chamado *método indirecto*).

Exemplo 5. *Se n^2 é par então n é par.*

Prova. A contraposta desta afirmação é "se n é ímpar então n^2 é ímpar", que é verdadeira pelo Exemplo anterior. □

Teste (1 minuto). Prove as recíprocas dos Exemplo 4 e 5.

(2b) Prova por contradição (redução ao absurdo). Da tabela de verdade da implicação decorre imediatamente que "se p então q" é equivalente a "p e não q implica falso":

p	q	$p \to q$	$p \wedge \neg q$	$p \wedge \neg q \to \mathsf{F}$
V	V	V	F	V
V	F	F	V	F
F	V	V	F	V
F	F	V	F	V

Portanto temos aqui mais um método alternativo de demonstrar uma implicação: para provar "se p então q" é suficiente provar "p e não q implica falso", ou seja, assumir p e $\neg q$ e depois argumentar de modo a chegar a uma contradição (proposição sempre falsa, como vimos em 1.1). Chama-se a esta técnica de demonstração *prova por contradição* (ou *por redução ao absurdo*).

Exemplo 6. *Se n^2 é ímpar então n é ímpar.*

Prova. Suponhamos, *por absurdo*, que n^2 é ímpar e n é par. Então $n = 2k$ para algum inteiro k pelo que $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$ é também um inteiro par. Chegamos assim à conclusão que n^2 é simultaneamente um inteiro par e ímpar, o que é uma contradição. □

Exemplo 7. *Se $2|5n$ então n é par.*

Prova. Suponhamos, *por absurdo*, que $2|5n$ e n é ímpar. Então $5n = 2d$ para algum inteiro d e $n = 2k + 1$ para algum inteiro k. Juntando tudo obtemos $2d = 5n = 5(2k + 1) = 10k + 5$. Logo $5 = 2d - 10k = 2(d - 5k)$, o que é uma contradição pois afirma que 5 é um número par! □

Exemplo 8. *$\sqrt{2}$ é um número irracional.*

Prova. Suponhamos, *por absurdo*, que $\sqrt{2}$ é racional. Então $\sqrt{2} = \frac{p}{q}$ para algum par de inteiros p e q (podemos assumir que a fracção p/q está já escrita na sua forma reduzida, isto

é, $\text{mdc}(p,q) = 1$). Elevando ao quadrado ambos os membros da igualdade anterior obtemos $2q^2 = p^2$, pelo que p^2 é par. Então, pelo Exemplo 5, p é par. Sendo p par, é claro que p^2 é um múltiplo de 4, ou seja, $p^2 = 4k$ para algum inteiro k. Consequentemente, $2q^2 = 4k$, isto é, $q^2 = 2k$ é par, pelo que q também é par. Chegámos aqui a uma contradição: p e q são pares mas $\text{mdc}(p,q) = 1$. □

(3) Prova de equivalências ("se e só se"; abreviadamente "sse"). Uma proposição da forma $p \Leftrightarrow q$ ("p se e só se q") significa a conjunção de "p implica q" e "q implica p". Portanto, é preciso apresentar duas provas. Por vezes, estas provas podem ser escritas como uma só prova na forma "p sse r sse s sse ... sse q", onde cada "... sse ..." é conclusão evidente da informação anterior.

Exemplo 9. *n é par se e só se $n^2 - 2n + 1$ é ímpar.*
Prova.

n é par	sse	$n = 2k$ para algum inteiro k	(definição)
	sse	$n - 1 = 2k - 1$ para algum inteiro k	(álgebra)
	sse	$n - 1 = 2(k-1) + 1$ para algum inteiro $k - 1$	(álgebra)
	sse	$n - 1$ é ímpar	(definição)
	sse	$(n-1)^2$ é ímpar	(Exemplo 4 e Teste)
	sse	$n^2 - 2n + 1$ é ímpar	(álgebra) □

Muitas vezes um teorema afirma que determinadas proposições $p_1, p_2, \ldots, p_n$ são equivalentes, isto é, que $p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n$ é uma tautologia (o que assegura que as n proposições têm a mesma tabela de verdade). Uma maneira de provar o teorema é usar a tautologia

$$(p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n) \leftrightarrow ((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \cdots \wedge (p_n \rightarrow p_1))$$

que assegura a equivalência

$$(p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n) \equiv ((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \cdots \wedge (p_n \rightarrow p_1)).$$

Exemplo 10. *Para cada inteiro n as proposições seguintes são equivalentes:*

(i) *n é ímpar.*

(ii) *n^2 é ímpar.*

(iii) *$n^2 - 2n + 1$ é par.*

Prova. Para provar a equivalência das três asserções, basta provar que as implicações (i)→(ii), (ii)→(iii) e (iii)→(i) são verdadeiras:
(i)→(ii): Provada no Exemplo 4.
(ii)→(iii): Se n^2 é ímpar então $n^2 + 1$ é par. Como $2n$ é sempre par, então $n^2 - 2n + 1$ é par.
(iii)→(i): No Exemplo 6 provámos que se n é par então $n^2 - 2n + 1$ é ímpar. A implicação (iii)→(i) é a sua contraposta, pelo que também é verdadeira. □

(4) Prova por indução matemática. A que é igual a soma dos n primeiros inteiros positivos ímpares? Para $n = 1, 2, 3, 4, 5$ tem-se

$$\begin{aligned} &1 = 1, \\ &1 + 3 = 4, \\ &1 + 3 + 5 = 9, \\ &1 + 3 + 5 + 7 = 16, \\ &1 + 3 + 5 + 7 + 9 = 25. \end{aligned}$$

Destes valores particulares é razoável conjecturar[4] que a soma, para qualquer n, deverá ser igual a n^2. O *método de indução matemática* permite-nos provar facilmente que esta conjectura está correcta[5]. Trata-se de um método muito potente para provar asserções deste tipo, enunciadas sobre o conjunto $\mathbb{N}$ dos naturais. Baseia-se na observação óbvia que todo o subconjunto não vazio de $\mathbb{N}$ tem um elemento mínimo e no seguinte:

Base do Princípio de Indução Matemática. *Seja $S \subseteq \mathbb{N}$, $1 \in S$ e suponhamos que $k \in S$ implica $k + 1 \in S$. Então $S = \mathbb{N}$.*

Prova. Suponhamos, por absurdo, que $S \neq \mathbb{N}$. Então $\mathbb{N} - S \neq \varnothing$ logo tem um elemento mínimo m. Como $1 \in S$ e $m \notin S$, então $m > 1$. Portanto, $m - 1$ é um natural e pertence a S (pois m é o mínimo de $\mathbb{N} - S$). Logo, por hipótese, $(m - 1) + 1 \in S$, isto é, $m \in S$. Chegamos assim a uma contradição: $m \notin S$ e $m \in S$. Em conclusão, $S = \mathbb{N}$. □

Princípio de Indução Matemática (PIM). *Seja $P(n)$ uma proposição para cada $n \in \mathbb{N}$. Para provar que $P(n)$ é verdadeira para qualquer $n \in \mathbb{N}$ basta:*

(1) (Passo inicial) *Mostrar que $P(1)$ é verdadeira.*

(2) (Passo indutivo) *Mostrar que a implicação $P(n) \rightarrow P(n + 1)$ é verdadeira para qualquer $n \in \mathbb{N}$.*

Prova. Suponhamos que os dois passos foram provados. Seja $S = \{n \mid P(n) \text{ é verdadeira}\}$. O passo inicial garante que $1 \in S$; por outro lado, o passo indutivo garante que $k \in S$ implica $k + 1 \in S$. Então, pela Base do PIM, podemos concluir que $S = \mathbb{N}$. □

No passo indutivo, $P(n)$ chama-se *hipótese de indução.*

Exemplo 11. *Para qualquer natural n, a soma dos n primeiros inteiros positivos ímpares é igual a n^2.*

Prova. Seja $P(n)$ a proposição $1 + 3 + 5 + \cdots + (2n - 1) = n^2$. $P(1)$ é claramente verdadeira: $1 = 1^2$. Assumindo que $P(n)$ é verdadeira, provemos que $P(n + 1)$ é verdadeira. O membro esquerdo de $P(n + 1)$ é:

$$\begin{aligned} 1 + 3 + 5 + \cdots + (2n - 1) + (2n + 1) &= (1 + 3 + 5 + \cdots + (2n - 1)) + (2n + 1) \\ &= n^2 + (2n + 1) && \text{(hip. indução)} \\ &= (n + 1)^2 && \text{(álgebra)} \end{aligned}$$

[4] Veja `www.mat.uc.pt/~picado/ediscretas/somatorios/Matematica_sem_palavras_files/soma_impares.html`.

[5] Assim como todas as outras em `www.mat.uc.pt/~picado/ediscretas/somatorios`.

que é o membro direito de $P(n+1)$. Portanto, $P(n+1)$ é verdadeira e, pelo PIM, segue que $P(n)$ é verdadeira para qualquer n. □

Exemplo 12. *Seja* $f : \mathbb{N} \to \mathbb{N}$ *definida por*

$$f(n) = \begin{cases} 1 & \text{se } n = 1 \\ f(n-1) + n^2 & \text{se } n \neq 1. \end{cases}$$

Então $f(n) = \dfrac{n(n+1)(2n+1)}{6}$ *para qualquer* n.

Prova. Seja $P(n)$ a proposição $f(n) = n(n+1)(2n+1)/6$. Como $f(1) = 1$ e $1(1+1)(2+1)/6 = 1$, então $P(1)$ é verdadeira. Assumindo que $P(n)$ é verdadeira, provemos que $P(n+1)$ é verdadeira. O membro esquerdo de $P(n+1)$ é:

$$\begin{aligned} f(n+1) &= f(n+1-1) + (n+1)^2 && \text{(definição de } f\text{)} \\ &= f(n) + (n+1)^2 && \text{(álgebra)} \\ &= \tfrac{n(n+1)(2n+1)}{6} + (n+1)^2 && \text{(hip. indução)} \\ &= \tfrac{(n+1)(2n^2+7n+6)}{6} && \text{(álgebra)} \\ &= \tfrac{(n+1)(n+2)(2n+3)}{6} && \text{(álgebra)} \\ &= \tfrac{(n+1)((n+1)+1)(2(n+1)+1)}{6} && \text{(álgebra)} \end{aligned}$$

que é o membro direito de $P(n+1)$. Portanto, $P(n+1)$ é verdadeira e, pelo PIM, segue que $P(n)$ é verdadeira para qualquer n. □

Observação. É claro que se verificarmos que uma dada proposição sobre os naturais n é válida para muitos valores de n, isso não significa automaticamente que a proposição seja verdadeira para todo o n (daí a utilidade do PIM). Por exemplo, um número natural n diz-se de *tipo par* caso a sua decomposição em primos tenha um número par de factores, caso contrário diz-se de *tipo ímpar*. Os números 4 e 24 são de tipo par (pois $4 = 2 \times 2$ e $24 = 2 \times 2 \times 2 \times 3$) enquanto 30 e 18 são de tipo ímpar (pois $30 = 2 \times 3 \times 5$ e $18 = 2 \times 3 \times 3$). Seja $P(n)$ o número de naturais $\leq n$ de tipo par e seja $I(n)$ o número de naturais $\leq n$ de tipo ímpar. Se verificarmos para alguns valores de $n \geq 2$ constataremos sempre que $I(n) \geq P(n)$. Esta afirmação é a conhecida *conjectura de Polya*, formulada em 1919. Depois de verificada para todo o n inferior a 1 milhão, muitas pessoas convenceram-se que teria de ser verdadeira. No entanto, em 1962, R. Sherman Lehman encontrou um contra-exemplo: para $n = 906\,180\,359$, tem-se $I(n) = P(n) - 1$. O menor contra-exemplo é para $n = 906\,150\,257$ (encontrado por Minoru Tanaka em 1980). Portanto, para qualquer inteiro n no intervalo $[2, 906\,150\,256]$ tem-se de facto $I(n) \geq P(n)$.

O PIM também funciona para proposições $P(n)$ com $n \in \{a, a+1, \ldots\}$. Neste caso, o elemento mínimo é a, pelo que basta modificar o passo inicial para "mostrar que $P(a)$ é verdadeira".

Teste. Usando indução matemática, prove:

(a) A fórmula $1 + 2 + 3 + \cdots + (n-1) = \frac{n^2-n}{2}$ que usámos na página 40.

(b) Os números harmónicos $H(n)$ satisfazem a desigualdade $H(2^n) > \frac{n}{2}$.

Em algumas situações pode ser difícil definir um objecto de modo explícito e ser mais fácil defini-lo em função dele próprio. A este processo chama-se *recursão* e pode ser usado para definir sequências, sucessões e conjuntos. Por exemplo, a sequência das potências de 2 pode ser definida explicitamente por $a_n = 2^n$ para $n = 0, 1, 2, \ldots$, mas também pode ser definida recursivamente pelo primeiro termo $a_0 = 1$ e pela regra que permite definir um termo à custa dos anteriores: $a_{n+1} = 2a_n$ para $n = 0, 1, 2, \ldots$.

Portanto, podemos definir uma função sobre os inteiros não negativos

- especificando o valor da função em 0 e
- dando uma regra que permita calcular o seu valor num inteiro a partir dos valores em inteiros menores.

Tal definição chama-se uma *definição recursiva* ou *indutiva*.

Teste (1 minuto). Encontre uma definição recursiva da função factorial

$$f(n) = n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1.$$

As definições recursivas são também utilizadas frequentemente para definir conjuntos. Nesse caso, especifica-se um colecção inicial de elementos como pertencendo ao conjunto que se pretende definir, e depois especificam-se as regras de construção dos elementos do conjunto a partir de elementos que já se sabe estarem no conjunto. Foi o que fizemos, quando logo na Secção 1.1 definimos deste modo o conjunto das fórmulas bem formadas do cálculo proposicional. Os conjuntos definidos deste modo ficam bem definidos e os teoremas sobre eles podem ser provados usando a definição recursiva.

Exemplo. Seja S o conjunto definido recursivamente por

- $3 \in S$,
- Se $x \in S$ e $y \in S$ então $x + y \in S$.

Mostre que S é o conjunto dos inteiros positivos divisíveis por 3. (Assume-se implicitamente neste tipo de definições que um elemento só pertence a S se puder ser gerado usando as duas regras na definição de S.)

Prova. Seja C o conjunto de todos os inteiros positivos divisíveis por 3. Para provar a igualdade $C = S$ temos que verificar as inclusões $C \subseteq S$ e $S \subseteq C$.

$C \subseteq S$: Provemos por indução matemática que todo o inteiro positivo divisível por 3 pertence a S. Para isso seja $P(n)$ a proposição "$3n \in S$". O passo inicial $P(1)$ é verdadeiro pela primeira regra da definição recursiva. Para estabelecer o passo indutivo, assumimos que $P(n)$ é verdadeira, ou seja, que $3n \in S$. Mas então, como 3 também pertence a S, pela segunda regra da definição recursiva, $3n + 3 = 3(n + 1)$ também está em S.

$S \subseteq C$: Basta mostrar que as regras de definição de S só geram elementos que estão contidos em C. A primeira é evidente: $3 \in C$. Quanto à segunda, se $x, y \in S$ são divisíveis por 3 então $x + y$ também é divisível por 3, o que completa a prova. □

1.3. Algoritmos e complexidade

Na matemática discreta abordamos muitos tipos de problemas. Em muitos deles, para chegarmos à solução, temos que seguir um procedimento que, num número finito de passos, conduz à tão desejada solução. A uma tal sequência chama-se algoritmo[6]. Um *algoritmo* é um procedimento para resolver um problema num número finito de passos.

Exemplo. O problema da determinação do maior elemento numa sequência finita de inteiros pode ser facilmente descrito por um algoritmo, formulado em português da seguinte maneira:

- Tome o *máximo temporário* igual ao primeiro inteiro da sequência. (O máximo temporário será o maior inteiro encontrado até ao momento, em cada passo do procedimento.)
- Compare o inteiro seguinte na sequência com o máximo temporário, e se for maior, tome o máximo temporário igual a esse inteiro.
- Repita o passo anterior se existirem mais inteiros na sequência.
- Pare quando chegar ao fim da sequência. O máximo temporário será então o maior inteiro da sequência.

Abreviadamente:

```
procedure max(a_1, a_2, ..., a_n : inteiros)
max := a_1
for i := 2 to n
if max < a_i then max := a_i
{max é o maior elemento}
```

É claro que um algoritmo pode ser formulado explicitamente numa qualquer linguagem de computação, mas nesse caso só poderemos utilizar expressões válidas dessa linguagem. Exemplifiquemos isso convertendo cada linha do algoritmo acima em código `Maple`[7]. Podemos considerar uma lista de números como um vector (matriz). Para usar esta funcionalidade no `Maple` temos primeiro que abrir a package de Álgebra Linear `linalg`:

```
> with(linalg):
Warning, the protected names norm and trace have been redefined and unprotected
```

Continuando:

[6] O termo *algoritmo* deriva do nome *al-Khowarizmi* de um matemático persa do século IX, cujo livro sobre numerais hindus esteve na base da notação decimal moderna que hoje utilizamos.

[7] Se estiver interessado no programa de cálculo simbólico `Maple` recomendamos a leitura do manual *Maple Experiments in Discrete Mathematics* de James Hein (`www.cs.pdx.edu/~jhein/books/MapleLabBook09.pdf`). Alternativamente, recomendamos, na biblioteca do DMUC, o livro *Exploring Discrete Mathematics with Maple* de Kenneth Rosen (o mesmo autor do manual indicado na Bibliografia do curso).

```
> with(linalg):
Warning, the protected names norm and trace have been redefined and unprotected

> Max := proc(t::array)
>   local max, max_temp;
>   max := t[1];
>   for max_temp from 1 to vectdim(t) do
>      if t[max_temp] > max then
>         max := t[max_temp]
>      fi;
>      od;
>      RETURN([max]);
> end:
```

Fica assim traduzido o algoritmo em `Maple`. Dando como input uma qualquer sequência t

```
> t := array(1..10, [1,20,45,3,2,10,99,98,45,32]);
                                    t := [1, 20, 45, 3, 2, 10, 99, 98, 45, 32]
```

basta mandar calcular $Max(t)$:

```
> Max(t);
                                    [99]
```

Em geral, os algoritmos têm características comuns:

- `Entrada (Input)`: conjunto de valores de entrada, definidos num determinado conjunto.
- `Saída (Output)`: a partir de cada conjunto de valores de entrada, um algoritmo produz valores de saída num determinado conjunto. Estes valores de saída contêm a solução do problema.
- `Precisão`: os passos do algoritmo têm que estar definidos com precisão.
- `Finitude`: o algoritmo deve produzir os valores de saída ao cabo de um número finito de passos.
- `Realizável`: deve ser possível realizar cada passo do algoritmo em tempo útil.
- `Generalidade:` o procedimento deve ser aplicável a todos os problemas da forma desejada, e não somente a um conjunto particular de valores de entrada.

Algoritmos recursivos. Encontramos muitas vezes algoritmos recursivos quando pretendemos resolver problemas discretos. Um algoritmo chama-se *recursivo* se resolve um problema reduzindo-o a uma instância do mesmo problema com input mais pequeno.

Uma definição recursiva exprime o valor de uma função num inteiro positivo em termos dos valores da função em inteiros mais pequenos. Isto significa que podemos ter sempre um algoritmo recursivo para calcular o valor de uma função definida por recursão. Por exemplo:

Teste. Especifique um algoritmo recursivo para calcular a potência a^n de um número real a para qualquer inteiro não negativo n.

Solução. A definição recursiva de a^n diz-nos que $a^{n+1} = a \times a^n$ e $a^0 = 1$ (condição inicial). Então podemos fazer:

```
procedure potencia(a : número real ≠ 0, n : inteiro não negativo)
if n = 0 then potencia(a, n) := 1
else potencia(a, n) := a * potencia(a, n − 1)
```

Também não é difícil especificar um algoritmo recursivo para o cálculo do máximo divisor comum de dois inteiros:

```
procedure mdc(a, b : inteiros não negativos com a < b)
if a = 0 then mdc(0, b) := b
else mdc(a, b) := mdc(b mod a, a)
```

Há outro modo de calcular a função potência $f(n) = a^n$ a partir da sua definição recursiva: em vez de reduzir sucessivamente o cálculo a inteiros mais pequenos, podemos começar com o valor da função em 1 e aplicar sucessivamente a definição recursiva para encontrar os valores da função em números sucessivamente maiores. Tal procedimento diz-se *iterativo*. Por outras palavras, para calcular a^n usando um processo iterativo, começamos em 1 e multiplicamos sucessivamente por cada inteiro positivo $\leq n$:

```
procedure potencia iterativa(a : número real ≠ 0, n : inteiro positivo)
x := 1
for i := 1 to n
   x := a * x
{x é a^n}
```

Apresentemos mais um exemplo. A famosa **conjectura de Collatz** (ou conjectura "$3x+1$"), que até hoje ninguém conseguiu provar (!), afirma que se pegarmos num inteiro x arbitrário, e se iterarmos sucessivamente a função

$$f(x) = \begin{cases} \dfrac{x}{2} & \text{se } x \text{ é par} \\ \\ 3x+1 & \text{se } x \text{ é ímpar} \end{cases}$$

chegaremos inevitavelmente ao inteiro 1 ao cabo de um número finito de passos.

```
procedure Collatz(n : inteiro positivo)
x := n
while x ≠ 1 do
   if x mod 2 = 0 then x := x/2
   else x := 3x + 1
```

Procedimento Maple que define a função de Collatz:

```
> Collatz := proc(n::integer)
>    if type(n,even) then
>       n/2;
>    else
>       3*n+1;
>    fi;
> end:
```

Procedimento que itera a função até que o valor obtido seja 1: (incluímos um contador dessas iterações para vermos quanto tempo demoram as iterações a estabilizar no 1; como não temos a certeza de estabilizarem sempre no 1, impomos um limite superior — razoavelmente grande — no número de iterações a efectuar)

```
> IC := proc(raiz::integer)
>    local sentinela, contador;
>    contador := 0;
>    sentinela := raiz;
>    while sentinela <> 1 and contador < 1000^1000 do
>        sentinela := Collatz(sentinela);
>        contador := contador + 1;
>    od;
>    RETURN(contador);
> end:
```

Verificação da conjectura para os primeiros 200 inteiros:

```
> seq(IC(i), i=1..200);
```

```
 0, 1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7, 7, 15, 15, 10,
23, 10, 111, 18, 18, 18, 106, 5, 26, 13, 13, 21, 21, 21, 34, 8, 109, 8, 29, 16, 16,
16, 104, 11, 24, 24, 24, 11, 11, 112, 112, 19, 32, 19, 32, 19, 19, 107, 107, 6, 27,
27, 27, 14, 14, 14, 102, 22, 115, 22, 14, 22, 22, 35, 35, 9, 22, 110, 110, 9, 9, 30,
30, 17, 30, 17, 92, 17, 17, 105, 105, 12, 118, 25, 25, 25, 25, 25, 87, 12, 38, 12,
100, 113, 113, 113, 69, 20, 12, 33, 33, 20, 20, 33, 33, 20, 95, 20, 46, 108, 108,
108, 46, 7, 121, 28, 28, 28, 28, 28, 41, 15, 90, 15, 41, 15, 15, 103, 103, 23, 116,
116, 116, 23, 23, 15, 15, 23, 36, 23, 85, 36, 36, 36, 54, 10, 98, 23, 23, 111, 111,
111, 67, 10, 49, 10, 124, 31, 31, 31, 80, 18, 31, 31, 31, 18, 18, 93, 93, 18, 44,
18, 44, 106, 106, 106, 44, 13, 119, 119, 119, 26, 26, 26, 119, 26
```

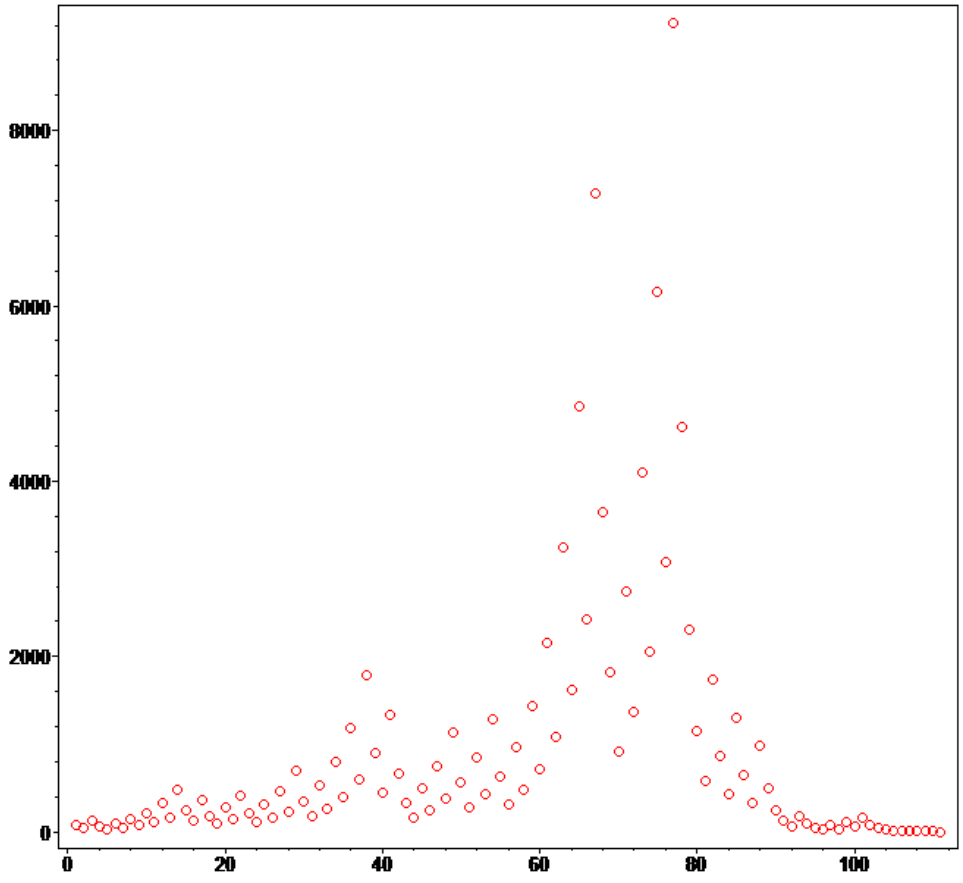
Cada número indica o número de iterações necessárias para levar a função até 1. Por exemplo, para $n = 3$ são necessárias 7 iterações e para $n = 27$ são necessárias 111 iterações. Calculemos explicitamente estas últimas:

Procedimento que calcula a sequência $n, f(n), f(f(n)), \ldots, 1$**:**

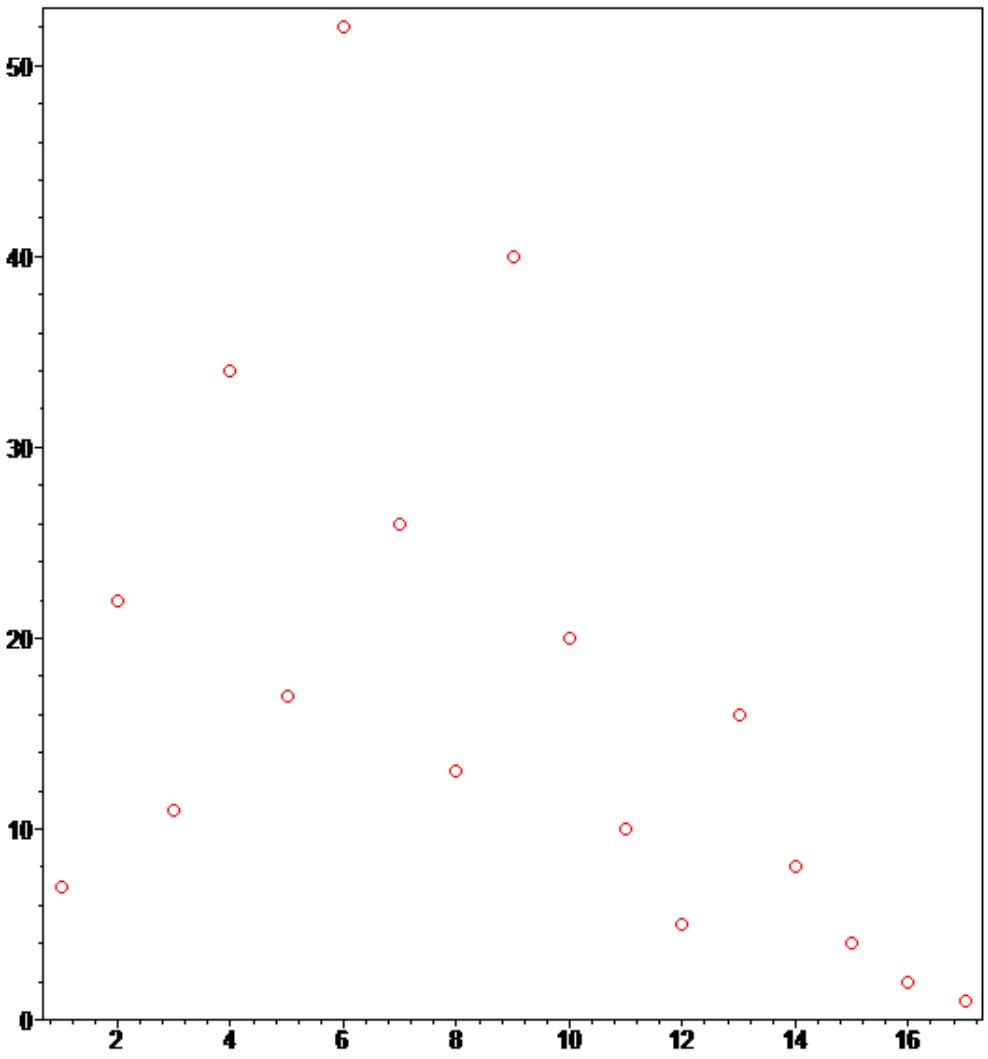
```
> sCollatz := proc(n::integer)
>    local x;
>    x := n;
>    while x<>1 do
>       print(x);
>       x := Collatz(x);
>    od;
> end:
>
> sCollatz(27);
```

```
27 82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412
206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668
334 167 502 251 754 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079
3238 1619 4858 2429 7288 3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308
1154 577 1732 866 433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160
80 40 20 10 5 16 8 4 2 1
```

Desde o valor inicial 27 até 1 a função atinge um pico na 77^a iteração com o valor 9232. Graficamente:



$n = 14$:



Eficiência de algoritmos. Além de produzir uma solução satisfatória e precisa para o problema que pretende resolver, um algoritmo tem que ser eficiente (em termos de velocidade de execução). Um dos objectivos da *algoritmia* consiste em medir a eficiência de algoritmos. Muitas vezes dispomos de diferentes algoritmos que resolvem correctamente o problema, mas algum poderá ser mais eficiente que os outros. Uma medida de eficiência será, claro, o tempo dispendido por um computador para resolver o problema executando o algoritmo.

Seja P um problema e A um algoritmo para resolver P. O *tempo de execução* de A pode ser analisado contando o número de determinadas operações que são efectuadas durante a sua execução. Esta contagem pode depender do tamanho do input.

Exemplos. (1) No problema da determinação do elemento máximo de uma sequência com n elementos, será natural tomar como medida de eficiência o número de comparações entre elementos, que dependerá evidentemente de n. Calculemos esse número. Para encontrar o elemento máximo, o máximo temporário começa por ser igual ao termo inicial da sequência. Em seguida, depois de uma comparação ter sido feita para verificar que o final da sequência ainda não foi atingido, o máximo temporário é comparado com o segundo termo da sequência, actualizando o máximo temporário para este valor, caso seja maior. O procedimento continua, fazendo mais duas comparações no passo seguinte. Como são feitas duas comparações em cada um dos passos (desde o segundo termo da sequência até ao último) e mais uma comparação é feita para sair do ciclo (quando $i = n + 1$), são feitas ao todo

$$2(n-1)+1 = 2n-1$$

comparações.

(2) Se P consiste em verificar se um determinado objecto pertence a uma dada lista, será também natural contar o número de comparações efectuadas por A, que dependerá do tamanho da lista.

(3) Para resolver o problema da *ordenação de listas* existem diversos algoritmos de ordenação, entre os quais o chamado *algoritmo da inserção*. A ideia por detrás deste algoritmo consiste em dividir a lista L que se pretende ordenar em duas sublistas. A sublista L_1 inclui os elementos de L já ordenados e a sublista L_2, que é um sufixo da lista inicial, inclui os elementos de L ainda não analisados. Cada passo do algoritmo consiste na inserção do primeiro elemento de L_2 ordenadamente na lista L_1 e, claro, na sua remoção da lista L_2. O algoritmo inicia-se com

$$L_1 = \{Primeiro[L]\} \text{ e } L_2 = Resto[L]$$

e termina quando $L_2 = \varnothing$. No caso dos algoritmos de ordenação é típico tomar-se como medida de eficiência o número de comparações entre elementos. Claro que o número de comparações depende da lista dada inicialmente.

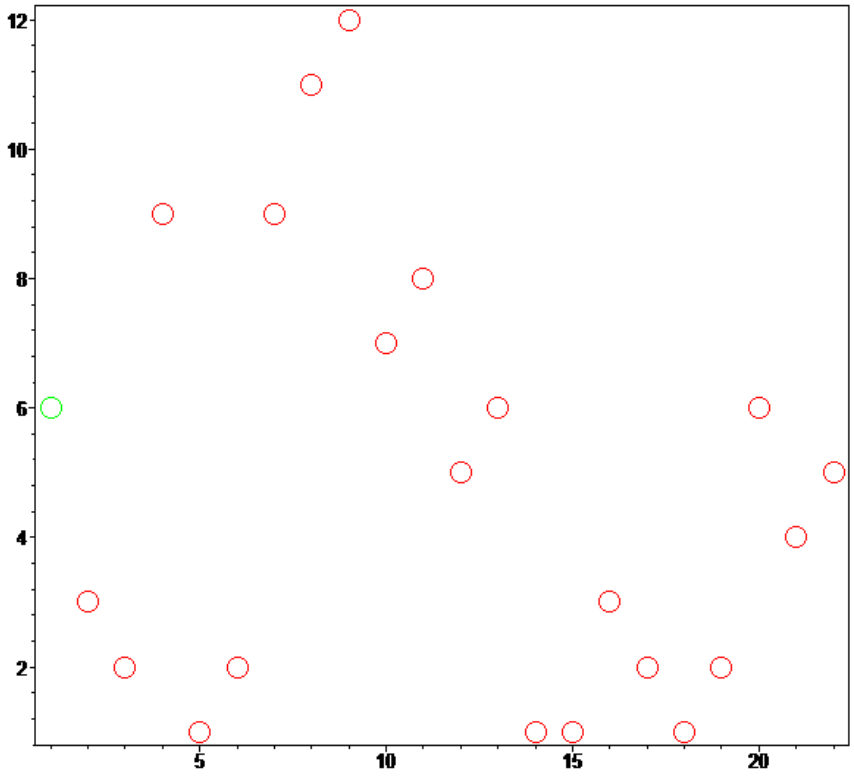
Implementemos este algoritmo no `Maple`. Começamos por definir uma função auxiliar `plotperm` que representa graficamente uma lista de números. As bolas verdes correspondem a elementos já ordenados (ou seja, elementos da sublista L_1), enquanto que as vermelhas correspondem a elementos ainda não ordenados, da sublista L_2. Para tal importamos o pacote `plots`.

```
> with(plots);
> plotperm := proc(w::list,i::integer)
>   local c, n, g_1, g_2;
>   c := nops(w);
>   if i-1 <> c then
>      g_1 := pointplot({seq([n,w[n]],n=1..i-1)},symbolsize=30,color=green):
       g_2 := pointplot({seq([n,w[n]],n=i..c)},symbolsize=30,color=red):
>      display([g_1,g_2],axes=boxed,symbol=circle);
>   else
>      g_1 := pointplot({seq([n,w[n]],n=1..c)},symbolsize=30,color=green):
       display([g_1],symbolsize=30,color=green,axes=boxed,symbol=circle);
>   fi;
> end:
```

Fazendo, por exemplo,

```
> plotperm([6,3,2,9,1,2,9,11,12,7,8,5,6,1,1,3,2,1,2,6,4,5],2);
```

obtemos a seguinte figura:



Com esta função gráfica definida, podemos agora implementar o algoritmo da inserção de modo a obtermos uma visão dinâmica da ordenação. Para evitar o trabalho de escrever listas de números como inputs, podemos aproveitar a função `randperm` do Maple que gera permutações aleatórias dos números $\{1, 2, \ldots, n\}$. Para isso importamos o pacote `combinat` de Combinatória.

```
> with(combinat,randperm);
```

[*randperm*]

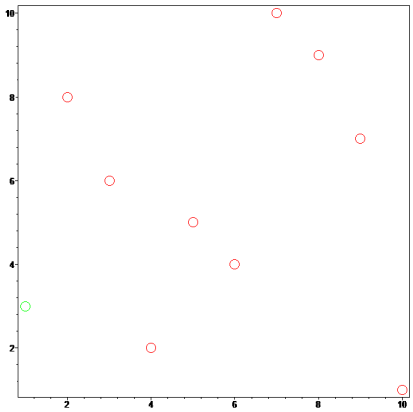
```
> inserc := proc(w::list)
>   local c,v,i,j,m;
>   i := 2;
>   v := w;
>   c := nops(w);
>   plotperm(v,i);
```

```
>   while i <= c do
>     j := i-1;
>     m := v[i];
>     while (j > 0 and v[j] > m) do
>       v[j+1] := v[j];
>       j := j-1;
>    end do;
>    v[j+1] := m;
>    print(plotperm(v,i));
>    print(v);
>    i := i+1;
>   end do;
>  plotperm(v,c+1);
>  end:
```

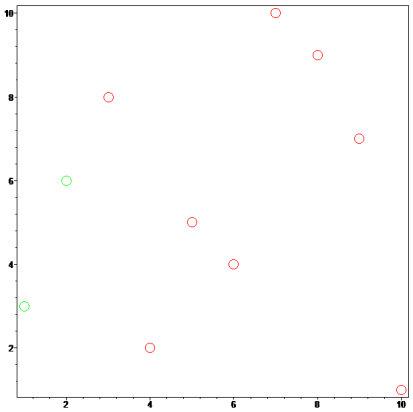
Agora, com a instrução

```
> inserc(randperm(10));
```

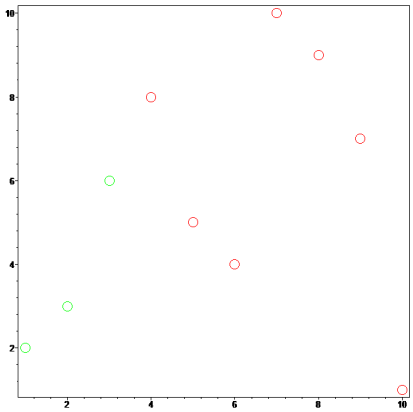
fazemos correr o procedimento de ordenação de uma permutação aleatória dos números $1, 2, \ldots, 10$:



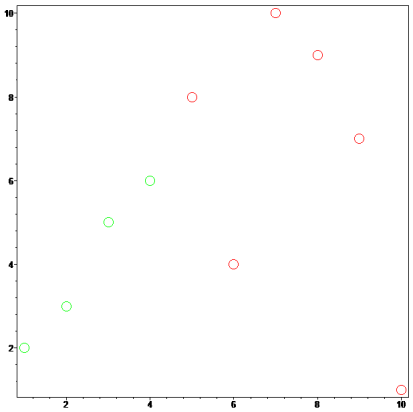
[3, 8, 6, 2, 5, 4, 10, 9, 7, 1]



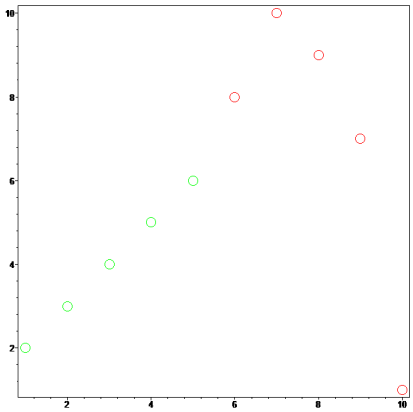
[3, 6, 8, 2, 5, 4, 10, 9, 7, 1]



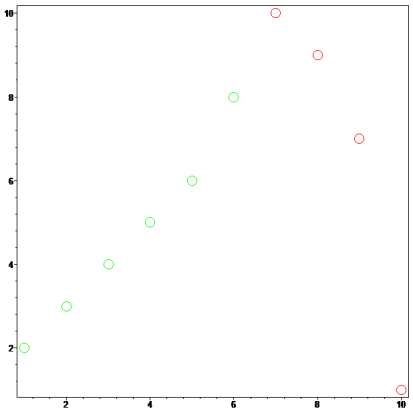
[2, 3, 6, 8, 5, 4, 10, 9, 7, 1]



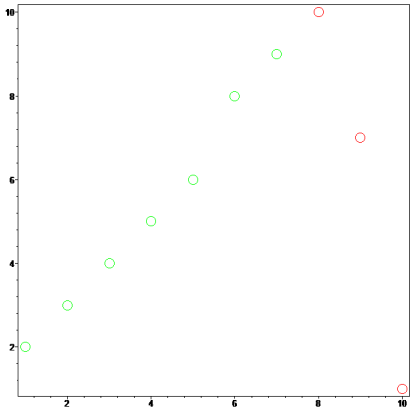
[2, 3, 5, 6, 8, 4, 10, 9, 7, 1]



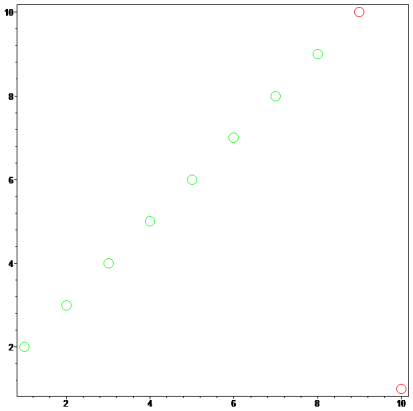
[2, 3, 4, 5, 6, 8, 10, 9, 7, 1]



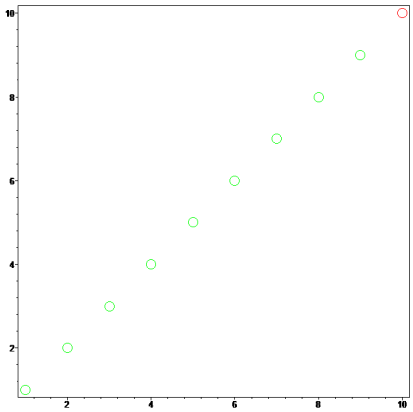
[2, 3, 4, 5, 6, 8, 10, 9, 7, 1]



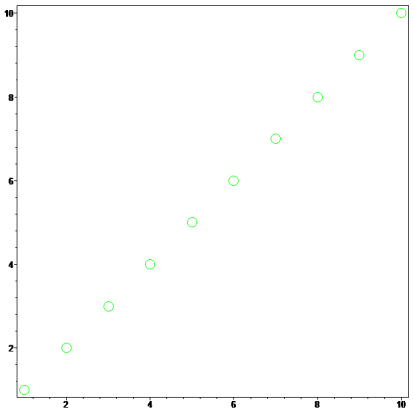
[2, 3, 4, 5, 6, 8, 9, 10, 7, 1]



[2, 3, 4, 5, 6, 7, 8, 9, 10, 1]



[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Em todos estes exemplos, a análise da eficiência baseia-se na contagem do número de comparações a realizar, o que obviamente depende do tamanho da lista. Para determinar essa eficiência é costume considerar dois casos:

- pior situação (ou seja, situação mais desfavorável dos dados)
- situação média.

Um input no caso da pior situação é um input que leva A a executar o maior número de operações. No caso da determinação do elemento máximo de uma sequência (exemplo (1)), fixado o comprimento n da sequência, a pior situação acontece para qualquer lista de números (pois o número de comparações é constante, igual a $2n-1$). No exemplo (2) a pior situação será uma lista que não contém o objecto procurado.

No caso do algoritmo de inserção (exemplo (3)), a pior situação é quando a lista dada está ordenada por ordem inversa.

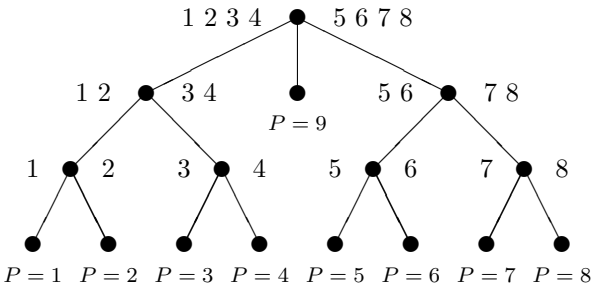
A análise na situação média obriga a considerações probabilísticas pois é necessário atribuir a cada situação uma probabilidade. Muitas vezes considera-se que as situações têm todas a mesma probabilidade (a distribuição das mesmas é então uniforme).

Estudemos um pouco o caso da pior situação. Seja $T_A(n)$ o tempo de execução máximo de A para inputs de tamanho n. A função T_A chama-se a *função da pior situação* para A. Um algoritmo A para resolver um problema $\mathcal{P}$ diz-se *optimal na pior situação* se qualquer algoritmo B que resolve $\mathcal{P}$ satisfaz $T_A(n) \leq T_B(n)$ para qualquer $n \in \mathbb{N}$.

Uma *árvore de decisão* para um algoritmo é uma árvore cujos nós representam pontos de decisão no algoritmo e cujas folhas representam os resultados.

Teste. Dado um conjunto de nove moedas, uma das quais é mais pesada que as outras, use uma balança de dois pratos (sem pesos) para determinar a moeda mais pesada.

Solução. Denotemos as moedas por $1, 2, \ldots, 9$ (e por P a mais pesada). Podemos fazer a seguinte sequência de pesagens (cada nó interno representa uma pesagem; os números de cada lado de cada nó interno representam os conjuntos de moedas colocados em cada prato da balança, na respectiva pesagem; $P = 1, P = 2, \ldots, P = 9$ são os 9 resultados possíveis):



Esta árvore tem *profundidade* três[8], o que significa que, neste algoritmo, o caso da pior situação são 3 pesagens.

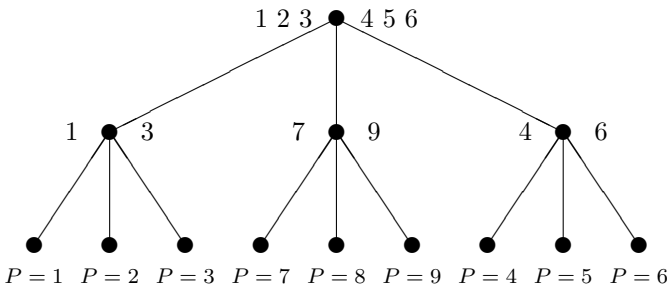
Será o algoritmo optimal na pior situação? Em cada pesagem pode acontecer uma de três coisas: o prato da esquerda está mais pesado, o prato da direita está mais pesado ou os pratos estão equilibrados. Portanto, neste tipo de problemas com balanças, as árvores de decisão são

[8] A *profundidade* de uma árvore é o comprimento do maior caminho desde a raiz da árvore até às folhas. Na árvore em questão, desde a raiz até às folhas há caminhos com um ramo (no caso da folha $P = 9$) ou três ramos (nas restantes folhas). Mais tarde, quando estudarmos os grafos, estudaremos as árvores com mais cuidado.

ternárias (em cada nó haverá no máximo três ramos). Uma árvore ternária de profundidade d tem, no máximo, 3^d folhas. Como há 9 possíveis resultados, então

$$3^d \geq 9, \text{ isto é, } d \geq \log_3 9 = 2.$$

Portanto, poderá haver, eventualmente, algum algoritmo cuja árvore tenha profundidade 2. E, de facto, há:



Pela discussão acima podemos agora concluir que este é um algoritmo optimal na pior situação (2 pesagens).

Teste. Dado um conjunto de nove moedas, uma das quais é defeituosa (mais pesada ou mais leve que as outras), determine um algoritmo optimal na pior situação para balanças de dois pratos (sem pesos) que determine a moeda defeituosa e dê como output se a moeda é mais pesada ou mais leve.

Solução. Comecemos por determinar um minorante para a profundidade da árvore de decisão. Denotemos as moedas por $1, 2, \ldots, 9$ e usemos a letra P quando a moeda defeituosa for mais pesada e a letra L caso contrário. Existem assim 18 resultados possíveis:

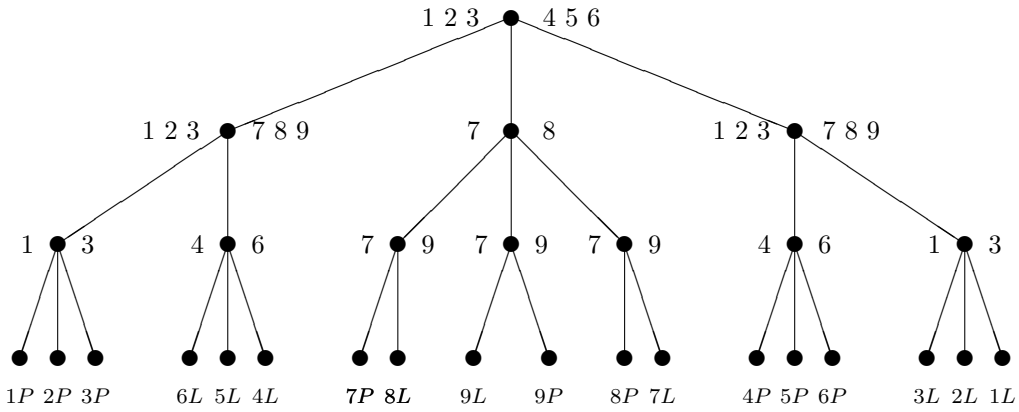
$$1P,\ 1L, \ldots, 9P, 9L.$$

Uma árvore de decisão ternária com profundidade d terá no máximo 3^d folhas, donde $3^d \geq 18$, isto é,

$$d \geq \log_3 18 \geq \lceil \log_3 18 \rceil = 3.$$

(A função $\lceil - \rceil : \mathbb{R} \to \mathbb{N}$ nos números reais é a chamada função *tecto* (*ceiling*) e aplica cada número real x no menor inteiro $\lceil x \rceil$ tal que $\lceil x \rceil \geq x$. De modo análogo, podemos definir a função *chão* (*floor*) $\lfloor - \rfloor : \mathbb{R} \to \mathbb{N}$ que a cada real x faz corresponder o maior inteiro $\lfloor x \rfloor \leq x$. Por exemplo, $\lfloor \frac{1}{2} \rfloor = 0$, $\lceil \frac{1}{2} \rceil = 1$, $\lfloor -\frac{1}{2} \rfloor = -1$, $\lceil -\frac{1}{2} \rceil = 0$.)

Portanto, qualquer algoritmo que resolva o problema terá sempre que efectuar pelo menos 3 pesagens. No algoritmo seguinte esse valor é igual a 3:



Portanto, este é um algoritmo optimal para a pior solução.

O `Maple` tem algumas ferramentas que permitem medir a *performance* de um algoritmo. Nomeadamente, com a função `time(-)` que indica a hora actual, é fácil medir o tempo de CPU (Unidade de Processamento Central) que uma função leva a calcular um resultado:

```
> ha := time():
> Funcaoqualquer(x):
> time() - ha;
```

Por exemplo, se definirmos uma função `Oper` que efectua diversas operações consecutivas, é possível medirmos o tempo de execução dessas operações:

```
> Oper := proc(x)
>   local a,b,c,d,e;
>   a := x;
>   b := x^2;
>   c := x^3;
>   d := x!;
>   e := x^x;
> end:
> ha := time():
> Oper(100000):
> time() - ha;
                3.157
```

Se substituirmos $x = 100000$ por $x = 300000$ já o output será `23.094` (segundos).

O `Maple` também permite registar o número de adições, multiplicações e funções efectuadas, por meio da função `cost` da package `codegen`:

```
> with(codegen,cost):
> cost(a^4 + b + c + (d!)^4 + e^e);
```

$$5\ additions + 8\ multiplications + 3\ functions$$

Vamos agora usar estas funções para comparar dois algoritmos que calculam o valor de um polinómio $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ num ponto específico c, ou seja, o número $p(c) = a_0 + a_1c + a_2c^2 + \cdots + a_nc^n$. Os valores de entrada são o número c e a lista de coeficientes $a_0, a_1, a_2, \ldots, a_n$ do polinómio.

```
> Polinomio := proc(c::float, coef::list)
>   local potencia, i, y;
>   potencia := 1;
>   y := coef[1];
>   for i from 2 to nops(coef) do
>     potencia := potencia*c;
>     y := y + coef[i] * potencia;
>   od;
>   RETURN(y);
> end:

> Horner := proc(c::float, coef::list)
>   local i, y;
>   y := coef[nops(coef)];
>   for i from nops(coef)-1 by -1 to 1 do
>     y := y * c + coef[i];
>   od;
>   RETURN(y);
> end:
```

Por exemplo, para o polinómio $p(x) = 4 + 3x + 2x^2 + x^3$, o valor de $p(5)$ é igual a 194:

```
> input_lista := [4,3,2,1];
```

$$input\_lista := [4, 3, 2, 1]$$

```
> Polinomio(5.0, input_lista);
```

$$194.000$$

```
> Horner(5.0, input_lista);
```

$$194.000$$

De modo a testarmos um algoritmo contra o outro precisamos de gerar listas de coeficientes. O comando seguinte gera aleatoriamente um polinómio de grau 2000:

```
> p2000 := randpoly(x,degree=2000,dense):
```

e se fizermos

```
> q2000 := subs(x=1,convert(p2000,list)):
```

obtemos a lista dos coeficientes correspondentes, que podemos usar como input nos algoritmos `Polinomio` e `Horner`. Agora, usando as ferramentas para medir o tempo de execução, obtemos:

```
> ha := time():
> Horner(104567890000000.0, q2000);
```

$$0.3913255222\ 10^{27971}$$

```
> time() - ha;
```

$$0$$

```
> ha := time():
> Polinomio(104567890000000.0, q2000);
```

$$0.3913255222\ 10^{27971}$$

```
> time() - ha;
```

$$0$$

Experimentando polinómios de grau superior obtemos os tempos de execução seguintes:

	4000	6000	8000
`Polinomio`	0.031	0.047	0.063
`Horner`	0	0.015	0.031

Podemos assim concluir que o método de Horner de cálculo polinomial é marginalmente mais rápido que o método tradicional da substituição da indeterminada x pelo valor onde queremos calcular a função polinomial.

Voltemos ao primeiro algoritmo `Max` desta secção (determinação do elemento máximo de uma sequência). Na altura verificámos que para uma sequência de comprimento n, o algoritmo efectua $2n-1$ comparações. Usando a chamada *notação assimptótica* da seguinte definição, diz-se que o número de comparações no pior caso para este algoritmo é $O(n)$.

Definição. Diz-se que uma função $f : \mathbb{N} \to \mathbb{R}$ é *da ordem de* $g : \mathbb{N} \to \mathbb{R}$, o que se denota por $f(n) = O(g(n))$ se existe uma constante real c e um $k \in \mathbb{N}$ tais que

$$|f(n)| \leq c\,|g(n)|$$

para $n \geq k$.

Teste. Mostre que qualquer função polinomial p com coeficientes reais, dada por $p(n) = a_t n^t + a_{t-1} n^{t-1} + \cdots + a_1 n + a_0$, é da ordem de q onde $q(n) = n^t$.

No algoritmo `Max`, tomando $f(n) = 2n-1$ então $f(n) = O(n)$: basta tomar $c = 2$ e $k = 1$, pois $2n - 1 \leq 2n$ para qualquer $n \in \mathbb{N}$. Portanto, o algoritmo `Max` tem complexidade $O(n)$, ou seja, *linear*, na pior situação, atendendo à seguinte classificação:

Terminologia para a complexidade de algoritmos	
Complexidade	**Terminologia**
$O(1)$	Complexidade constante
$O(\log n)$	Complexidade logarítmica
$O(n)$	Complexidade linear
$O(n \log n)$	Complexidade $n \log n$
$O(n^b)$	Complexidade polinomial
$O(b^n)$, onde $b > 1$	Complexidade exponencial
$O(n!)$	Complexidade factorial

Qual é a complexidade do algoritmo de inserção (de ordenação de listas)? No pior caso, onde a lista está ordenada por ordem inversa, para cada i é necessário fazer $i-1$ comparações. Como i varia de 2 até ao comprimento n da lista, temos que o número de comparações no pior caso é igual a

$$\sum_{i=2}^{n}(i-1) = 1+2+3+\cdots+(n-1),$$

ou seja, é dado pela soma dos $n-1$ primeiros números naturais (termos de uma progressão aritmética de razão 1):

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

Poderíamos ter calculado este somatório com o auxílio do `Maple`:

```
> sum('(i-1)', 'i'=2..n);
```

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

Este cálculo mostra que o `Maple`, embora seja muito bom a realizar cálculo simbólico, é somente um programa de computador que não é tão inteligente quanto um ser humano normal pode ser. Por vezes necessita da nossa orientação para fazer o cálculo da forma mais simples. Para isso temos a instrução `simplify`, que permite simplificar a expressão obtida:

```
> simplify(sum('(i-1)', 'i'=2..n));
```

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

Demonstrámos, assim, o seguinte resultado:

Proposição. *O número de comparações na pior situação para o algoritmo da inserção é* $O(n^2)$. *Tem assim complexidade polinomial.*

Façamos agora um exemplo de análise do caso médio no caso do algoritmo da inserção.

Suponhamos que queremos inserir um elemento na parte da lista já ordenada (a sublista L_1 com $i-1$ elementos) de modo que a lista resultante com i elementos esteja ordenada. Potencialmente, existem i posições onde esse elemento pode ser colocado (para além das $i-1$ ocupadas pela lista dada, ainda existe a possibilidade do novo elemento ser maior que todos os outros). Vamos impor uma hipótese probabilística que consiste em assumir que todas as posições são equiprováveis para colocar o novo elemento. Isto é, cada posição tem probabilidade $1/i$.

O número de comparações necessárias se o novo elemento tiver que ser colocado na posição $k \geq 2$ é $i-k+1$ e na posição $k=1$ é $i-1$. Logo *o número médio de comparações para introduzir o novo elemento numa das* i *posições é dado pelo somatório*

$$\left(\sum_{k=2}^{i} \frac{1}{i}(i-k+1)\right) + \frac{1}{i}(i-1).$$

Utilizando o sistema `Maple`, podemos calcular este somatório:

```
> sum('(i-k+1)/i', 'k'=2..i) + (i-1)/i;
```

$$i - 1 + \frac{3(i+1)}{2i} - \frac{(i+1)^2}{2i} - \frac{1}{i} + \frac{i-1}{i}$$

Simplificando, obtemos

```
> simplify(%);
```

$$\frac{i^2+i-2}{2i}$$

(A instrução `%` permite-nos fazer referência ao último cálculo evitando a sua reescrita.)
Logo, o número médio de comparações para que a lista fique ordenada é dado pelo somatório

$$\sum_{i=2}^{n} \left(\frac{1}{2} - \frac{1}{i} + \frac{i}{2}\right).$$

Calculemo-lo:

```
> simplify(sum('(i^2+i-2)/(2*i)', 'i'=2..n));
```

$$\frac{n^2}{4} + \frac{3n}{4} - \Psi(n+1) - \gamma$$

Aqui teremos que consultar o menu Help do `Maple`: γ é a chamada constante de Euler (vale aproximadamente $0.5772156649\ldots$) e $\Psi(n+1)+\gamma$ é o chamado *número harmónico* $H(n)$ de

ordem n. Os números harmónicos são a versão discreta do logaritmo natural e são utilizados em diversos contextos. O n-ésimo número harmónico $H(n)$ é igual a $\sum_{i=1}^{n} \frac{1}{i}$, para qualquer natural n. Para o que nos interessa, basta reter o seguinte resultado (note que log denota o logaritmo na base 2):

Proposição. *Para todo o natural n tem-se*

$$\frac{\log(n)}{2} < H(n) \leq \log(n) + 1.$$

Note que, embora $H(n)$ tenda para infinito, cresce apenas de forma logarítmica.

Teste. Mostre que $H(n) = O(\log(n))$.

Como estamos a ver, ao determinar as medidas de eficiência de algoritmos surgem somatórios, às vezes não triviais, para calcular. Portanto, se quisermos fazer análise de algoritmos temos que saber calcular somatórios (e, neste caso concreto, precisamos de saber um pouco mais sobre números harmónicos). Estudaremos, em seguida, algumas técnicas de cálculo de somatórios.

Proposição. *Seja I um conjunto finito. Os somatórios gozam das seguintes propriedades*:

(1) *Distributividade:* $\sum_{i\in I} c\, a_i = c \sum_{i\in I} a_i$.

(2) *Associatividade:* $\sum_{i\in I}(a_i + b_i) = \sum_{i\in I} a_i + \sum_{i\in I} b_i$.

(3) *Comutatividade:* $\sum_{i\in I} a_i = \sum_{i\in I} a_{p(i)}$ *para qualquer bijecção (permutação)* $p : I \to I$.

(4) *Progressão constante:* $\sum_{i\in I} c = c\,|I|$.

(5) *Aditividade dos índices:* $\sum_{i\in I} a_i + \sum_{i\in J} a_i = \sum_{i\in(I\cup J)} a_i + \sum_{i\in(I\cap J)} a_i$ *(sendo J um conjunto finito também).*

(6) *Mudança de variável:* $\sum_{i\in I} a_{f(i)} = \sum_{j\in J} a_j$, *para qualquer função bijectiva* $f : I \to J$*; mais geralmente, para qualquer função* $f : I \to J$, $\sum_{i\in I} a_{f(i)} = \sum_{j\in J} \big(a_j \cdot \#(f^{-1}(\{j\}))\big)$.

Teste. Mostre que $\sum_{0\leq i<n}(a_{i+1} - a_i)b_i = a_n b_n - a_0 b_0 - \sum_{0\leq i<n} a_{i+1}(b_{i+1} - b_i)$.

Vejamos alguns exemplos de aplicação destas propriedades.

Exemplo: progressão aritmética de razão r. Provemos que

$$\boxed{\sum_{i=0}^{n}(a + ri) = \left(a + \frac{1}{2}rn\right)(n+1) = (2a + rn)\frac{n+1}{2} = [a + (a + rn)]\frac{n+1}{2}}$$

$$\sum_{i=0}^{n}(a+ri) = \sum_{i=0}^{n}(a+r(n-i)) \qquad \text{(por comutatividade, com } p(i)=n-i\text{)}$$

$$\Leftrightarrow \quad \sum_{i=0}^{n}(a+ri) = \sum_{i=0}^{n}(a+rn-ri)$$

$$\Leftrightarrow \quad 2\sum_{i=0}^{n}(a+ri) = \sum_{i=0}^{n}(a+rn-ri) + \sum_{i=0}^{n}(a+ri)$$

$$\Leftrightarrow \quad 2\sum_{i=0}^{n}(a+ri) = \sum_{i=0}^{n}((a+rn-ri)+(a+ri)) \qquad \text{(por associatividade)}$$

$$\Leftrightarrow \quad 2\sum_{i=0}^{n}(a+ri) = \sum_{i=0}^{n}(2a+rn)$$

$$\Leftrightarrow \quad 2\sum_{i=0}^{n}(a+ri) = (2a+rn)\sum_{i=0}^{n}1 \qquad \text{(por distributividade)}$$

$$\Leftrightarrow \quad 2\sum_{i=0}^{n}(a+ri) = (2a+rn)(n+1) \qquad \text{(por progressão constante)}$$

$$\Leftrightarrow \quad \sum_{i=0}^{n}(a+ri) = (a+\frac{1}{2}rn)(n+1).$$

Podemos ainda usar o `Maple` para verificar este resultado

```
> simplify(sum('(a+r*i)', 'i'=0..n));
```

$$an + a + \frac{1}{2}rn^2 + \frac{1}{2}rn$$

```
> factor(%)
```

$$\frac{(n+1)(rn+2a)}{2}$$

e os passos da prova:

```
> evalb( factor(sum('(a+r*i)', 'i'=0..n)) = factor(sum('(a+r*(n-i))', 'i'=0..n)));
```

$$true$$

etc.

Alternativamente, se tivermos já na mão a prova da fórmula $\sum_{i=1}^{n-1} i = \frac{n^2-n}{2}$ (é o caso $a = 0, r = 1$) que usámos anteriormente, podemos simplificar muito a demonstração do caso geral:

$$\begin{aligned}
\sum_{i=0}^{n}(a+ri) &= \sum_{i=0}^{n} a + \sum_{i=0}^{n} ri && \text{(por associatividade)}\\
&= a\sum_{i=0}^{n} 1 + r\sum_{i=0}^{n} i && \text{(por distributividade)}\\
&= a\sum_{i=0}^{n} 1 + r\left(\sum_{i=1}^{n-1} i + n\right) && \text{(por progressão constante)}\\
&= a(n+1) + r\left(\frac{n^2-n}{2} + n\right)\\
&= a(n+1) + r\left(\frac{n^2+n}{2}\right).
\end{aligned}$$

Exemplo: progressão geométrica de razão r. Provemos que

$$\boxed{\sum_{i=0}^{n} ar^i = \frac{ar^{n+1}-a}{r-1}}$$

$$\begin{aligned}
&\sum_{i=0}^{n} ar^i + ar^{n+1} = \sum_{i=0}^{n+1} ar^i && \text{(por aditividade com } I=\{0,\ldots,n\} \text{ e } J=\{n+1\})\\
\Leftrightarrow\ &\sum_{i=0}^{n} ar^i + ar^{n+1} = ar^0 + \sum_{i=1}^{n+1} ar^i && \text{(por aditividade com } I=\{0\} \text{ e } J=\{1,\ldots,n+1\})\\
\Leftrightarrow\ &\sum_{i=0}^{n} ar^i + ar^{n+1} = ar^0 + \sum_{i=0}^{n} ar^{i+1} && \text{(por mudança de variável com } I=\{0,\ldots,n\})\\
\Leftrightarrow\ &\sum_{i=0}^{n} ar^i + ar^{n+1} = a + r\sum_{i=0}^{n} ar^i && \text{(por distributividade)}\\
\Leftrightarrow\ &\sum_{i=0}^{n} ar^i = \frac{ar^{n+1}-a}{r-1}.
\end{aligned}$$

Leituras suplementares. Como vimos, muitos problemas algorítmicos relativamente simples requerem por vezes o cálculo de somatórios um pouco complicados. Não se conhecem métodos gerais que permitam resolver qualquer somatório, a maior parte das vezes os problemas têm que ser atacados de forma *ad hoc* (esta é uma das características de muitas áreas da matemática discreta: a não existência de métodos gerais de resolução que obrigam uma abordagem *ad hoc* ao problema; aqui o conhecimento e a destreza na manipulação dos diversos métodos particulares de resolução, que poderão só funcionar em alguns casos, é crucial).

Nestas observações finais indicaremos muito resumidamente algumas das técnicas mais elegantes e importantes para resolver somatórios.

Método 1: Método *ad hoc*. Calculando as primeiras somas parciais do somatório (a_1+a_2, a_1+a_2+a_3, etc.), é por vezes possível adivinhar a correspondente fórmula geral. A ilustração deste método em muitos exemplos interessantes pode ser vista e experimentada (interactivamente) no módulo *Somatórios*[9] na página da disciplina.

A fórmula deverá depois ser confirmada com uma prova formal, para termos a certeza da sua validade. Essa prova pode ser feita de modo análogo como fizemos nalguns exemplos acima, usando as propriedades dos somatórios que enunciámos, ou, mais facilmente, pelo método de indução matemática estudado na secção anterior.

Método 2: Método da perturbação. A ideia por detrás deste método é a seguinte: tentar obter duas expressões diferentes que tenham o mesmo valor, "perturbando" levemente a soma a calcular. Um exemplo ilustra o funcionamento deste método:

Suponhamos que pretendemos calcular o valor de $q(n) = \sum_{i=1}^{n} i^2$. Vamos perturbar levemente a sua definição e tentar escrever $q(n+1)$ de duas formas diferentes. Por um lado, $q(n+1) = q(n) + (n+1)^2$ e, por outro lado, por uma mudança de variável,

$$\begin{aligned} q(n+1) &= \sum_{i=0}^{n} (i+1)^2 \\ &= \sum_{i=0}^{n} (i^2 + 2i + 1) \\ &= \sum_{i=0}^{n} i^2 + 2\sum_{i=0}^{n} i + \sum_{i=0}^{n} 1 \\ &= q(n) + 2\sum_{i=0}^{n} i + (n+1). \end{aligned}$$

Comparando ambos os resultados obtemos

$$q(n) + (n+1)^2 = q(n) + 2\sum_{i=0}^{n} i + (n+1),$$

ou seja,

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

[9] `www.mat.uc.pt/~picado/ediscretas/somatorios`.

Parece que não fizemos muitos progressos! Limitámo-nos a obter a soma $\sum_{i=1}^{n} i$ (note também que temos aqui uma prova formal, rigorosa, da fórmula para $\sum_{i=1}^{n-1} i$ que utilizámos anteriormente, na página 40). Mas isto sugere imediatamente o seguinte: se perturbando um pouco a soma $q(n)$ dos quadrados conseguimos obter uma fórmula para $\sum_{i=1}^{n} i$, será que perturbando a soma $c(n) = \sum_{i=1}^{n} i^3$ dos cubos conseguimos uma fórmula para $q(n)$?

A fórmula de recorrência de $c(n)$ é $c(n+1) = c(n) + (n+1)^3$ e, por outro lado, por uma mudança de variável,

$$\begin{aligned} c(n+1) &= \sum_{i=0}^{n} (i+1)^3 \\ &= \sum_{i=0}^{n} (i^3 + 3i^2 + 3i + 1) \\ &= \sum_{i=0}^{n} i^3 + 3\sum_{i=0}^{n} i^2 + 3\sum_{i=0}^{n} i + \sum_{i=0}^{n} 1 \\ &= c(n) + 3q(n) + \frac{3}{2}n(n+1) + (n+1). \end{aligned}$$

Igualando ambas as expressões, obtemos

$$c(n) + (n+1)^3 = c(n) + 3q(n) + \frac{3}{2}n(n+1) + (n+1),$$

ou seja,

$$\begin{aligned} (n+1)^3 = 3q(n) + \frac{3}{2}n(n+1) + (n+1) &\Leftrightarrow 3q(n) = (n+1)^3 - \frac{3}{2}n(n+1) - (n+1) \\ &\Leftrightarrow q(n) = \frac{2(n+1)^3 - 3n(n+1) - 2(n+1)}{6} \\ &\Leftrightarrow q(n) = \frac{2n^3 + 6n^2 + 6n + 2 - 3n^2 - 3n - 2n - 2}{6} \\ &\Leftrightarrow q(n) = \frac{2n^3 + 3n^2 + n}{6} \\ &\Leftrightarrow q(n) = \frac{1}{6}n(n+1)(2n+1). \end{aligned}$$

Em `www.mat.uc.pt/~picado/ediscretas/somatorios/Matematica_sem_palavras_files/soma_quadrados.html` pode ver uma "prova" geométrica, sem palavras, desta fórmula.

Método 3: Método do integral. Este método consiste em aproximar o somatório por um integral. Por exemplo, para calcular $q(n) = \sum_{i=0}^{n} i^2$, aproximamos $q(n)$ por $\int_0^n x^2\,dx$ que no Maple se obtém da seguinte maneira:

```
> int('x^2', 'x'=0..n);
```

$$\frac{n^3}{3}$$

Analisemos agora o erro $e(n) = q(n) - \frac{n^3}{3}$ desta aproximação:

$$\begin{aligned} e(n) &= q(n-1) + n^2 - \frac{n^3}{3} \\ &= q(n-1) - \frac{(n-1)^3}{3} + n^2 - \frac{n^3}{3} + \frac{(n-1)^3}{3} \\ &= e(n-1) + n - \frac{1}{3}. \end{aligned}$$

De modo análogo, podemos concluir que $e(n-1) = e(n-2) + (n-1) - \frac{1}{3}$. Portanto,

$$e(n) = 0 + \sum_{i=1}^{n} (i - \frac{1}{3}) = \frac{(n+1)n}{2} - \frac{n}{3}.$$

Consequentemente,

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

Coincide com o resultado calculado anteriormente pelo método da perturbação?

```
> simplify(n^3/3+(n+1)*n/2-n/3);
```

$$\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

```
> factor(n^3/3+n^2/2+n/6);
```

$$\frac{n(n+1)(2n+1)}{6}$$

Sim, coincide, claro!