1.2. 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[4]. 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 $max(a_1, a_2, \ldots, a_n$: 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`[5]. 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
```

[4] 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.

[5] Para se familiarizar com o programa de cálculo simbólico `Maple` recomendamos a resolução da Ficha 2-P na aula prática, acompanhada da leitura do manual *Maple Experiments in Discrete Mathematics* de James Hein, cujo ficheiro pdf pode encontrar na Bibliografia da disciplina (`www.mat.uc.pt/~picado/ediscretas/apontamentos.html`). 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).

Continuando:

```
> 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.

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.

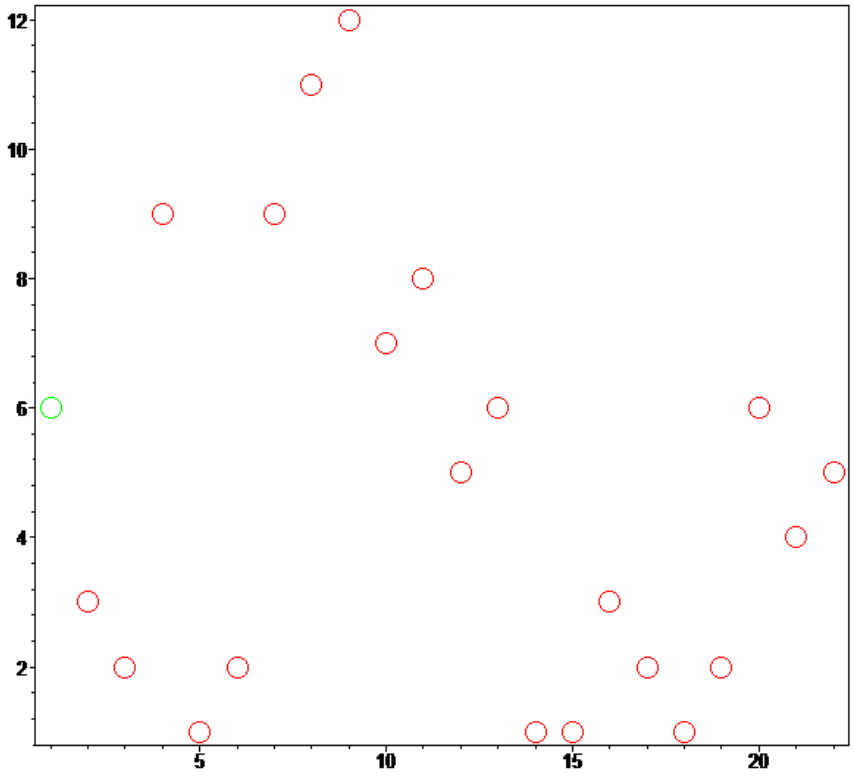
Implementemos este algoritmo no `Maple`. Começamos por definir uma função auxiliar `plotperm` que representa graficamente uma lista de números. As bolas verdes correspondem a elementos já ordenados (ou seja, elementos da sublista L_1), enquanto que as vermelhas correspondem a elementos ainda não ordenados,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

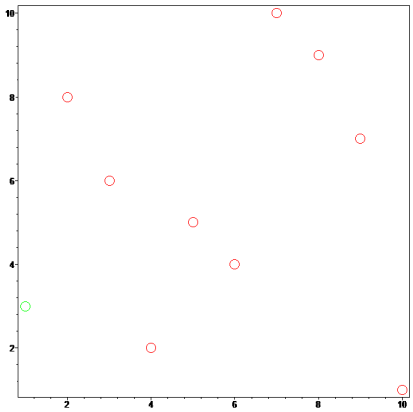
```
> inserc := proc(w::list)
>   local c,v,i,j,m;
>   i := 2;
>   v := w;
>   c := nops(w);
>   plotperm(v,i);
```

```
>   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:
```

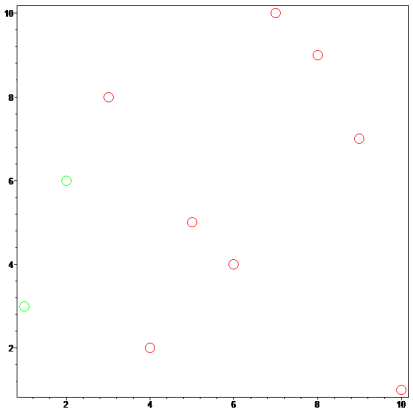
Agora, com a instrução

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

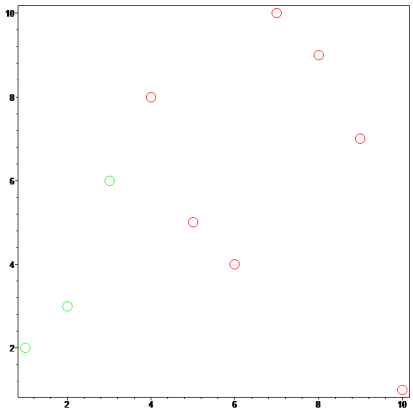
fazemos correr o procedimento de ordenação de uma permutação aleatória dos números $1, 2, \ldots, 10$:



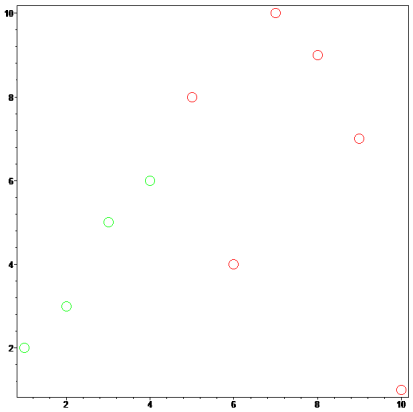
[3, 8, 6, 2, 5, 4, 10, 9, 7, 1]



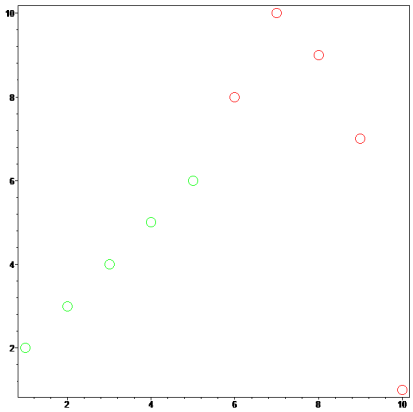
[3, 6, 8, 2, 5, 4, 10, 9, 7, 1]



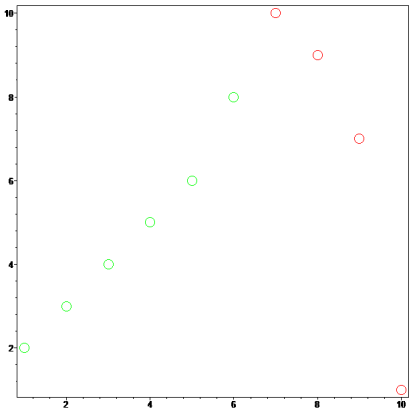
[2, 3, 6, 8, 5, 4, 10, 9, 7, 1]



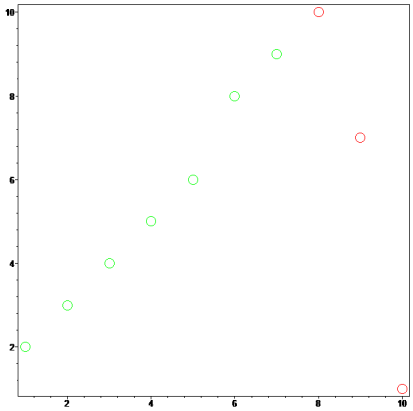
[2, 3, 5, 6, 8, 4, 10, 9, 7, 1]



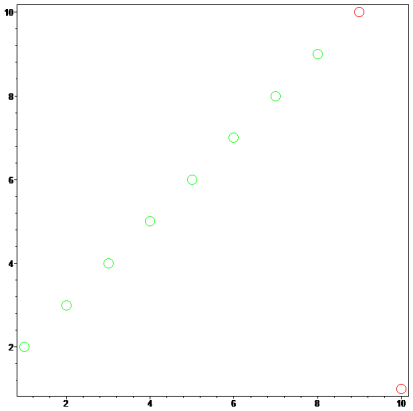
[2, 3, 4, 5, 6, 8, 10, 9, 7, 1]



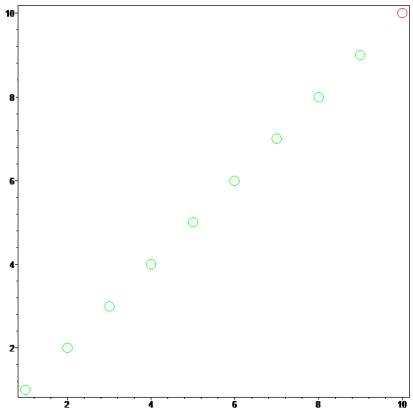
[2, 3, 4, 5, 6, 8, 10, 9, 7, 1]



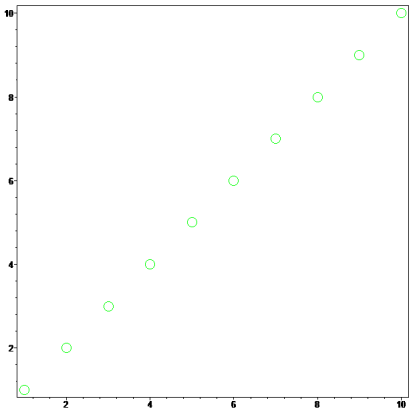
[2, 3, 4, 5, 6, 8, 9, 10, 7, 1]



[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.

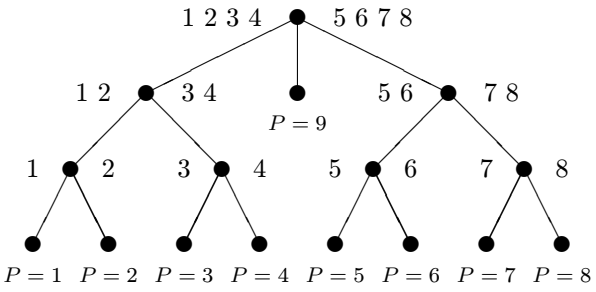
A análise na situação média obriga a considerações probabilísticas pois é necessário atribuir a cada situação uma probabilidade. Muitas vezes considera-se que as situações têm todas a mesma probabilidade (a distribuição das mesmas é então uniforme).

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[6], o que significa que, neste algoritmo, o caso da pior situação são 3 pesagens.

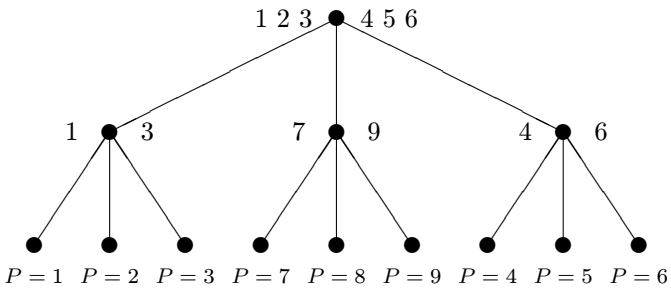
Será o algoritmo optimal na pior situação? Em cada pesagem pode acontecer uma de três coisas: o prato da esquerda está mais pesado, o prato da direita está mais pesado ou os pratos estão equilibrados. Portanto, neste tipo de problemas com balanças, as árvores de decisão são

[6] 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:

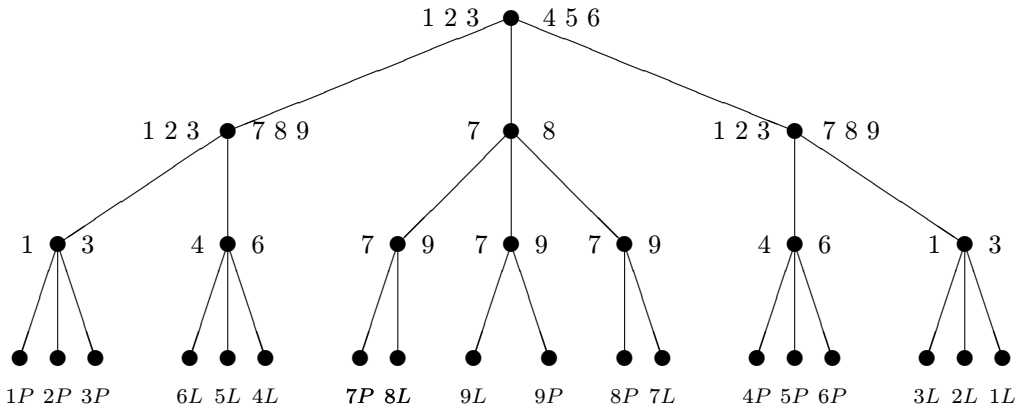
$$1P,\ 1L, \ldots, 9P, 9L.$$

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 criam por vezes a necessidade de calcular 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*[7] 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 que apresentaremos em pormenor no Capítulo 1.4.

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,

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

[7] `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!