

## Módulo 2.2 - Processos e os recursos computacionais que consomem

Um procedimento é um pedaço de texto que pode ser lido, mas que nada faz! Por outro lado, a chamada de um procedimento gera um processo no interior do computador que, normalmente, produzirá resultados, obviamente à custa dos recursos computacionais que consome.

Neste módulo é mostrado que os *procedimentos recursivos* geram *processos recursivos* ou então *processos iterativos* e, conforme o caso, são normalmente muito diferentes os recursos computacionais que consomem. São também apresentadas as *recursividades linear* e *em árvore*, e como exemplo desta última surge a sequência de Fibonacci. Para avaliar o comportamento dos processos face à *dimensão do problema* que resolvem, introduz-se a *Ordem de Crescimento* ou *Ordem de Complexidade* e apresentam-se exemplos de ordem de crescimento *constante*, *linear*, *exponencial* e *logarítmica*.

Na resolução dos exercícios propostos, sugere-se o uso do Scheme.

### Palavras-Chave

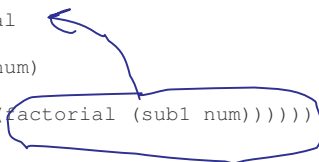
procedimento recursivo, processo recursivo, processo iterativo, recursividade em cauda (*tail recursion*), recursos computacionais de tempo e espaço de memória, recursividade linear, recursividade em árvore, nó, raiz, folha, ramos, árvore binária, sequência de Fibonacci, permutações de  $n$  elementos, arranjos de  $n$  elementos  $k$  a  $k$ , combinações de  $n$  elementos  $k$  a  $k$ , ordem de crescimento ou ordem de complexidade, dimensão do problema, ordem de crescimento constante, linear, exponencial e logarítmica.

### Procedimento recursivo chama-se a si próprio!

Saberá facilmente se um procedimento é recursivo, verificando no corpo desse procedimento a existência de alguma chamada para si próprio, como acontece no procedimento *factorial*.

```
(define factorial
  (lambda (num)
    (if (zero? num)
        1
        (* num (factorial (sub1 num))))))

> (factorial 4)
24
>
```



Espaço para testar o procedimento *factorial*.

O ecrã abre com este procedimento e com o diálogo indicado.

Em *factorial*, experimente substituir `(sub1 num)` por `(- num 2)` e explique o que acontece.

### Processo recursivo suspende actividade e cativa memória

Um procedimento ao ser chamado cria um *processo* que executará a tarefa definida naquele procedimento.

Por exemplo, uma chamada do procedimento *factorial* cria um processo que toma um inteiro como argumento, processa-o e no final devolve o factorial desse inteiro. Pela definição do procedimento *factorial* percebe-se que uma chamada deste desencadeia uma sequência de processos, até se atingir o *caso base*. O que se segue corresponde à sequência de processos criados pela chamada `(factorial 4)`, sequência que vai crescendo até ao caso base, correspondente à chamada `(factorial 0)`. Atingido este ponto, verifique a morte dos processos da sequência, mas na ordem inversa à sua criação.

```

(factorial 4) = (* 4 (factorial 3))
              = (* 4 (* 3 (factorial 2)))
              = (* 4 (* 3 (* 2 (factorial 1))))
              = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))  <-- caso base

              = (* 4 (* 3 (* 2 (* 1 1))))
              = (* 4 (* 3 (* 2 1)))
              = (* 4 (* 3 2))
              = (* 4 6)
              = 24

```

A chamada `(factorial 4)` cria um processo que provoca a chamada `(factorial 3)` e fica suspenso até receber a resposta desta última chamada. Por sua vez, a chamada `(factorial 3)` cria também um processo que chama `(factorial 2)` e assim sucessivamente vão sendo criados outros processos até se atingir o *caso base*, `(factorial 0)`. Verifique a existência de uma cadeia de multiplicações suspensas, o que significa que esta cadeia vai progressivamente ocupando memória do computador.

A partir do *caso base*, as multiplicações vão sendo calculadas, a expressão vai reduzindo de tamanho, e a memória vai sendo libertada.

Pode facilmente deduzir que os processos gerados pelas chamadas do procedimento *factorial* gastam recursos computacionais (*tempo de cálculo* e *memória*) que são proporcionais ao valor do argumento. Quanto maior for o valor do argumento maior será o tempo de cálculo e maior será o espaço de memória ocupado. Não será difícil aceitar que uma chamada `(factorial 8)` gaste aproximadamente o dobro do tempo e da memória que é gasto pela chamada `(factorial 4)`.

*Os recursos de tempo e espaço gastos por (factorial 8) correspondem ao dobro dos recursos gastos por (factorial 4)*  
*1- mostre que assim deve ser.*

*E agora um pouco mais difícil...*  
*2- e se lhe disser que esta previsão de gastos é mais correcta para valores elevados do argumento.*  
*Concorda com esta afirmação?*

Esta característica, que consiste em diferir ou suspender a execução das operações, guardando-as em memória, até encontrar o *caso base*, é típica dos chamados *Processos Recursivos*.

*Já consegue distinguir um procedimento do respectivo processo?*

Convém aqui frisar a diferença entre *procedimento* e *processo*. O procedimento é o próprio texto, uma entidade estática que se pretende suficientemente legível, e que expressa uma ideia numa certa linguagem. O processo é uma entidade dinâmica que existe no interior do computador, gasta recursos, mas fornece resultados.

### Processo iterativo gerado por procedimento recursivo!

Vamos encarar o cálculo do factorial de uma forma diferente, que permitirá definir um novo procedimento diferente de *factorial*. O novo procedimento cria um processo com outras características das indicadas para *factorial*, mas no final continuará a fornecer o resultado pretendido, ou seja, o factorial de *n*.


Em vez da definição recursiva, utilizada em *factorial*  $n! = n * (n-1)!$   
 agora a ideia a explorar baseia-se em  $n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$

A implementação desta nova ideia resume-se a uma sequência de multiplicações, *n* que multiplica por *(n-1)*, o respectivo resultado multiplicado por *(n-2)* e assim sucessivamente até que se encontre *n*

= 0, que já não será multiplicado. Esta implementação baseia-se na existência de um *acumulador* que inicialmente toma o valor 1 (*valor neutro da multiplicação*). Seguidamente, o valor do *acumulador* será multiplicado por  $n$ , resultado que vai actualizar o valor do *acumulador*. Este processo repete-se com  $n-1$ ,  $n-2$ , e assim sucessivamente até  $n=0$ , altura em que o valor de factorial de  $n$  se encontra no *acumulador*.

Tomando (factorial 4) como exemplo,  $n$  vai começar com o valor 4 e *acumulador* com o valor 1.

iteração	$n$	acumulador	acumulador (no final da iteração)
1ª	4	1	$4 * 1 = 4$
2ª	3	4	$3 * 4 = 12$
3ª	2	12	$2 * 12 = 24$
4ª	1	24	$1 * 24 = 24$
5ª	0	24	



O *acumulador* é actualizado, no final de cada iteração, com o resultado do seu próprio valor multiplicado por  $n$ . Este  $n$  vai sendo reduzido de 1 em cada iteração, até se atingir o *caso base*,  $n = 0$ .

Podemos agora escrever e testar o procedimento *factorial-novo*, baseado na ideia acabada de expor, introduzindo um segundo parâmetro designado por *acumulador*.

```
(define factorial-novo
  (lambda (n acumulador)
    (if (zero? n)
        acumulador                ; caso base, em que n=0...
        (factorial-novo (sub1 n) (* n acumulador)))) ; caso geral

> (factorial-novo 4 1)
24
```

Espaço para testar o procedimento *factorial-novo*.

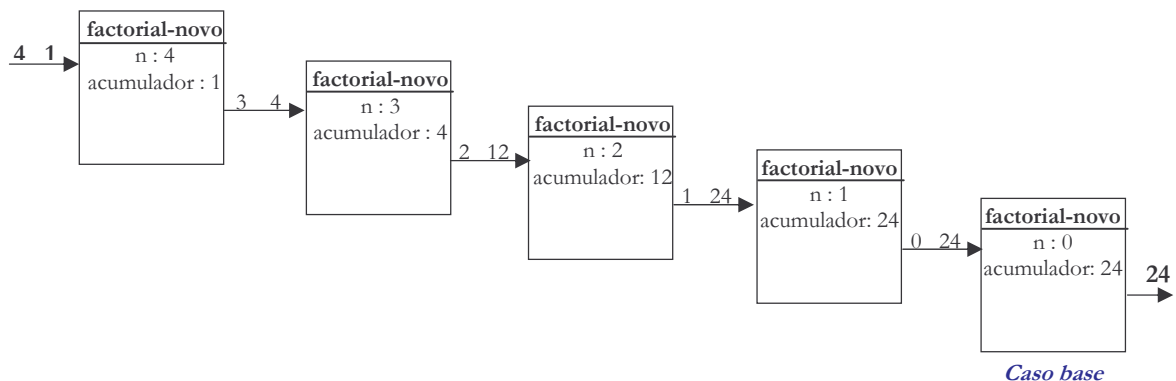
O ecrã abre com o procedimento a testar.

Experimente dar ao segundo argumento valores diferentes de 1 e justifique os resultados.

Analise agora como se desenvolve a chamada (factorial-novo 4 1), seguindo a definição de *factorial-novo*.

(factorial-novo 4 1)	como $n$ não é 0, resulta	(factorial-novo 3 (* 4 1))
(factorial-novo 3 4)	como $n$ não é 0, resulta	(factorial-novo 2 (* 3 4))
(factorial-novo 2 12)	como $n$ não é 0, resulta	(factorial-novo 1 (* 2 12))
(factorial-novo 1 24)	como $n$ não é 0, resulta	(factorial-novo 0 (* 1 24))
(factorial-novo 0 24)	como $n = 0$ , resulta	24, valor no acumulador

Na representação gráfica do mesmo exemplo, facilmente se observa que os sucessivos processos criados não suspendem a actividade (como acontece com os processos recursivos), pois não ficam à espera de uma resposta. Limitam-se a lançar um processo idêntico a eles próprios, mas com outros argumentos, sempre na direcção do *caso base*, mas morrem de seguida.



Esta característica, que consiste em não suspender a execução das operações, como acontecia nos *processos recursivos*, é típica dos chamados *processos iterativos*.

### Processo iterativo menos consumidor de recursos que o processo recursivo

O procedimento *factorial-novo* continua a ser um procedimento recursivo, pois encontrará no seu corpo uma chamada a si próprio. Verifique, contudo, que uma chamada deste procedimento, por exemplo, `(factorial-novo 4 1)`, cria um processo que não fica à espera de qualquer resposta. Este processo provoca a chamada `(factorial 3 4)`, antes de morrer. Este modo de funcionamento continua com `(factorial 2 12)`, depois com `(factorial 1 24)` e, finalmente, com `(factorial 0 24)` que, por corresponder ao *caso base*, responde devolvendo o valor 24. Contrariamente ao que acontecia com *factorial*, o procedimento *factorial-novo* vai executando as multiplicações à medida que aparecem, o que implica um gasto constante em memória (apenas a memória para guardar *num* e *acumulador*).

Pode deduzir que os processos gerados pelas chamadas do procedimento *factorial-novo* gastam recursos constantes em memória e recursos em tempo de máquina proporcionais ao valor do argumento. Ou seja, o processo gerado pela chamada `(factorial-novo 8 1)` deve gastar aproximadamente o dobro do tempo e a mesma memória que são gastos pelo processo gerado pela chamada `(factorial-novo 4 1)`.

*Mostre que no procedimento factorial-novo*

*1- o gasto em memória é constante (contrariamente ao que acontecia com o procedimento factorial, cujo gasto em memória era proporcional a n)*

*2- e que o gasto em tempo é proporcional a n (como acontecia com o procedimento factorial)*

O procedimento *factorial-novo* gera *processos iterativos* que, em geral, são menos consumidores de recursos computacionais que os correspondentes *processos recursivos*, como os gerados por *factorial*.

Não parece muito correcto fazer chamadas a procedimentos que calculam factoriais, fornecendo dois argumentos: o número de que se pretende conhecer o factorial e o valor que iniciliza o acumulador, normalmente o valor 1. Para ultrapassar este tipo de situação, pode definir o procedimento *factorial-iter*, baseado em *factorial-novo*.

```

(define factorial-iter
  (lambda (num)
    (factorial-novo num 1)))

```

### Forma expedita de identificar o tipo de processo gerado por um procedimento

Reconhecer o tipo de processo (recursivo ou iterativo) gerado por um procedimento recursivo surge assim como uma necessidade importante para quem programa, uma vez que para a mesma tarefa os recursos consumidos podem ser muito diferentes, num e noutro caso.

*Em que situação se torna praticamente indiferente usar procedimentos que criam processos recursivos ou iterativos?*

Compare cuidadosamente os procedimentos *factorial* e *factorial-novo*, ou mais especificamente o caso geral ou passo recursivo destes procedimentos, pois irá encontrar uma forma expedita de reconhecer se um procedimento recursivo gera processos recursivos ou iterativos.

Em *factorial*, que gerava processos recursivos, o caso geral corresponde a uma multiplicação

```
(* n (factorial (sub1 n)))
```

em que um dos operandos é a chamada recursiva *(factorial (sub1 n))*. Como os operandos são calculados em primeiro lugar, significa que a multiplicação terá que ficar suspensa até que se conheça o valor correspondente a *n* e a *(factorial (sub1 n))*. Assim, o processo correspondente à execução da multiplicação entra em suspensão, é mantido em memória, ao mesmo tempo que é lançado um novo processo correspondente a *(factorial (sub1 n))*. Estamos claramente face à criação de uma sequência de *processos recursivos* que vão ficando suspensos, até se chegar ao caso base.

*Analisando o caso geral do procedimento recursivo factorial, já sabe como identificar que ele gera uma sequência de processos recursivos?*

Por outro lado, em *factorial-novo*, que gera processos iterativos, o caso geral corresponde a uma chamada recursiva

```
(factorial-novo (sub1 n) (* n acumulador))
```

Agora, a multiplicação encontra-se num dos argumentos da chamada recursiva e terá, por este motivo, de ser calculada em primeiro lugar. Só depois dos argumentos calculados é que a chamada recursiva terá lugar, sendo assim o último passo do procedimento. Esta recursividade, em que a chamada recursiva se encontra na *cauda* do procedimento, é designada por *recursividade em cauda* (*tail recursion*).

Na *recursividade em cauda*, quando um novo processo é criado, o anterior não fica suspenso, pois morre e assim não exige espaço de memória. Esta situação cria uma sequência de *processos iterativos* que termina com o caso base, mas estes processos não ficam suspensos como acontecia com os processos recursivos.

*Analisando o caso geral do procedimento recursivo factorial-novo, já sabe como identificar que ele gera uma sequência de processos iterativos?*

Em termos de recursos de memória, os procedimentos recursivos que geram processos iterativos (*recursividade em cauda*, como se verifica em *factorial-novo*) são mais eficientes que os procedimentos recursivos que geram processos recursivos (como em *factorial*). Pelo menos, assim acontece em Scheme, cujo interpretador, segundo norma do IEEE, deve apresentar uma implementação *recursividade em cauda* (*tail-recursive*), em que um processo iterativo pode ser gerado por um procedimento recursivo. Noutras linguagens, os processos iterativos deverão ser gerados a partir de estruturas de repetição: *do*, *repeat*, *for*, *while*, e não através de procedimentos recursivos.

Em termos de tempo de cálculo, o comportamento dos processos iterativos pode ser menos consumidor do que nos correspondentes processos recursivos. Não é o caso dos procedimentos *factorial* e *factorial-novo*, pois em ambos os recursos em tempo são semelhantes, ou seja, proporcionais ao valor do argumento.

Em situações em que os recursos computacionais não constituem uma grande preocupação, o melhor será recorrer a procedimentos recursivos que geram processos recursivos, pois, normalmente, baseiam-se em ideias muito mais simples e mais fáceis de implementar do que os procedimentos recursivos que geram processos iterativos. Isto pode confirmar-se analisando as ideias que estiveram na base das soluções encontradas para *factorial* e *factorial-novo*, bem como as que foram utilizadas em exemplos que se seguem.

### Exemplo 1

Baseando-se numa solução recursiva, escrever um procedimento que toma um argumento  $n$ , inteiro e positivo, e gera um processo recursivo para calcular  $n + (n-1) + \dots + 1 + 0$ .

Neste exemplo, o *caso base* é  $n = 0$ , a *operação de redução* é *sub1* aplicada a  $n$ , e o *caso geral* é adicionar  $n$  à soma dos restantes números da série, suposta conhecida. Da tradução desta ideia em *Scheme* resulta o procedimento *soma-sequencia*.

```
(define soma-sequencia
  (lambda (n)
    (if (zero? n)
        0
        (+ n
           (soma-sequencia (sub1 n))))))

> (soma-sequencia 3)
6
```

Reconhece-se nesta solução a geração de processos recursivos, pois, no caso geral ou passo recursivo, a chamada recursiva é um dos operandos da expressão, não sendo a última operação do procedimento. Isto significa que as operações de adição vão ficar suspensas e guardadas em memória, até que se encontre o caso base.

Para o mesmo problema, tenta-se agora um procedimento que executa a mesma tarefa, mas gerando processos iterativos. Com este objectivo, introduz-se um acumulador para guardar o resultado das sucessivas adições, acumulador que na chamada do procedimento será inicializado com zero, o elemento neutro da adição.

```
(define soma-sequencia-aux
  (lambda (n acumulador)
    (if (zero? n)
        acumulador
        (soma-sequencia-aux (sub1 n)
                             (+ n acumulador)))))
```

Para esconder o parâmetro *acumulador*, poderia ser utilizado o procedimento *soma-sequencia-iter*.

```
(define soma-sequencia-iter
  (lambda (numero)
    (soma-sequencia-aux numero 0)))
```

Espaço para testar os procedimentos *soma-sequencia* e *soma-sequencia-iter*.  
O ecrã abre com os dois procedimentos a testar e o procedimento auxiliar.  
Experimente o procedimento auxiliar dando ao segundo argumento valores diferentes de 0 e justifique os resultados.

### Exercício 1

Vamos agora inventar uma operação que designaremos de *misturar-dois-inteiros*. Em que consistirá a referida operação? Analise o diálogo que se segue.

```
> (misturar-dois-inteiros 123 789)
718293

> (misturar-dois-inteiros 12345 89)
1238495

> (misturar-dois-inteiros 89 12345)
1234859
```

- a- Escreva em Scheme um procedimento recursivo para misturar dois inteiros, o qual deverá gerar processos recursivos.
- b- Escreva um segundo procedimento para fazer a mesma coisa, mas gerando processos iterativos.

Pista: Será necessário, na alínea b-, utilizar um procedimento auxiliar com um terceiro parâmetro para servir de acumulador?

Espaço para desenvolver e testar os dois procedimentos pedidos.  
O ecrã abre vazio.

### Exercício 2 - Um desafio muito mais difícil...

Vamos agora inventar uma operação que designaremos de *misturar-dois-numeros*. Em que consistirá a referida operação? Analise o diálogo que se segue.

```
> (misturar-dois-numeros 123 789)
718293

> (misturar-dois-numeros 12345.67 89.1)
1238495.167

> (misturar-dois-numeros 89.1 12345.67)
1234859.617
```

- a- Escreva em Scheme um procedimento recursivo para misturar dois números, o qual deverá gerar processos recursivos.
- b- Escreva um segundo procedimento para fazer a mesma coisa, mas gerando processos iterativos.

Espaço para desenvolver e testar os dois procedimentos pedidos.  
O ecrã abre vazio.

### Exemplo 2

Escreva em *Scheme* o procedimento *troca-posi* que espera um único argumento, inteiro positivo, e visualiza-o com os dígitos na ordem inversa, e com o carácter "." entre eles.

```
> (troca-posi 123)
3.2.1
> (troca-posi 5)
5
```

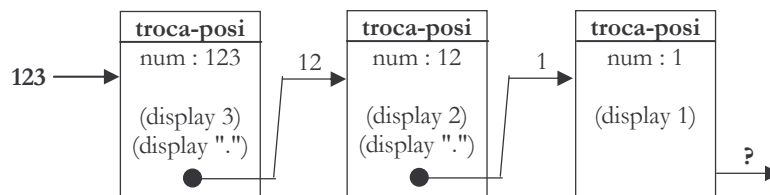
A solução que se apresenta baseia-se em dois procedimentos auxiliares, *digito-menos-significativo* e *retira-digito-menos-significativo*. O caso base corresponde a um número menor que 10, representado por um único dígito. O caso geral traduz-se pela visualização do dígito menos significativo seguido de um "." e finalizando como uma chamada recursiva a *troca-posi*, com um argumento equivalente ao actual retirado o dígito menos significativo.

```

(define troca-posi
  (lambda (num)
    (cond
      ((< num 10) ; caso base
       (display num))
      (else ; caso geral
       (display (digito-menos-significativo num))
       (display ".")
       (troca-posi (retira-digito-menos-significativo num))))))

```

Neste procedimento, é utilizada *recursividade em cauda*, pois a última operação é a chamada recursiva. Assim, o processo gerado é iterativo, o que se comprova com a representação gráfica da chamada (troca-posi 123). Também se verifica, pela representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã.



Espaço para desenvolver e testar os procedimentos *digito-menos-significativo* e *retira-digito-menos-significativo* e testar o procedimento *troca-posi*.  
 O ecrã abre com o procedimento *troca-posi*.  
 No procedimento *troca-posi*, experimente substituir 10 por 100 e justifique os resultados.

### Exemplo 3

Escreva em *Scheme* o procedimento *nao-troca-posi* que espera um único argumento, inteiro positivo, e visualiza-o com os dígitos na ordem normal, e com o carácter "." entre eles.

```

> (nao-troca-posi 123)
1.2.3
> (nao-troca-posi 5)
5

```

A solução que se apresenta parece idêntica à utilizada para *troca-posi*, do exemplo anterior, com umas trocas de posição de algumas instruções, mas uma análise mais cuidada leva a concluir que o processo agora gerado tem um comportamento muito diferente. Neste caso, não aparece *recursividade em cauda*, ou seja, os processos gerados não são iterativos, mas sim recursivos, uma vez que a chamada recursiva não é o último passo de *nao-troca-posi*. Como pode verificar, esta chamada é seguida por duas chamadas a *display*. A execução dos *display* vai ser suspensa e a morte do processo vai ser adiada, enquanto que outro processo idêntico é criado. E isto acontece até se atingir o caso base.

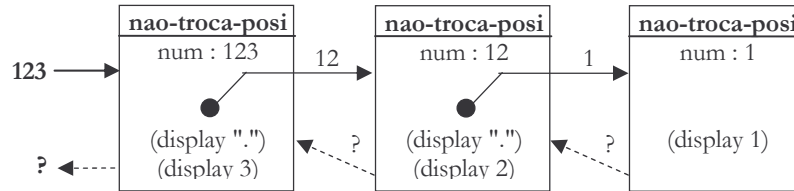
```

(define nao-troca-posi
  (lambda (num)
    (cond
      ((< num 10)
       (display num))
      (else (nao-troca-posi (retira-digito-menos-significativo num))
            (display ".")
            (display (digito-menos-significativo num))))))

```



Verá, nesta representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã. A primeira visualização a acontecer situa-se no processo mais à direita, (display 1).

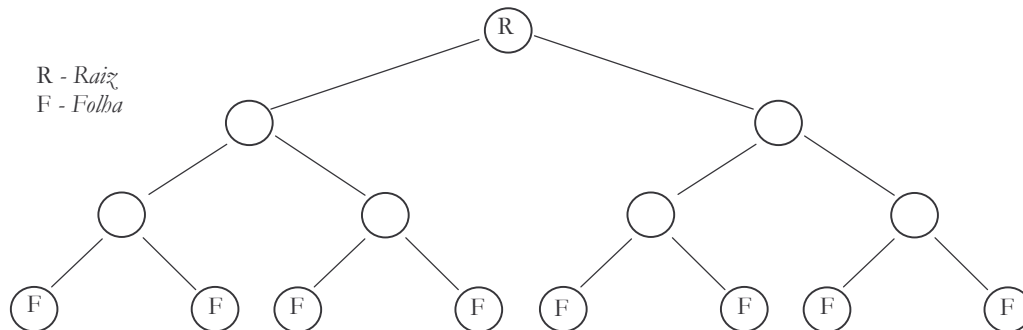


Espaço para testar o procedimento *nao-troca-posi*.  
O ecrã abre com o procedimento a testar.

### Recursividade linear e em árvore

No caso geral ou passo recursivo dos procedimentos *factorial* e *factorial-novo*, era presente apenas uma chamada recursiva dos respectivos procedimentos. Por exemplo, no passo recursivo de *factorial*, (\* n (factorial (sub1 n))), a chamada recursiva existente, (factorial (sub1)), é única. A isto se chama *Recursividade linear*. As várias representações gráficas da sequência dos processos gerados são bem prova desta linearidade.

Alguns casos, que serão analisados nesta secção, usam no passo recursivo duas ou mais chamadas do procedimento, ao que se chama *Recursividade em árvore* (*Tree recursion* ou *multinway recursion*), designação que advém da forma típica das figuras utilizadas para esboçar a sequência dos processos que criam, ou seja, árvore em posição invertida, com o tronco para cima e a copa para baixo.



Na figura, os círculos representam os *nós* da árvore, dos quais o mais acima é a *raiz* e os terminais, os mais abaixo, as *folhas*. De cada nó não terminal saem dois *ramos* ou duas *sub-árvores* (*ramo da esquerda* e *ramo da direita*), o que caracteriza as *árvores binárias*.

Um exemplo muitas vezes utilizado para ilustrar a *recursividade em árvore* é a geração da *sequência de Fibonacci*, na qual, cada número é igual à soma dos dois números anteriores, sendo os dois primeiros, por convenção, 0 e 1. Os primeiros elementos são 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...e a sequência pode ser gerada através de uma definição recursiva.

$$fib(n) = \begin{cases} n & \text{se } n < 2 \\ fib(n-1) + fib(n-2) & \text{nos outros casos} \end{cases}$$

Desta definição retira-se o procedimento respectivo, que apresenta duas chamadas no passo recursivo.

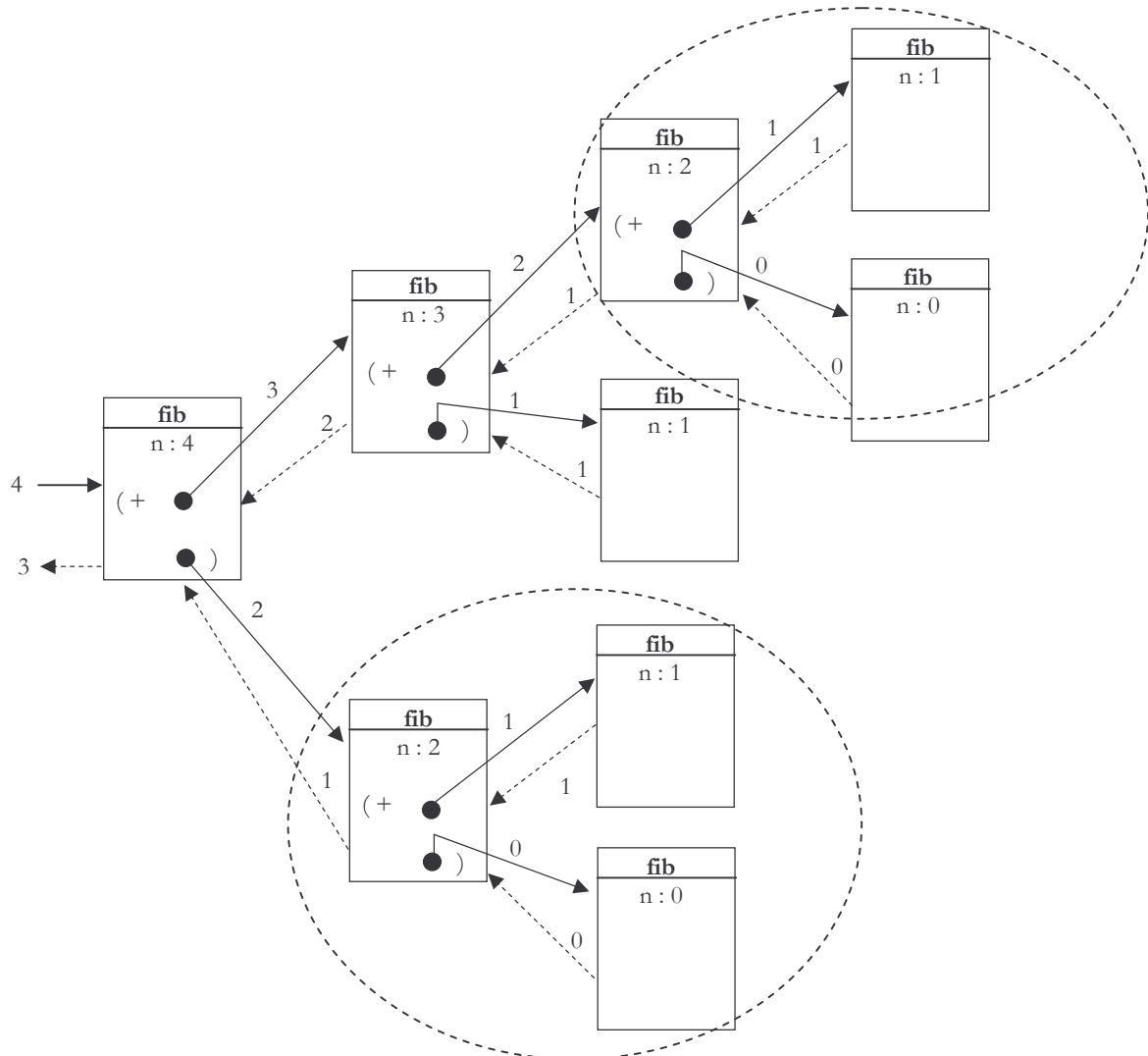
```
(define fib
  (lambda (n)
    (if (< n 2)
        n ; caso base
        (+ (fib (- n 1))
            (fib (- n 2))))))

> (fib 4)
3
```

Espaço para testar o procedimento *fib*.

O ecrã abre com o procedimento a testar e com o diálogo indicado.

Experimente com um argumento entre 30 e 40, vá depois aumentado esse argumento, de 1 em 1, meça o tempo de resposta e comente os resultados...



A chamada (*fib* 4), sob a forma gráfica, mostra que o processo gerado é *recursivo em árvore*.

Por uma questão de espaço, optou-se por desenhar a árvore rodada. Assim, a *árvore* tem a *raiz* em (*fib* 4), no lado esquerdo, e as *folhas* em (*fib* 1) e (*fib* 0), do lado direito. De cada *nó* (que não é *folha*) saem 2 *ramos*, sendo por isso uma *árvore binária*.

Em termos de tempo, o comportamento do processo recursivo em árvore é terrivelmente gastador. Basta verificar na figura que, por exemplo, o cálculo (*fib* 2) é duplicado. O número de chamadas ao procedimento *fib* é igual ao número de nós da árvore, o que corresponde a um comportamento exponencial. Sendo *n* o dado inicial, o número de nós é  $q^n$ , em que *q* não chega a ser 2, apesar de saírem 2 ramos de cada nó, uma vez que a árvore não é perfeitamente simétrica.

*Prova-se que  $q$ , designado por golden ratio, é igual a  $q = (1 + \sqrt{5}) / 2 \approx 1.6180$ , mas não é propriamente uma demonstração que nos interesse agora muito.  
No entanto, não fica impedido de a procurar.*

Em relação ao espaço de memória, verifica-se que o comportamento do processo é *apenas* linear, conclusão a que se chega seguindo a trajetória indicada pelas setas sobre a árvore. Os cálculos que se mostram nos diferentes ramos da árvore nunca estão todos suspensos ao mesmo tempo, pois vão sendo calculados à medida que se atinge algum dos casos base (*as folhas*). Por exemplo, o ramo inferior, (*fib* 2), só será percorrido depois de (*fib* 3) estar totalmente calculado.

*Como o Scheme não garante a ordem de cálculo dos operandos de uma expressão, poderá ser (*fib* 3) chamado só depois de (*fib* 2) calculado!  
Este facto terá alguma influência no que foi dito sobre o comportamento linear do processo?*

O número máximo de cálculos suspensos, que é de 4, situa-se na zona da árvore com maior profundidade (*ramo onde se encontra (fib 3)*), profundidade que revela uma certa proporcionalidade linear com o número a processar. Variando o número a processar, varia linearmente a profundidade da árvore.

O procedimento recursivo *fib* gera processos recursivos em árvore com um comportamento exponencial em tempo e linear em memória. Pode sempre tentar um procedimento recursivo que gere um processo iterativo, bastante mais eficiente, quer em tempo (*que passa a linear*) quer em espaço de memória (*que passa a constante*), tarefa que poderá revelar-se não muito simples.

A ideia a explorar, para a solução iterativa, é manter dois acumuladores, aos quais se associam, em cada iteração, dois números seguidos da sequência de Fibonacci, o número corrente (*ac-corrente*) e o seguinte (*ac-seguinte*). Serão inicializados com *ac-corrente* = *fib*(0) = 0, e *ac-seguinte* = *fib*(1) = 1.

Em cada iteração, desenrolar-se-ão, simultaneamente, as seguintes transformações:

- *ac-seguinte* + *ac-corrente* vai para *ac-seguinte*
- *ac-seguinte* vai para *ac-corrente*

Para completar a ideia, bastará inicializar um *contador* com *n*, o qual, em cada iteração, sofre uma redução de uma unidade. Quando *contador* atinge zero, a resposta encontra-se em *ac-corrente*.

```
(define fib-iter
  (lambda (contador ac-corrente ac-seguinte)
    (if (zero? contador)
        ac-corrente
        (fib-iter (sub1 contador)
                   ac-seguinte
                   (+ ac-seguinte ac-corrente)))))
```

Para esconder os parâmetros que não devem ser visíveis pelos utilizadores normais do procedimento, pois eles foram introduzidos apenas para facilitar a definição de uma ideia de resolução do problema, apresenta-se *fib-novo*.

```
(define fib-novo
  (lambda (n)
    (fib-iter n 0 1)))
```

Espaço para testar o procedimento *fib-novo*.

O ecrã abre com os procedimentos *fib-novo*, *fib-iter* e *fib*.

Experimente *fib-novo* e *fib* com os mesmos argumentos, primeiro com valores pequenos, entre 10 e 20, e depois valores maiores, entre 30 e 50, e comente os tempos de resposta.

#### Exemplo 4

Ao misturar cores, a partir de um conjunto de cores iniciais, produzem-se novas cores. Supõe-se, neste método de produção, que se utilizam sempre doses iguais de cada cor misturada. Assim, partindo das cores iniciais C1 e C2 conseguem-se as seguintes 3 cores distintas: C1, C2 e C1 + C2 (neste contexto, o sinal + significa mistura de cores em quantidades iguais). Com as cores iniciais C1, C2 e C3 já se conseguem 7 cores: C1, C2, C3, C1 + C2, C1 + C3, C2 + C3 e C1 + C2 + C3.

Estamos interessados em saber quantas cores distintas é possível produzir a partir de 4, 5, ..., ou  $n$  cores iniciais.

```
> (n-cores 0)
0
> (n-cores 1)
1
> (n-cores 2)
3
> (n-cores 3)
7
> (n-cores 4)
15
> (n-cores 5)
31
> (n-cores 10)
??
```

Vamos procurar uma definição recursiva para determinar o número de cores distintas que se conseguem obter com  $n$  cores iniciais. Em seguida, escrever em *Scheme* o procedimento *n-cores* com base na definição apresentada, que responda como se indica.

A definição recursiva para determinar o número de cores é apresentada, bem como a respectiva justificação.

$$nCores(n) = \begin{cases} n & \text{se } n \leq 1 \\ 1 + 2 * nCores(n-1) & \text{nos outros casos} \end{cases}$$

Com  $n = 0$  cores, não se produz qualquer cor. Com  $n = 1$  cor, só se produz uma cor. Estas duas situações constituem o *Caso base*.

Nos outros casos, considere uma cor qualquer do conjunto inicial de cores.

- essa cor isolada produz 1 cor
- sem essa cor, apenas com as restantes cores, produz-se  $nCores(n-1)$  distintas
- agora, se a todas essas  $nCores(n-1)$  se juntar a cor que se isolou, obtém-se mais  $nCores(n-1)$  distintas.

Ao traduzir esta definição recursiva em *Scheme*, resulta o procedimento *n-cores*.

```
(define n-cores
  (lambda (n)
    (if (<= n 1)
        n ; caso base
        (+ 1 ; caso geral
           (* 2
              (n-cores (sub1 n)))))))
```

Espaço para testar o procedimento *n-cores*.  
O ecrã abre com o procedimento a testar.

*Como desafio, tente identificar o tipo de comportamento da solução apresentada, quer em tempo quer em espaço.*

Por uma questão de curiosidade e também para confirmar os resultados obtidos, vamos aproveitar para relembrar algumas definições de Análise Combinatória.

- *Permutações de  $n$  elementos*,  $P_n = n!$ , são os agrupamentos formados por esses  $n$  elementos, diferindo uns dos outros apenas pela ordem dos elementos. Permutações de A, B e C,  $3! = 6$ , são: ABC, ACB, BAC, BCA, CAB e CBA.
- *Arranjos de  $n$  elementos  $k$  a  $k$* ,  $A_k^n = \frac{n!}{(n-k)!}$ , são os agrupamentos de  $k$  elementos que diferem entre si quer pelos elementos que os constituem quer pela ordem. Arranjos de A, B e C 2 a 2,  $3!/1! = 6$ , são: AB, BC, AC, CA, BC e CB.
- *Combinações de  $n$  elementos  $k$  a  $k$* ,  $C_k^n = \frac{n!}{(n-k)!k!}$ , são os agrupamentos de  $k$  elementos que diferem entre si pelos elementos que os constituem, independentemente da ordem. Combinações de A, B e C 2 a 2,  $3!/(1!2!) = 3$ , são AB, AC e BC.

Veja uma tentativa de explicação dos resultados do procedimento *n-cores*, baseada em Análise Combinatória.

```
> (n-cores 2)
3
> (n-cores 3)
7
```

número de cores equivalente a  $C_2^2 + C_1^2 = 1 + 2 = 3$

número de cores equivalente a  $C_3^3 + C_2^3 + C_1^3 = 1 + 3 + 3 = 7$

*Tente uma alternativa a *n-cores*, baseando-se na definição de combinações de  $n$  elementos  $k$  a  $k$ .*

*Como desafio, tente identificar o tipo de comportamento da solução a que chegar, quer em tempo quer em espaço.*

*Conseguiu uma solução melhor que a apresentada no Exemplo?*

### Exercício 3

Relativamente à situação do exemplo anterior, suponha agora que, por limitações tecnológicas, terá que haver um limite máximo ao número de cores misturadas na obtenção de novas cores.

- Apresente uma definição recursiva para determinar o número de cores distintas que se conseguem obter com  $n$  cores iniciais, sabendo que não se podem misturar mais do que  $k$  cores, em que  $k$  menor ou igual que  $n$ .

- Escreva em *Scheme* o procedimento *n-cores-lim* com base na definição apresentada, que responde como se mostra mais à frente.
- Apresente a representação gráfica da chamada *(n-cores-lim 3 2)* e verificar se a recursividade é *linear* ou *em árvore*.
- Indique se os processos gerados são recursivos ou iterativos e o tipo de comportamento em relação ao tempo e espaço.

Observe alguns exemplos de aplicação do procedimento *n-cores-lim* em que o primeiro parâmetro representa o número de cores iniciais e o segundo é o limite máximo de cores que se podem misturar.

```
> (n-cores-lim 3 0)
0
> (n-cores-lim 3 1)
3
> (n-cores-lim 3 2)
6
> (n-cores-lim 3 3)
7
> (n-cores-lim 3 4)
7
```

Espaço para desenvolver e testar o procedimento pedido.  
O ecrã abre procedimento *n-cores*.

#### Exercício 4

Observe com atenção a seguinte pirâmide de números inteiros.

										1	.....	linha 1
									1	1	.....	linha 2
								1	2	1	.....	linha 3
							1	3	3	1	.....	linha 4
						1	4	6	4	1	.....	linha 5
			1	5	10	10	5	1	.....	.....	.....	linha 6
		1	6	15	20	15	6	1	.....	.....	.....	linha 7
	1	7	21	35	35	21	7	1	.....	.....	.....	linha 8

O primeiro e último número de cada linha é sempre 1.

Os números intermédios são iguais à soma dos dois números mais próximos da linha anterior. Por exemplo, o número na 4ª posição da linha 8, o 35, é igual à soma dos números na 3ª e 4ª posições da linha 7, ou seja,  $35 = 15 + 20$ .

A função *ponto-piramide* tem dois parâmetros, *lin* e *col*, e devolve o número que se encontra na linha *lin* e na coluna *col* da pirâmide. Por exemplo, *ponto-piramide (8, 4)* devolve 35.

- Comece por apresentar uma definição recursiva (em notação matemática) para a função *ponto-piramide (lin, col)*, em que *lin* e *col* são garantidamente inteiros positivos. A função devolve 0 quando *col* é maior que *lin*.
- Escreva em *Scheme* o procedimento *ponto-piramide* correspondente à função apresentada na questão anterior, sabendo que os argumentos serão sempre inteiros positivos.
- Escreva em *Scheme* o procedimento *linha-piramide* com o parâmetro *lin* que visualiza todos os pontos da linha *lin* da pirâmide. Este procedimento pode utilizar o procedimento *ponto-piramide* como auxiliar.

```
> (linha-piramide 4)
linha 4: 1, 3, 3, 1
> (linha-piramide 8)
linha 8: 1, 7, 21, 35, 35, 21, 7, 1
```

Espaço para desenvolver e testar os procedimentos pedidos.  
O ecrã abre vazio.

*Identifique o tipo de comportamento das soluções a que chegar, quer em tempo quer em espaço.*

### **Avaliação dos recursos computacionais consumidos pelos processos**

Neste momento sabe muito bem que os recursos computacionais consumidos pelos processos são o *tempo de cálculo* e o *espaço de memória*. Para um mesmo problema, processos resultantes de procedimentos diferentes podem requerer recursos muito diferentes, como já teve ocasião de verificar ao comparar soluções recursivas e iterativas. Uma medida não muito rigorosa dos recursos consumidos, mas suficiente para permitir comparar e escolher, entre várias soluções a mais adequada para um problema, baseia-se na designada *Ordem de Crescimento* ou *Ordem de Complexidade* (*Order of growth* ou *Order of complexity*).

Através da Ordem de Crescimento é possível caracterizar o consumo dos processos em função da *dimensão do problema*.

*Pare um momento e tente imaginar o que será a dimensão de um problema.  
Por exemplo, no factorial, o que será a dimensão do problema?  
E na série de Fibonacci?*

A dimensão do problema poderá ser o valor numérico a processar, como acontecia no cálculo do factorial, na determinação da sequência de *Fibonacci* e nos exemplos das misturas de cores. Em todos estes casos, os recursos computacionais gastos pelos processos gerados estavam relacionados com o valor numérico fornecido como argumento. Normalmente, para valores maiores correspondiam recursos gastos também maiores. O objectivo é saber, se a dimensão do problema duplicar como é que este facto se manifesta nos recursos computacionais consumidos? Também duplicam, variando linearmente? Variam exponencial ou logaritmicamente? Permanecem constantes, ou seja, são independentes do valor do argumento?

Já percebeu que não interessa muito ter uma medida rigorosa, mas apenas uma ideia de como reagirá o processo gerado por um procedimento, qual o seu comportamento especialmente quando se faz crescer (para valores muito elevados) a dimensão do problema.

*Antes de avançar pense um pouco na situação que se descreve ou noutra que invente...  
Imagine que vai fazer uma viagem no seu carro. Entra na garagem, abre a porta do carro, instala-se, põe o carro a trabalhar e depois vai fazer o percurso desejado.  
Vai fazer uns metros, se resolver tomar o pequeno-almoço no café da esquina ou uns longos quilómetros se planeia ir a Coimbra ou a Lisboa...  
O que representará, neste caso, a dimensão do problema?  
Quais serão, neste caso, os recursos que serão sempre consumidos, qualquer que seja a dimensão do problema e que são desprezados para efeitos da análise da Ordem de Crescimento?  
Em relação a estes últimos recursos, em que situações podem ser considerados desprezáveis?*

Considere

- $n$  o valor que, de alguma forma, reflecte a dimensão do problema
- $R(n)$  os recursos consumidos por um processo, para um  $n$  suficientemente elevado, que “dilua” os recursos que são sempre gastos, qualquer que seja a dimensão do problema
- $O(f(n))$  a Ordem de Crescimento de função  $f(n)$ , em que  $f(n)$  pode ser constante, linear, ou outra.

Neste contexto,

se  $R(n) \leq K * f(n)$ , para um  $K$  constante e  $n$  suficientemente elevado,  
então os recursos consumidos apresentam uma Ordem de Crescimento  $f(n)$ , que se exprime  
através de  $O(f(n))$ .

*Pode parecer estranho, mas uma situação em que  $R(n) \leq 10 * f(n)$   
e outra em  $R(n) \leq 1000 * f(n)$   
terão ambas uma ordem de crescimento igual a  $O(f(n))$ !  
Parece-lhe isto aceitável?*

*Mas não terão ambas o mesmo tipo de comportamento quando a dimensão  $n$  varia?*

Analise algumas situações típicas...

#### Ordem de complexidade constante

- Se  $R(n)$  tiver um comportamento constante, independente de  $n$ , a Ordem de Crescimento diz-se constante e representa-se por  $O(1)$ .  
Veja a razão desta afirmação. Por exemplo, se  $R(n) = 1000$ , podemos encontrar uma constante  $K$  (neste caso,  $K=1000$ ), tal que  $R(n) \leq K * 1$ . Comparando com  $R(n) \leq K * f(n)$ , resulta  $f(n) = 1$ , ou seja,  $O(1)$ .

Aqui poderão começar a surgir algumas dúvidas, pois um processo que gaste constantemente recursos equivalentes a 1 unidade (de tempo ou de memória) e outro 1000 unidades do mesmo recurso seriam ambos classificados com Ordem de Crescimento  $O(1)$ . De facto, se existe um  $K_1$  tal que  $1000 \leq K_1 * 1$ , por maioria de razão se encontraria um  $K_2$  em  $1 \leq K_2 * 1$ . Ou seja, em ambos os casos  $f(n)=1$ , logo  $O(1)$ .

*À primeira vista, esta conclusão não parece nada aceitável, pois em termos absolutos, de facto, um dos processos gasta 1000 vezes mais recursos do que o outro.*

*Mas não acha que os dois processos, com  $O(1)$ , reagem da mesma maneira, ou seja, consomem recursos de uma forma constante, independente da dimensão do problema?*

*Dos casos encontrados, apresente pelo menos um com comportamento  $O(1)$  e indique se esse comportamento é relativo ao tempo ou ao espaço ou a ambos.*

*Imagine uma turma com 50 alunos e quer saber se o primeiro aluno, aquele que está mais perto de si, pesa 60 quilos.*

*Entretanto, a turma aumentou e agora, em vez de 50, tem 100 alunos e continuava a querer saber a mesma coisa sobre o primeiro aluno.*

*Que tipo de comportamento tem este processo? Justifique.*



### Ordem de complexidade linear

- Se  $R(n)$  tiver um crescimento linear com  $n$ , a Ordem de Crescimento diz-se linear e representa-se por  $O(n)$ . Por exemplo, se  $R(n) = 1000 * n$ , então  $R(n) \leq K * n$ , e  $O(n)$ .

Aplicando o que se acaba de apresentar às soluções recursiva e iterativa do cálculo do factorial, podemos resumir.

- *factorial*, com processos recursivos: Tempo:  $O(n)$ ; Espaço:  $O(n)$
- *factorial-novo*, com processos iterativos: Tempo:  $O(n)$ ; Espaço:  $O(1)$ .

*É de prever que factorial esgotará a memória disponível acima de um valor de  $n$ , situação que nunca ocorrerá com factorial-novo. Em termos de tempo, ambos apresentam um comportamento linear.*  
*Concorda com o que acaba de se afirmar? Justifique.*

*Imagine uma turma de alunos e quer saber se algum dos alunos pesa 60 quilos! Na melhor hipótese, o primeiro aluno pesa 60 quilos e o problema poder-se-ia considerar resolvido.*  
*Mas neste tipo de análise, o mais seguro é contar com a pior hipótese.*  
*E qual é a pior hipótese?*  
*O tipo de comportamento deste processo é  $O(n)$ . Justifique.*

### Ordem de complexidade exponencial

- Se  $R(n)$  tiver um crescimento  $M^n$  com  $n$ , a Ordem de Crescimento diz-se *exponencial* e representa-se por  $O(M^n)$ . Por exemplo, se  $R(n) = 1000 * M^n$ , então  $R(n) \leq K * M^n$ , e  $O(M^n)$ .

Se, por exemplo,  $R(n) = 2^n$ , então  $O(2^n)$ , caso em que os recursos duplicam quando  $n$  aumenta de um valor unitário. Para  $R(4) = 2^4 = 16$ , para  $R(5) = 2^5 = 32$ , para  $R(6) = 2^6 = 64$ , ...

*Verifique o crescimento vertiginoso neste tipo de comportamento, certamente impraticável para valores de  $n$  elevados, por conduzir ao esgotamento dos recursos computacionais...*

*Imagine que para  $n=1$ , um processo  $O(2^n)$  gasta 100 bytes de memória e consome 1 segundo de processador.*

*Calcule em MB e em dias o consumo do mesmo processo para  $n=100$  e  $n=1000$ ...*

Relembrando os procedimentos *fib* e *fib-iter* da série de Fibonacci:

- *fib*: Tempo:  $O(q^n)$ , com  $q \approx 1.6180$  (*golden ratio*); Espaço:  $O(n)$
- *fib-iter*: Tempo:  $O(n)$ ; Espaço:  $O(1)$ .

Uma solução com um comportamento exponencial será de evitar, pois tende a gastar recursos de uma forma exagerada, quando a dimensão aumenta.

### Ordem de complexidade logarítmica

- Se  $R(n)$  tiver um crescimento  $\log_M n$  com  $n$ , a Ordem de Crescimento diz-se logarítmica e representa-se por  $O(\log_M n)$ . Por exemplo, se  $R(n) = 1000 * \log_M n$ , então  $R(n) \leq K * \log_M n$ , e  $O(\log_M n)$ .

Sendo  $R(n) = \log_2 n$ , então  $O(\log_2 n)$ , caso em que os recursos aumentam de um valor unitário quando  $n$  duplica.

Para  $R(8) = \log_2 8 = 3$ ,  $R(16) = \log_2 16 = 4$ ,  $R(32) = \log_2 32 = 5$ , ...

*Confirme, mesmo sem ter visto ainda qualquer exemplo, que  $O(\log_{MN})$  é melhor que a ordem de crescimento linear, muito melhor que a exponencial e apenas inferior à constante.*

*Pista: Esboce um gráfico com funções dos tipos referidos (exponencial, linear, logarítmica e constante), por exemplo para valores de  $n$  iguais a 0, 8, 16 e 32.*

Até agora, não encontramos qualquer situação que possa ilustrar a ordem de complexidade é logarítmica,  $O(\log_k n)$ . Com o objectivo de encontrar uma dessas situações, vamos por momentos esquecer que o *Scheme* disponibiliza o procedimento *expt*, cuja chamada (*expt b n*) devolve  $b^n$ .

Pretende-se desenvolver o procedimento *potencia* que espera 2 argumentos,  $b$  e  $n$ , sendo  $n$  um inteiro positivo, e a chamada (*potencia b n*) devolve  $b^n$ .

De imediato, surge uma definição recursiva e o respectivo procedimento.

- Caso base:  $b^0 = 1$
- Caso geral:  $b^n = b \cdot b^{n-1}$

```
(define potencia
  (lambda (b n)
    (if (zero? n)
        1
        (* b (potencia b (sub1 n))))))
```

O processo gerado por *potencia* não é iterativo mas sim recursivo.

Trata-se de um caso em que não existe *recursividade em cauda*, pois a última operação do procedimento é uma multiplicação, que fica suspensa até ao caso base, definido por  $n=0$ . Este processo, em termos de tempo e espaço, apresenta ordens de crescimento linear,  $O(n)$ .

Uma solução iterativa, *potencia-nova*, é obtida com um procedimento auxiliar, com um terceiro parâmetro, para acumular o resultado das sucessivas multiplicações.

```
(define potencia-nova
  (lambda (b n)
    (potencia-aux b n 1)))

(define potencia-aux
  (lambda (base contador acumulador)
    (if (zero? contador)
        acumulador
        (potencia-aux base (sub1 contador) (* base acumulador)))))
```

Nesta solução, é evidente a *recursividade em cauda*. O processo gerado é iterativo, com  $O(1)$  em termos de espaço e  $O(n)$  em termos de tempo.

*Por que razão é dito que este processo é  $O(1)$  em espaço e  $O(n)$  em tempo?*

Neste problema, é ainda possível explorar uma outra ideia, em que a redução do expoente acontece muito mais rapidamente.

- Caso base:  $b^0 = 1$
- Casos gerais

- Se  $n$  é par:  $b^n = (b^{n/2})^2$
- Se  $n$  é ímpar:  $b^n = b \cdot b^{n-1}$

O processo gerado por *potencia-melhorada* é recursivo e, em termos de tempo e espaço, é  $O(\log_2 n)$ . Basta reflectir um pouco para se chegar a esta conclusão. Sempre que o expoente é dividido por 2, passando de  $2n$  para  $n$ , o que corresponde a dividir por 2 a dimensão do problema, apenas se utiliza mais uma multiplicação.

```
(define potencia-melhorada
  (lambda (b n)
    (cond ((zero? n) 1)
          ((even? n) (quadrado (potencia-melhorada b (/ n 2))))
          (else
           (* b (potencia-melhorada b (sub1 n)))))))

(define quadrado
  (lambda (x)
    (* x x)))
```

### Exercício 5

Procurar uma solução iterativa para *potencia-melhorada*, a designar por *potencia-melhorada-nova*, identificando e justificando a Ordem de Crescimento da solução encontrada.

Espaço para desenvolver e testar o procedimento pedido.  
O ecrã abre com o procedimento *potencia-melhorada*.

Já percebeu que a ordem de crescimento só nos permite caracterizar, de forma aproximada e não absoluta, o consumo de recursos computacionais em função da dimensão do problema. Dá-nos, no entanto, uma ideia suficientemente clara do comportamento dos processos gerados.

Para finalizar este tema, apresenta-se mais um exemplo em que se torna evidente o carácter aproximado deste tipo de medida. Para uma situação hipotética em que os recursos consumidos fossem especificados por  $R(n) = 15 \cdot n^7 + 30 \cdot n^6$ , a ordem de crescimento respectiva seria  $O(n^7)$ .

Sendo  $R(n) = 15 \cdot n^7 + 30 \cdot n^6$ , então será aproximadamente  $R(n) = 15 \cdot n^7$ .

E agora  $R(n) = 15 \cdot n^7 \leq K \cdot n^7$  de onde resulta  $O(n^7)$ .

Como curiosidade, o erro que se comete ao fazer a aproximação  $R(n) = 15 \cdot n^7$  é de cerca de 4% para  $n = 50$ , caindo para 2% para  $n = 100$ .

*Tente confirmar estes erros?*

*Se não foi capaz de confirmar os erros, tem agora uma pista.*  
*Considerando apenas o termo de maior grau,  $R(n) = 15 \cdot n^7$ , o erro cometido seria*  
 *$(15 \cdot n^7 + 30 \cdot n^6 - 15 \cdot n^7) / 15 \cdot n^7$  ou seja,  $30n^6 / 15n^7 = 2/n$ .*  
*Assim, para  $n = 50$ , o erro seria de  $2/50 = 4\%$  e para  $n = 100$ ,  $2/100 = 2\%$ .*

Uma maneira prática de chegar à Ordem de crescimento de uma situação é aplicar à expressão dos recursos consumidos as seguintes simplificações:

- ignorar os termos constantes
- ignorar os termos que se tornam menos importantes quando  $n$  cresce

Se os recursos consumidos forem  $R(n) = n - 1$ , a constante -1 é ignorada e resulta  $O(n)$ .

Mas se forem  $R(n) = 5 \cdot n^7 + 500 \cdot n^6 - 56$ , as constantes 5, 500 e -56 são ignoradas, o termo  $n^6$  torna-se menos importante que  $n^7$  quando  $n$  cresce e resulta  $O(n^7)$ .

*Imagine uma turma de alunos e quer saber se algum aluno pesa o mesmo que outro.*

*Na pior hipótese...*

*Testa-se o primeiro com todos os outros.*

*Depois, exclui-se o primeiro e testa-se o segundo com os restantes...*

*O tipo de comportamento deste processo é  $O(n^2)$ .*

*Trata-se de um comportamento quadrático. Justifique.*

*Apresente um gráfico com funções de todos os tipos referidos (exponencial, linear, logarítmica e constante), incluindo agora também a função quadrática, por exemplo para valores de  $n$  iguais a 0, 8, 16 e 32.*

*Inclua também um valor muito mais elevado de  $n$  (por ex., 1024), mesmo que surjam dificuldades ao desenhar o valor de algumas das funções.*

*Aproveite para verificar no gráfico o que ocorre para os valores de  $n$  mais elevados, aqueles que normalmente nos preocupam em termos da análise da Ordem de Crescimento...*