Módulo 2.0 - **Resumo dos módulos 2.1, 2.2 e 2.3, exemplos e exercícios**

Resumo dos assuntos abordados nos módulos
- Módulo 2.1 - **Resolver problemas, passo a passo, na direcção de um caso conhecido!**
- Módulo 2.2 - **Processos e os recursos computacionais que consomem**
- Módulo 2.3 - **Projecto - Jogo *O computador-adivinho***

Após o resumo são apresentados alguns exemplos e exercícios para treino da matéria relacionada com os módulos acima referidos.

A utilização do Scheme é fundamental neste módulo.

**Resumo dos assuntos abordados nos módulos 2.1, 2.2 e 2.3**

Neste conjunto de módulos é introduzido o conceito de recursividade e posta em evidência a importância que este conceito tem na resolução de problemas de programação. A abordagem *de-cima-para-baixo* (*top-down*) é apresentada como uma forma sistemática de atacar os problemas, dividindo-os em sub-problemas mais simples.

**Módulo 2.1**
A *recursividade* é o tema fulcral deste módulo, amplamente justificado pela sua enorme importância no âmbito da programação. Trata-se de um conceito cuja utilização nos acompanha quase sem darmos por ele em variadíssimas situações do nosso quotidiano. Os *procedimentos recursivos* usam a recursividade e identificam-se por se chamarem a si próprios.

**Módulo 2.2**
É muito importante saber distinguir entre *procedimento* e *processo*. Procedimento é uma peça de texto que se pode ler, enquanto que o processo existe no interior do computador, criado pela chamada do procedimento, e consome recursos computacionais para gerar resultados. Os *procedimentos recursivos* podem criar *processos recursivos* e *processos iterativos*. Os primeiros caracterizam-se por deixar operações suspensas em memória, até que se verifique uma certa condição, a *condição de terminação*. Os processos iterativos não deixam operações suspensas em memória. Assim, é de prever que os processos recursivos sejam mais consumidores de recursos computacionais, especialmente memória, do que os iterativos. Com a *Ordem de Crescimento* ou *Ordem de Complexidade* procura-se determinar o comportamento dos processos e os recursos computacionais que gastam face à dimensão dos problemas, sendo apresentados exemplos de ordem de crescimento *O(1) - constante*, *O(n) - linear*, *O($M^n$) - exponencial*, *O($\log_M n$) - logarítmica* e *O($n^2$) - quadrática*.

**Módulo 2.3**
O Scheme disponibiliza mecanismos para a escrita de procedimentos como *blocos* ou *caixas-pretas*, permitindo definir no interior de procedimentos outros procedimentos, através das formas especiais *let*, *letrec* e *let**. Esta característica dos procedimentos é especialmente importante na resolução de problemas de média ou elevada complexidade.
A segunda parte do módulo corresponde ao desenvolvimento de um pequeno projecto que aqui surge para introduzir a abordagem *de-cima-para-baixo* (*top-down*) na programação de problemas com alguma complexidade. Esta abordagem passa por várias fases, *especificação do problema*, *procura de uma ideia de solução*, *definição do algoritmo*, *desdobramento do problema em sub-problemas* e, finalmente, *codificação* e *teste* da solução encontrada para os vários sub-problemas e para o problema na sua globalidade.

## Exercícios e exemplos

São apresentados alguns exercícios e exemplos para consolidação da matéria abordada nos Módulos 2.1, 2.2 e 2.3.

### Exemplo 1

O procedimento *impar-cima-par-baixo* espera um argumento inteiro positivo. Se esse inteiro for par, dividi-o por 2. Se for ímpar, multiplica-o por 3 e depois soma 1. O resultado assim obtido é sujeito a um tratamento idêntico ao indicado e só pára quando o resultado for 1.

```
> (impar-cima-par-baixo 6)
6  3  10  5  16  8  4  2  1  OK
```

*Desenvolva manualmente mais algumas situações do problema.*
*Identifique a ideia de solução indicada no código que se segue.*
*Apresente o algoritmo respectivo.*

```
(define impar-cima-par-baixo
  (lambda (num)
    (display num)
    (display "  ")
    (cond ((= num 1)
           (display "OK")
           (newline))
          ((even? num)
           (impar-cima-par-baixo (quotient num 2)))
          (else
            (impar-cima-par-baixo (add1 (* num 3)))))))
```

Espaço para testar o procedimento *impar-cima-par-baixo*.
O ecrã abre com este procedimento.

### Exemplo 2

O procedimento *imprime-bin* espera um argumento inteiro positivo, na base 10. Este procedimento imprime o inteiro dado, convertido para a base 2.

```
> (imprime-bin 76)
1001100
> (imprime-bin 6)
110
```

*Desenvolva manualmente mais algumas situações do problema.*

A visualização do valor na base 2 começa pelo dígito mais significativo (no sentido da esquerda para a direita), mas sabemos que é muito mais fácil começar pelo dígito menos significativo. Por exemplo, para determinar o dígito menos significativo de um número na base 2 bastará determinar o resto da divisão inteira desse número por 2. E para retirar o dígito menos significativo ao número, bastará calcular o quociente daquela divisão. Uma solução recursiva, que deixe suspensa a operação de visualização, resolve facilmente o facto de pretendermos começar a visualização pelo dígito mais significativo quando o menos significativo está "muito mais à mão".

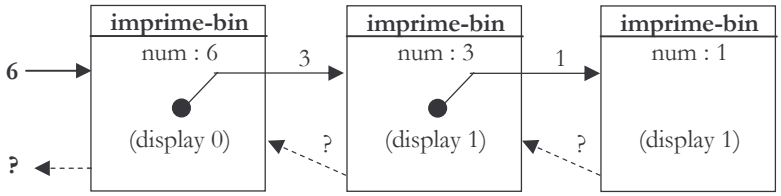
*Identifique a ideia de solução indicada no código que se segue.*
*Apresente o algoritmo respectivo.*

```
(define imprime-bin
  (lambda (num)
    (cond ((< num 2) (display num))
          (else
            (imprime-bin (quotient num 2))
            (display (remainder num 2))))))
```

Os processos gerados não são iterativos, mas sim recursivos, com *O(n)* em tempo e espaço, conclusão que também se poderá retirar da representação gráfica relativa à chama `(imprime-bin 6)`.

*É dito que os processos gerados são recursivos. Justifique.*

Verifique, pela representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã.




Espaço para testar o procedimento *imprime-bin*.
O ecrã abre com este procedimento.
Experimente na *cláusula else* trocar a posição da chama *recursiva imprime-bin* com *display* e explique o que acontece.
Depois desta troca, os processos gerados são recursivos ou iterativos. Justifique.

Variante
O procedimento *imprime-na-base* tem 2 parâmetros, *num* e *base*, para os quais espera dois argumentos inteiros positivos, ambos na base 10. Este procedimento imprime *num* na base especificada por *base*.

```
> (imprime-na-base 76 2)
1001100
> (imprime-na-base 76 3)
2211
> (imprime-na-base 76 10)
76
```

*Desenvolva manualmente mais algumas situações do problema.*
*Identifique a ideia de solução indicada no código que se segue.*
*Apresente o algoritmo respectivo.*

```
(define imprime-na-base
  (lambda (num base)
    (cond ((< num base) (display num))
          (else
            (imprime-na-base (quotient num base) base)
            (display (remainder num base))))))
```

Espaço para testar o procedimento *imprime-na-base*.
O ecrã abre com este procedimento.

**Exemplo 3**

O procedimento *caminho-errante* espera, como argumento, um valor inteiro que é considerado o ponto de partida e gera, aleatoriamente, 1, 2 ou 3. Quando o número gerado é 1 subtrai 1 ao ponto de partida, quando é 2 soma-lhe 1, e quando é 3 nada altera. O ponto de partida é assim actualizado e o procedimento chama-se a si próprio, pois não tem fim.

```
> (caminho-errante 20)
20  19  19  20  21  22  22  21  20  21  22  22  23  24  23  22  23
23  24  25  26  25  26  26  26  26  26  25  25  25  26

user break
```

*Tente desenvolver manualmente mais algumas situações do problema.*
*Que tipo de problema encontra?*
*Identifique a ideia de solução indicada no código que se segue.*
*Apresente o algoritmo respectivo.*

Notar que neste procedimento recursivo não existe caso base, pois não tem fim.

```
(define caminho-errante
  (lambda (origem)
    (display origem)
    (display "  ")
    (let ((numero-aleat (aleatorio-de-1-ate-limite 3)))
      (cond ((= numero-aleat 1)
             (caminho-errante (sub1 origem)))
            ((= numero-aleat 2)
             (caminho-errante (add1 origem)))
            (else
              (caminho-errante origem))))))

(define aleatorio-de-1-ate-limite     ; gera, aleatoriamente, um valor
  (lambda (limite)                    ; entre 1 e limite
    (add1 (random limite))))
```

Espaço para testar o procedimento *caminho-errante*.
O ecrã abre com este procedimento e com o procedimento auxiliar *aleatorio-de-1-ate-limite*.

**Exercício 1**

Em relação ao exemplo anterior (procedimento *caminho-errante*), apresente uma solução que prevê enganos do utilizador, quando este indica um argumento que não obedece às condições enunciadas. A solução deverá ser um procedimento com a designação *caminho-errante-melhorado*, apresentado como uma *caixa-preta*, portanto com todos os seus procedimentos auxiliares definidos localmente e não definidos no Ambiente Global do Scheme.

Espaço para desenvolver e testar o procedimento *caminho-errante-melhorado*.
O ecrã abre com o procedimento *caminho-errante* e o respectivo procedimento auxiliar.

**Exercício 2**

Escreva em Scheme o procedimento *caminho-errante-2d* que espera, como argumentos, dois valores inteiros, considerados as coordenadas *x* e *y* de um ponto 2D, tomado como local de partida. O procedimento gera, aleatoriamente, 1, 2 ou 3. Quando o número gerado é 1 subtrai 1 à coordenada *x*, quando é 2 soma 1 a essa mesma coordenada, e quando é 3 nada altera. Depois gera mais um

inteiro aleatório entre 1 e 3, e faz a mesma coisa para a coordenada *y*. Este procedimento não tem fim.

```
> (caminho-errante-2d 20 20)
20:20  21:21  22:20  21:21  21:21  21:20  21:20  22:20  22:19
22:18  21:18  21:19  20:19  21:20  22:20  22:19  21:19

user break
```

| Espaço para desenvolver e testar o procedimento *caminho-errante-2d*.<br>O ecrã abre com o procedimento *caminho-errante* e o respectivo procedimento auxiliar. |
|---|

**Exercício 3**

Observe o comportamento do procedimento *imprime-na-base* apresentado num dos exemplos anteriores, quando o argumento correspondente à base é superior a 10.

```
> (imprime-na-base 76 11)
610                          <---- deveria ser 6a
> (imprime-na-base 76 16)
412                          <---- deveria ser 4c
```

| Espaço para testar o procedimento *imprime-na-base*.<br>O ecrã abre com este procedimento. |
|---|

| *Explique os resultados obtidos com os argumentos 76 e 11 e com os argumentos 76 e 16.* |
|---|

| O espaço aberto é também utilizado para desenvolver e testar procedimento *imprime-ate-base-20* definido de seguida. |
|---|

Desenvolva um procedimento designado por *imprime-ate-base-20*, que responda correctamente até à base 20, ou seja, para além dos dígitos decimais (0 a 9) ainda utiliza as letras de *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, e *i*.

```
> (imprime-ate-base-20 76 11)
6a
> (imprime-ate-base-20 76 16)
4c
> (imprime-ate-base-20 76 20)
3g
```

**Exercício 4**

Escreva em Scheme um procedimento para calcular a função de *Ackermann*, sabendo que esta é definida para inteiros não negativos, recorrendo à recursividade.

$$A(m,n)=\begin{cases} n+1 & se \quad m=0 \\ A(m-1,1) & se \quad m>0 \quad e \quad n=0 \\ A(m-1,A(m,\ n-1)) & se \quad m>0 \quad e \quad n>0 \end{cases}$$

| Espaço para desenvolver e testar o procedimento pedido.<br>O ecrã abre vazio.<br>Experimente aumentar os argumentos, primeiro segundo *m* mantendo n constante e depois fazendo o contrário. Tente interpretar o comportamento do procedimento especialmente em termos do tempo de resposta |
|---|

*Identifique e justifique a Ordem de Crescimento da solução encontrada, em relação ao espaço e ao tempo.*

**Exercício 5**
Escreva em Scheme um procedimento para calcular o *n-ésimo* número na sequência de *Mewman-Conway* definida por:

$$P(n)=\begin{cases} 1 & se \quad n=1 \quad ou \quad n=2 \\ P(P(n-1))+P(n-P(n-1)) & se \quad n>2 \end{cases}$$

Espaço para desenvolver e testar o procedimento pedido.
O ecrã abre vazio.
Experimente aumentar o argumento e tente interpretar o comportamento do procedimento especialmente em termos do tempo de resposta.

*Identifique e justifique a Ordem de Crescimento da solução encontrada, em relação ao espaço e ao tempo.*

**Exercício 6** - este não é nada fácil...
Pretende-se determinar o número de hipóteses de trocar *n* cêntimos do euro, assumindo apenas a existência das moedas de 1, 2, 5, 10, 20 e 50 cêntimos, 1 e 2 euros.
Por exemplo,

1 cêntimo - 1 hipótese: 1 moeda de 1
2 cêntimos - 2 hipóteses: 1 moeda de 2, 2 moedas de 1
3 cêntimos - 2 hipóteses: 3 moedas de 1, 1 moeda de 1 + 1 moeda de 2
4 cêntimos - 3 hipóteses: 4 moedas de 1, 2 moedas de 1 + 1 moeda de 2, 2 moedas de 2

*Determine manualmente o número de hipóteses de troca para outras situações, nomeadamente, 25, 75 e 500 cêntimos.*

*Apresente uma definição recursiva para determinar o número de hipóteses de trocar n cêntimos do euro, assumindo apenas a existência das moedas de 1, 2, 5, 10, 20 e 50 cêntimos, 1 e 2 euros.*

Tendo por base a definição recursiva desenvolvida, escreva em Scheme o procedimento *conta-trocas* com um único parâmetro, *quantia*, que representa uma quantia em cêntimos e devolve o número de trocas possíveis dessa quantia com as moedas indicadas.

```
> (conta-trocas 1)
1
> (conta-trocas 2)
2
> (conta-trocas 3)
2
> (conta-trocas 4)
3
> (conta-trocas 0)
0
>
```

Segue-se uma hipótese de procedimento auxiliar importante para este problema:

```
(define outra-moeda
  (lambda (pos)
    (cond ((= pos 1) 200)  ; moeda de 2 euros em centimos
          ((= pos 2) 100)  ; moeda de 1 euro em centimos
          ((= pos 3) 50)     ; moeda de 50 centimos
          ((= pos 4) 20)     ; moeda de 20 centimos
          ((= pos 5) 10)     ; moeda de 10 centimos
          ((= pos 6) 5)      ; moeda de 5 centimos
          ((= pos 7) 2)      ; moeda de 2 centimos
          ((= pos 8) 1))))   ; moeda de 1 centimo
```

Espaço para desenvolver e testar o procedimento pedido.
O ecrã abre vazio.
Experimente aumentar o argumento e tente interpretar o comportamento do procedimento especialmente em termos do tempo de resposta.

*Identifique e justifique a Ordem de Crescimento da solução encontrada, em relação ao espaço e ao tempo.*

**Exercício 7**

O procedimento *somas-iguais* tem dois parâmetros, *n* e *soma*, ambos inteiros positivos. A chamada *(soma-iguais 9 40)* determina e devolve o número de hipóteses distintas de adicionar números de 1 a 9, sem repetições, sendo 40 a soma deles.

```
> (soma-iguais 9 40)
3
```

```
40 = 1+4+5+6+7+8+9
40 = 2+3+5+6+7+8+9
40 = 1+2+3+4+6+7+8+9
```

- Apresente uma definição recursiva para determinar o número de hipóteses distintas, de adicionar números de 1 a *n*, sem repetições, perfazendo *soma*.
- Escreva em Scheme o procedimento *somas-iguais* com base na definição apresentada.

Espaço para desenvolver e testar o procedimento pedido.
O ecrã abre vazio.
Experimente aumentar o argumento e tente interpretar o comportamento do procedimento especialmente em termos do tempo de resposta.

*Indique se a recursividade utilizada é linear ou em árvore. Justifique.*

*Identifique e justifique a Ordem de Crescimento da solução encontrada, em relação ao espaço e ao tempo.*

**Exercício 8**

Escrever em Scheme o programa *numero-de-somas-iguais* com três parâmetros, *n*, *lim-inf* e *lim-sup* e que se baseia no procedimento do exercício anterior, *somas-iguais*. Este programa responde do seguinte modo:

```
> (numero-de-somas-iguais 9 20 24)
Numeros de 1 a 9
20: a
21: b
```

```
22: c
23: d
24: e
```

em que *a*, *b*, *c*, *d*, e *e* representam, respectivamente, o número de somas iguais a 20, 21, 22, 23, e 24, conseguidas com os números de 1 a 9.

| Espaço para desenvolver e testar o procedimento pedido.<br>O ecrã abre vazio. |
|---|

**Exercício 9**

Escreva em Scheme um programa designado por *teste-da-tabuada*, com um só parâmetro, *n*, e que põe *n* questões sobre a tabuada de multiplicar (tabuadas de 1 a 10). Os números que surgem nas questões são gerados aleatoriamente.

```
> (teste-da-tabuada 10)

3 x 8 = 24
Resposta certa

2 x 7 = 15
Resposta errada

...
```

*Na primeira questão, o programa visualiza 3 x 8 e o utilizador escreveu 24. Portanto, resposta certa.*

No final das questões, se o número de erros for inferior a 2, a mensagem será:

```
Muito bem
```

Caso contrário será:

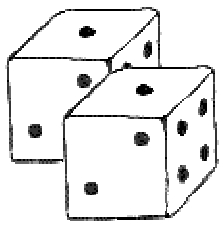
```
Deve estudar melhor a tabuada
```

**Exercício 10**

**Projecto - Um jogo de dados**

Desenvolva um programa em Scheme que simule um jogo de dados, cujas regras são seguidamente apresentadas. Sugere-se que se utilize uma abordagem do tipo *de-cima-para-baixo* apresentada num dos outros módulos.

O jogador lança 2 dados e os números das duas faces são adicionados, originando uma soma que se situa entre 2 e 12.
Se a soma for 7 ou 11, o jogador ganha uma quantia equivalente à soma.
Se a soma for 2, 3, ou 12, o jogador perde a essa mesma quantia.
Mas se a soma for 4, 5, 6, 8, 9, ou 10, então a regra complica-se um pouco. Neste caso, a soma adquire a designação de ponto e a jogada continua, ou seja, o jogador deverá lançar novamente os dois dados e tantas vezes até conseguir obter uma soma igual a:



- ponto e ganha uma quantia equivalente a ponto;
- 7 e perde uma quantia equivalente a ponto.

Apresenta-se agora uma sessão com o programa descrito.

```
> (jogo-de-dados)

jogar (lançar dados: 1; terminar: 0): 1
dados: 6 + 5 = 11
```

```
O jogador ganhou 11 e o saldo e': 11

jogar (lançar dados: 1; terminar: 0): 1
dados: 6 + 6 = 12
O jogador perdeu 12 e o saldo e': -1

jogar (lançar dados: 1; terminar: 0): 1
dados: 4 + 6 = 10
O ponto e' 10 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 2 + 4 = 6
O ponto e' 10 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 6 + 5 = 11
O ponto e' 10 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 6 + 4 = 10
O jogador ganhou 10 e o saldo e': 9

jogar (lançar dados: 1; terminar: 0): 1
dados: 1 + 3 = 4
O ponto e' 4 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 1 + 4 = 5
O ponto e' 4 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 5 + 4 = 9
O ponto e' 4 e continua a jogada.
jogar (lançar dados: 1): 1
dados: 5 + 2 = 7
O jogador perdeu 4 e o saldo e': 5

jogar (lançar dados: 1; terminar: 0): 0
O jogo terminou e o jogador ficou com o saldo: 5
```

Espaço para desenvolver e testar o projecto proposto.
Siga uma aproximação *de-cima-para-baixo*.