Aula Prática 6

Resumo:

- Funções recursivas.

Exercício 6.1

A função de Fibonacci[1] de um número inteiro (não negativo) n pode ser definida por:

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-2) + F(n-1) & \text{se } n > 1 \end{cases} . \tag{6.1}$$

a. Complete a função `fibonacci` e teste-a no programa `Fibonacci`, que escreve os números $F(n)$ para os valores de n dados nos argumentos. Experimente calcular $F(5), F(10), F(20), F(40)$, por exemplo.

b. A implementação recursiva direta desta relação de recorrência é muito ineficiente porque invoca a função repetidamente com os mesmos argumentos. Uma técnica geral que permite colmatar este problema é a chamada *memoização*. Consiste em memorizar os resultados de invocações anteriores numa tabela e devolver o valor memorizado caso volte a ser pedido. Implemente uma versão memoizada da função e confirme o seu desempenho.

Exercício 6.2

Construa uma função recursiva que imprima um array de strings, uma string por linha. O algoritmo é simples: para imprimir N linhas, imprimimos as $N-1$ primeiras e depois a última. Use-a no programa `PrintArgs` para imprimir os argumentos da linha de comando.

Repare que, além do array, é conveniente a função ter um parâmetro extra que permita indicar que parte do array deve imprimir. Este "truque" permite usar o mesmo array em todas as invocações, e variar apenas o segundo parâmetro. Assim não é preciso criar um novo array em cada invocação.

[1] Matemático italiano dos Séculos XII-XIII, responsável, entre outros feitos, pela introdução da chamada numeração árabe na Europa.

Exercício 6.3

Copie o programa anterior para `ReverseArgs.java` e altere a função recursiva por forma a que agora escreva as strings por ordem inversa.

Exercício 6.4

Construa uma função recursiva – `reverseString` – que inverta uma qualquer `String` passada como argumento. Para testar a função, implemente um programa que a aplique a cada um dos argumentos.

Exercício 6.5

Escreva um programa que mostre o conteúdo de um directório e de todos os seus subdirectórios recursivamente.

Por exemplo, se for executado o comando `java -ea ListRec ../aula02` , o resultado deverá assemelhar-se ao seguinte.

```
../aula02
../aula02/pt
../aula02/pt/ua
../aula02/pt/ua/prog2
../aula02/pt/ua/prog2/Contacto.java
../aula02/Complex.java
../aula02/Contacto.java
../aula02/TestContacto.java
(...)
```

Nota: Sugere-se a utilização das funções `listFiles` e `getPath` da classe `File` para obter, respectivamente, a lista de ficheiros existentes num directório e a localização de cada ficheiro.

Exercício 6.6

O programa `Ngrams` tem uma função `all3grams` que permite gerar todos os arranjos possíveis de três símbolos escolhidos de um dado alfabeto. Experimente compilar e correr `java -ea Ngrams ab`.

Crie uma função `allNgrams` que permita generalizar o programa para gerar todos os n-gramas (sequências de n símbolos) possíveis de um dado alfabeto. Um algoritmo recursivo para obter cada um dos n-gramas consiste em obter a lista de todos os $(n-1)$-gramas e a cada um deles acrescentar cada um dos símbolos do alfabeto. Qual será o caso base? E que resultado lhe corresponde?

Exercício 6.7

a. Escreva um programa que encontre numa árvore de directórios todos os ficheiros com um determinado nome.

   Por exemplo, se executar o comando `java -ea FindFile Contacto.java ..` , a saída deverá ser:

```
../aula02/Contacto.java
../aula02/pt/ua/prog2/Contacto.java
```

   Nota: Pode usar a função `getName` da classe `File` para obter o nome do ficheiro.

b. Generalize o programa anterior por forma a encontrar ficheiros que contenham um determinado texto no seu nome.

   Por exemplo, se for executado o comando `java -ea FindFile acto.jav ..` , o resultado deverá incluir os ficheiros anteriores, mas também outro(s).

   Nota: Sugere-se a utilização da função `indexOf` da classe `String` para verificar a ocorrência do texto no nome.

Exercício 6.8

O máximo divisor comum (mdc) de dois números inteiros não negativos a e b pode ser calculado usando o algoritmo de Euclides que se pode expressar pela seguinte definição recursiva:

$$mdc(a,b) = \begin{cases} a & \text{se } b = 0 \\ mdc(b, a \bmod b) & \text{se } b \neq 0 \end{cases} . \tag{6.2}$$

onde o operador mod corresponde à operação *resto da divisão inteira* implementada em `Java` pelo operador `%`.

Escreva uma função que implemente este algoritmo e teste-a num programa simples.

Exercício 6.9

Um cliente de um banco pede um empréstimo de M Euros com uma taxa de juro de $T\%$ ao mês e uma prestação de P Euros no fim de cada mês.

a. Determine a relação de recorrência que descreve o montante em dívida D_n ao fim de n meses.

b. Implemente, com o método iterativo, uma função para determinar D_n.

c. Implemente, com o método recursivo, uma função para determinar D_n.

Por exemplo, com um empréstimo de $M = 1000$ Eur, a uma taxa de $T = 1\%$ e prestação mensal de $P = 20$ Eur, a dívida ao fim de 2 meses pode calcular-se com o comando `java -ea -jar Loan.jar 2 1000 1 20`.

Exercício 6.10

Construa uma função recursiva que determine a chamada distância de Levenshtein entre duas palavras. Esta medida é o menor número de inserções, remoções ou substituições de um carácter necessárias para converter uma palavra na outra.

Por exemplo a distância entre as palavras `"lista"` e `"lata"` é 2, porque se consegue converter `"lista"` em `"lata"` com, no mínimo dos mínimos, duas operações (uma remoção e uma substituição).

Note que qualquer palavra não vazia P pode ser decomposta no seu primeiro carácter C e o resto da palavra S, ou seja: $P = C + S$ (se $\text{length}(P) > 0$).[2] Dessa constatação surge naturalmente a seguinte relação de recorrência para a distância de Levenshtein:

$$d(P_1, P_2) = \begin{cases} \text{length}(P_1) & \text{se length}(P_2) = 0 \\ \text{length}(P_2) & \text{se length}(P_1) = 0 \\ d(S_1, S_2) & \text{se } C_1 = C_2 \quad (*) \\ 1 + min(d(S_1, P_2), d(P_1, S_2), d(S_1, S_2)) & \text{se } C_1 \neq C_2 \quad (*) \end{cases} . \tag{6.3}$$

$(*)$ Nestes casos, obviamente que nem P_1 nem P_2 podem ser vazias.

[2]Em Java, pode determinar `C = P.charAt(0)` e `S = P.substring(1)`.