Aula Prática 7

Resumo:

- Funções recursivas (cont.).

Exercício 7.1

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 uma qualquer palavra não vazia P_k pode ser definida como sendo a junção do seu primeiro carácter c_k com o que resta da palavra S_k (eventualmente poderá até ser um string vazio). Ou seja: $P_k = c_k + S_k$ (se $\text{length}(P_k) > 0$). Dessa constatação surge naturalmente a seguinte relação de recorrência para este problema:

$$d(P_1, P_2) = \begin{cases} \text{length}(P_1) & \text{se length}(P_2) = 0 \\ \text{length}(P_2) & \text{se length}(P_1) = 0 \\ d(S_1, S_2) & \text{se } c_1 = c_2 \quad (*) \\ 1 + min(d(S_1, P_2), d(P_1, S_2), d(S_1, S_2)) & \text{se } c_1 \neq c_2 \quad (*) \end{cases} . \tag{7.1}$$

$(*)$ Nestes casos, obviamente que nenhuma das `Strings` pode ser vazia.

Exercício 7.2

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:

$$mdc(a, b) = \begin{cases} a & \text{se } b = 0 \\ mdc(b, a \bmod b) & \text{se } b \neq 0 \end{cases} . \tag{7.2}$$

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

(O operador mod corresponde à operação *resto da divisão inteira* implementada em `Java` pelo operador `%`.)

Nota: Apesar de $mdc(a, b) = mdc(b, a)$, a ordem com que os parâmetros são passados na chamada recursiva da função é relevante (experimente trocar).

Exercício 7.3

Numa aula anterior, foi-lhe proposto implementar um jogo, em que o seu programa escolhia um número e o utilizador tentava adivinhá-lo. No presente exercício vamos inverter os papeis. Neste caso, o utilizador escolhe o número secreto e o seu programa adivinha-o! O esqueleto do programa é fornecido em anexo, faltando implementar as seguintes funções:

- `getFeedback()` - Esta função lê do teclado e retorna informação do utilizador sobre uma tentativa previamente apresentada pelo programa. Essa informação é dada na forma de um inteiro, que poderá ser:

  0 - Terminação sem sucesso: utilizador perdeu a paciência,

  1 - Terminação com sucesso: programa descobriu o número,

  2 - Número secreto é mais alto, e

  3 - Número secreto é mais baixo.

  Caso a informação obtida seja inválida, deve ler novamente, várias vezes se necessário, até obter uma informação válida. Desenvolva uma implementação recursiva para esta função.

- `iGuessIt(min,max)` - Esta função implementa o processo de adivinhar o número secreto escolhido pelo utilizador. A função vai gerando tentativas entre `min` e `max` e obtém para cada uma delas informação do utilizador (usando a função anterior). Os limites mínimo e máximo devem ser ajustados mediante a informação recebida ao longo do processo. Desenvolva uma implementação recursiva para esta função. A função retorna `true` caso o processo seja concluído com sucesso ou `false` caso contrário.

Exercício 7.4

O função seguinte calcula a soma de um subarray de números reais:

```
// sum of subarray [start,end[ of arr:
static double sum(double[] arr, int start, int end)
{
   assert arr != null;
   assert start >= 0 && start <= end && end <= arr.length;

   double res = 0;
   for(int i = start; i < end; i++)
      res += arr[i];
   return res;
}
```

Implemente uma versão recursiva – `sumRec` – desta função.

Para testar a função, implemente um programa que faça o somatório de todos os seus argumentos.

Exercício 7.5

Implemente um programa que determine e escreva a raiz quadrada de um número real não

negativo utilizando o chamado algoritmo de Babilónia. Este algoritmo gera uma sucessão de aproximações à raiz quadrada de um número N de acordo com a seguinte relação de recorrência:

$$R_i = \begin{cases} inic, & \text{se } i = 0 \\ (R_{i-1} + N/R_{i-1})/2, & \text{se } i > 0 \end{cases} \tag{7.3}$$

em que *inic* é um numero inteiro positivo dado como aproximação inicial ao valor da raiz. O processo termina quando o erro $|N - R_i^2|$ for menor do que um dado ϵ. Na sua implementação, a relação de recorrência deverá ficar traduzida por uma função recursiva que recebe como entradas N, ϵ e *inic* e retorna a raiz quadrada.

Exercício 7.6

Resolva o problema de colocar N super-rainhas num tabuleiro de xadrez ($N \times N$) sem que nenhuma super-rainha ataque qualquer outra. Uma super-rainha tem os poderes de uma rainha normal de xadrez acrescidos com o poder do cavalo.

No problema apresentado na aula teórico-prática das rainhas só poderia haver uma rainha por cada linha, coluna e diagonal. Neste problema, acresce a restrição de nenhuma super-rainha poder atacar outra por um salto de cavalo.

Para visualizar o tabuleiro de xadrez pode (se quiser) fazer uso do pacote `GBoard`. O programa seguinte exemplifica a sua utilização:

```
import static java.lang.System.*;
import pt.ua.gboard.*;
import pt.ua.gboard.games.*;

public class TestChessBoard
{
   public static void main(String[] args)
   {
      ChessBoard cboard = new ChessBoard(8);
      cboard.put(ChessPieceType.WHITE_QUEEN, 0, 0);
      GBoard.sleep(1000); // 1 second
      cboard.remove(0, 0);
      GBoard.sleep(500);
      cboard.put(ChessPieceType.WHITE_QUEEN, 7, 7);
   }
}
```

Exercício 7.7

Modifique o problema anterior por forma a se encontrar (escrever e contar) todas as soluções para o problema de N super-rainhas.