Algoritmos e Estruturas de Dados
Primeiro Teste – Primeiro Teste – Tipo 2
Departamento de Informática, Universidade Nova de Lisboa

Atenção:

- Os Anexos ao teste poderão ser-lhe úteis.
- Leia todas as indicações no seu caderno de teste.

Grupo I – Análise de algoritmos

Considere o método `existsValue` apresentado abaixo.

```
private static boolean existsValue(int[] a, int aim, int low, int high) {
   if (low > high) return false;
   else {
         int mid = ( low + high ) / 2;
         return (a[mid] == aim) || existsValue(a, aim, low, mid-1)
                || existsValue(a, aim, mid+1, high);
        }
}

public static boolean existsValue(int[] a, int aim) {
   return existsValue(a, aim, 0, a.length-1);
}
```

Determine a complexidade temporal do método `existsValue`, no melhor e no pior caso, **justificando**.

Grupo II – Programação em AED

1. Um número inteiro positivo será um Número Perfeito se for igual à soma dos seus divisores próprios (os divisores próprios de um número são todos os seus divisores diferentes do próprio número). Os números 6 e 28 são exemplos de números perfeitos.
   O método `isPerfect` apresentado abaixo verifica se um número inteiro positivo é um número perfeito. Para devolver este resultado, o método auxilia-se do método privado recursivo `sumOfDivisors` que calcula a soma de todos os divisores próprios de um número. A primeira chamada deste método é efetuada a partir do método `isPerfect`, tendo como parâmetros o próprio número (`number`) e o seu maior divisor próprio potencial (`number/2`).
   Desenvolva o método recursivo `sumOfDivisors` e determine a sua complexidade temporal no melhor caso, no pior caso e no caso esperado, **justificando**.

   Nota: Implemente a solução em código Java. Apenas as soluções recursivas serão avaliadas.

(A pergunta 1 do grupo 2 continua na página 2)

```
private static int sumOfDivisors(int number, int divisor) {
   ...
}

//Requires: a >= 1
public static boolean isPerfect(int number) {
   if (number == 1)
     return false;
   return sumOfDivisors(number, number / 2 ) == number;
}
```

2. Considere a classe de lista duplamente ligada `DoublyLinkedList` apresentada nas aulas, concentrando-se no seu método público `append()`. Este método recebe, como parâmetro, uma lista duplamente ligada (`list`), remove todos os elementos contidos na mesma e insere-os no final da lista corrente (`this`). Implemente este método com a melhor complexidade temporal e tendo em conta todas as situações de excepção possíveis. Deve ainda determinar a complexidade temporal do método no melhor caso, no pior caso e no caso esperado, **justificando**.

```
package dataStructures;

public class DoublyLinkedList<E> implements List<E>{

   // Node at the head of the list.
   protected DListNode<E> head;
   // Node at the tail of the list.
   protected DListNode<E> tail;
   // Number of elements in the list.
   protected int currentSize;

. . .

   // Removes all of the elements from the specified list and
   // inserts them at the end of the list (in proper sequence).
   public void append( DoublyLinkedList<E> list )
      ...
   }

}
```

Grupo III – Tipos Abstratos de Dados e Estruturas de Dados

Neste grupo deve analisar o problema apresentado e conceber uma solução para o problema completo de acordo com os requisitos descritos. Deverá depois responder a cada uma das questões, de acordo com a solução que concebeu.
Nas suas respostas às questões 4 e 5, não deve desenvolver código em java, apenas fazer uma descrição da implementação das operações pedidas, de acordo com a sua escolha de estruturas de dados e variáveis de instância.

(O Grupo III continua na pagina 3)

O Tipo Abstrato de Dados (TAD) `PropertyRenting` define as operações associadas aos pagamentos de aluguer de **uma única** propriedade numa Imobiliária. A Propriedade deve manter informação relativa à localização, preço diário de aluguer e custo diário de manutenção da mesma. O custo de manutenção é cobrado apenas quando a propriedade está alugada. Deve ser possível atualizar o preço diário de aluguer da propriedade e listar os valores recebidos no final dos alugueres. Além disso, uma implementação do TAD deve permitir saber o valor total dos pagamentos recebidos assim como o lucro total obtido, que resulta de descontar os custos de manutenção. A listagem dos pagamentos deve ser efetuada por ordem cronológica, iniciando no aluguer mais antigo e terminando no mais recente.

Apresenta-se abaixo o interface `PropertyRenting`:

```
import dataStructures.Iterator;

interface PropertyRenting {

  // Devolve descrição da localização da propriedade.
  String getLocation();

  //Devolve preço diário de aluguer da propriedade.
  double getDailyPrice();

  //Devolve custo de manutenção diário da propriedade, quando alugada.
  double getDailyMaintenanceCosts();


  //Altera o preço diário da aluguer da propriedade (para price).
  void setPrice(double price);

  //Insere pagamento de aluguer da propriedade (relativo ao número de dias
  //guardado em numberOfDays.
  void addPayment(int numberOfDays);

  //Devolve iterador (de valores monetários) com os pagamentos recebidos
  //ao aluguer da propriedade. A iteração executada a partir deste iterador
  //deve apresentar os valores associados aos alugueres por ordem cronológica,
  //iniciando na mais antiga e terminando na mais recente.
  //Requires: a Propriedade terá de ter sido alugada pelo menos uma vez.
  Iterator<Double> listPayments();

  //Devolve valor total de pagamentos recebidos.
  double getTotalPayments();

  //Devolve o lucro total obtido, depois de subtraído os custos de manutenção.
  double getTotalProfit();
}
```

Relativamente ao interface apresentado, reflita sobre a melhor implementação para o mesmo e responda às seguintes questões **(separadamente)**:

1. Proponha as necessárias variáveis de instância e respetivos tipos para apoiar implementação dos métodos **`getLocation()`, `getDailyPrice()`, `getDailyMaintenanceCosts()`** e **`setPrice()`.**
2. Proponha uma Estrutura de Dados para apoiar a implementação dos métodos **`addPayment()`** e **`listPayments()`**. A sua resposta deve incluir:

a) **Tipo Abstrato de Dados genérico (TAD);**
b) **Estrutura de Dados(ED) genérica que deverá ser usada para implementar o TAD;**
c) **Tipo de dados (do problema) a guardar dentro da EDs (Tipo do Elemento (E); ou tipos associados ao par Entry<K,V>) e respetivo significado. Deve sempre justificar esta escolha.**
NOTA: Tenha em conta que, quando a escolha recai sobre um dicionário, será necessário escolher o tipo da Chave (K) e o tipo do Valor associado (V).

3. Proponha ainda, se achar conveniente, variáveis de instância adicionais, para apoiar a implementação dos métodos **`getTotalPayments()`** e **`getTotalProfit()`**.

4. Considerando as EDs e variáveis de instância propostas em 1, 2 e 3, descreva brevemente como implementaria a operação **`addPayment()`**, considerando que:
   - a operação **`listPayments()`** deverá devolver um iterador da ED proposta em 2.
   - a execução das operações **`getTotalPayments()`** e **`getTotalProfit()`** deve ter complexidade constante (O(1)), em todos os casos.

   Estude ainda a complexidade temporal da operação `addPayment()` no melhor caso, no pior caso e no caso esperado, justificando.

5. Com base nas EDs e variáveis de instância propostas em 1, 2 e 3, descreva brevemente como implementaria a operação **`getTotalProfit()`.**
   Estude ainda a complexidade temporal da operação `getTotalProfit()` no melhor caso, no pior caso e no caso esperado, justificando.

(Os anexos ao teste estão nas páginas seguintes)

Anexo A - Recorrências

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(n-1) + c & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

Recorrência 2a)

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(\frac{n}{2}) + O(1) & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

com $a \geq 0, \quad b = 1, 2$ **constantes**

Recorrência 2b)

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(\frac{n}{c}) + O(n) & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$

Anexo B – Interfaces e Classes de Apoio

```
public interface Comparable<T>{
   int compareTo( T object );
}
public interface Stack<E>{
   boolean isEmpty( );
   int size( );
   E top( ) throws EmptyStackException;
   void push( E element );
   E pop( ) throws EmptyStackException;
}
public interface Queue<E> {
   boolean isEmpty( );
   int size( );
   void enqueue( E element );
   E dequeue( ) throws EmptyQueueException;
}
public interface Iterator<E> {
   boolean hasNext( );
   E next( ) throws NoSuchElementException;
   void rewind( );
}
public interface List<E> {
   boolean isEmpty( );
   int size( );
   Iterator<E> iterator( );
   E getFirst( ) throws EmptyListException;
   E getLast( ) throws EmptyListException;
   E get( int position )
      throws InvalidPositionException;
   int find( E element );
   void addFirst( E element );
   void addLast( E element );
   void add( int position, E element )
      throws InvalidPositionException;
   E removeFirst( ) throws EmptyListException;
   E removeLast( ) throws EmptyListException;
   E remove( int position )
      throws InvalidPositionException;
   boolean remove( E element );
}
public interface Entry<K,V>{
   K getKey( );
   V getValue( );
}
public interface Dictionary<K,V>{
   boolean isEmpty( );
   int size( );
   Iterator<Entry<K,V>> iterator( );
   V find( K key );
   V insert( K key, V value );
   V remove( K key );
}
```

```
class DListNode<E> implements Serializable {
   public DListNode( E theElement, DListNode<E>
            thePrevious, DListNode<E> theNext );
   public DListNode( E theElement );
   public E getElement( );
   public DListNode<E> getPrevious( );
   public DListNode<E> getNext( );
   public void setElement( E newElement );
   public void setPrevious
                  ( DListNode<E> newPrevious );
   public void setNext( DListNode<E> newNext );
}
public class DoublyLinkedList<E>
                           implements List<E> {
   public boolean isEmpty( );
   public int size( );
   public Iterator<E> iterator( );
   public E getFirst( )
      throws EmptyListException;
   public E getLast( )
      throws EmptyListException;
   protected DListNode<E> getNode
                              ( int position );
   public E get( int position )
      throws InvalidPositionException;
   public int find( E element );
   public void addFirst( E element );
   public void addLast( E element );
   protected void addMiddle
                   ( int position, E element );
   public void add( int position, E element )
      throws InvalidPositionException;
   protected void removeFirstNode( );
   public E removeFirst( )
      throws EmptyListException;
   protected void removeLastNode( );
   public E removeLast( )
      throws EmptyListException;
   protected void removeMiddleNode
                         ( DListNode<E> node );
   public E remove( int position )
      throws InvalidPositionException;
   protected DListNode<E> findNode( E element );
   public boolean remove( E element );
   public void append
                  ( DoublyLinkedList<E> list );
}
```