Algoritmos e Estruturas de Dados 2016/17
Primeiro Teste – 24 de outubro de 2016
Departamento de Informática, Universidade Nova de Lisboa

Atenção: Os Anexos ao teste poderão ser-lhe úteis.

1. Considere o método `fiboTail` apresentado abaixo.

```
protected static int fiboTail(int rabbits, int a, int b){
   if (rabbits == 0) return a;
   if (rabbits == 1) return b;
   return fiboTail(rabbits - 1, b, a + b);
}

//requires: rabbits >= 0
public static int fiboTail(int rabbits){
   return fiboTail(rabbits, 0, 1);
}
```

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

2. A atualização anual do saldo de uma conta bancária implica o cálculo de juro incremental sobre o saldo atual. Veja a fórmula genérica (1) e o exemplo na tabela 1 (para uma conta com 10% de juro anual). O investidor abre uma conta com 100 Euros e no 1º aniversário dessa data, vê o saldo da conta incrementado para 110 Euros, o valor do investimento inicial somado a 10% de juro. No 2º aniversário da abertura de conta, também a taxa de juro de 10% é aplicada, mas agora sobre o saldo de 110 Euros do ano transato.

   A fórmula recursiva de cálculo no final de *(n)* anos de investimento será a seguinte:

$$saldo(n) = saldo(n-1) + saldo(n-1) \times tx.juro \quad (1)$$

Ano	Saldo (€)
0	100
1	110
2	121
3	133.1

Tabela 1

Desenvolva o método recursivo `calculateBalance()` que calcula o saldo da conta ao fim de um número *(n)* de anos, tendo como parâmetros, além do número de anos, o capital investido inicialmente (saldo no ano 0) e a taxa de juro (ver assinatura do método na página seguinte).

(O enunciado da pergunta 2 continua na página 2)

```
//Requires: n >= 0, amount > 0, interest >= 0
public static double calculateBalance(int n, double amount, double interest ){

    ...

}
```

Uma vez desenvolvido o método recursivo pedido, determine a sua complexidade temporal do método, no melhor caso, no pior caso e no caso esperado, **justificando**.
Nota: Implemente a solução em código Java. Apenas as soluções recursivas serão avaliadas.

3. Considere a classe de lista duplamente ligada `DoublyLinkedList` apresentada nas aulas, concentrando-se no seu método público de inserção à cauda da lista (assinatura apresentada abaixo), o **método `addLast()`**. Este método recebe, como parâmetro, um elemento genérico (do tipo `E`) a ser inserido na última posição da lista e deve incluir o seguinte:

   - Criar um novo nó da lista com o elemento recebido como parâmetro;
   - Criar as ligações necessárias entre o novo nó e os restantes nós da lista;
   - Alterar variáveis de instância da lista que necessitem de alteração de estado, devido à inserção efetuada (incluindo o número de elementos contidos na lista).

   Implemente o método `addlast()` e inclua comentários para cada linha, a explicar o objetivo de cada instrução do método. 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;

. . .

   // Inserts the specified element at the last position in the
   // list.
   public void addLast( E element ){
      ...
   }

}
```

if (tail == null)
    head = null;

(O teste continua na página 3)

4. O Tipo Abstrato de Dados (TAD) CalorieConsumer define as operações associadas a um indivíduo que pretende manter um registo das calorias que vai consumindo em refeições, ao longo de um período de tempo não especificado. Este TAD deve manter informação pessoal sobre o indivíduo (como o nome e a idade) e deve permitir a atualização dos dados relativos ao seu consumo de calorias, como a inserção de refeições e o acesso a algumas estatísticas (como o total de calorias consumido e o valor da refeição mais calórica consumida até ao momento). A partir deste TAD deve ainda ser possível listar as calorias de todas as refeições consumidas pelo consumidor até ao momento, por ordem cronológica, iniciando na mais recente e terminando na mais antiga.

Apresenta-se abaixo o interface CalorieConsumer:

```
import dataStructures.Iterator;

interface CalorieConsumer {

    //Devolve o nome do consumidor.
    String getName();

    //Devolve a idade do consumidor.
    int getAge();

    //Incrementa a idade do consumidor, somando 1.
    void incrementAge();

    //Adiciona as calorias de uma refeição ao consumidor.
    //Requires: mealCalories > 0
    void addMeal(int mealCalories) throws NotValidException;

    //Devolve iterador (de calorias) das refeições do consumidor.
    // A iteração executada a partir deste iterador deve apresentar
    // as (calorias associadas às) refeições por ordem cronológica, iniciando
    // na refeição mais recente e terminando na mais antiga.
    //Requires: O consumidor tem de ter consumido (pelo menos) uma refeição
    Iterator<Integer> listMeals() throws EmptyMealsException;

    //Devolve total de calorias consumido pelo consumidor.
    int getCalories();

    //Devolve o valor de calorias da refeição mais calórica consumida
    //pelo consumidor.
    //Requires: O consumidor tem de ter consumido (pelo menos) uma refeição
    int getHighestMeal() throws EmptyMealsException;

}
```

Relativamente ao problema proposto:

- Explicite detalhadamente as estruturas de dados e as variáveis de instância mais adequadas para implementar a interface calorieConsumer;
- Descreva brevemente como implementaria as operações addMeal(), listMeals(), getCalories() e getHighestMeal() e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando. **Não deve desenvolver código em Java, apenas fazer uma descrição da implementação das operações de acordo com a sua escolha de estruturas de dados e variáveis de instância.**