Exame de Análise e Desenho de Algoritmos

Duração: 3 horas

Departamento de Informática, FCT NOVA

28 de Junho de 2017

Número: _____ **Nome:** _____

1. [3 valores] Os vetores `ufA.partition` e `ufB.partition` contêm o estado de duas partições.

   (a) Assuma que a classe da partição `ufA` implementa **reunião por tamanho** e **representante com compressão do caminho**.
   Indique o conteúdo do vetor `ufA.partition` após a execução de cada um dos seguintes métodos, executados em sequência pela ordem indicada.

   `ufA.partition:`

−4	0	0	2	−5	4	4	6	4
0	1	2	3	4	5	6	7	8

   Depois da execução de `ufA.find(3)`:

   `ufA.partition:`

0	1	2	3	4	5	6	7	8

   Depois da execução de `ufA.union(0, 4)`:

   `ufA.partition:`

0	1	2	3	4	5	6	7	8

   (b) Assuma que a classe da partição `ufB` implementa **reunião por nível** e **representante com compressão do caminho**.
   Indique o conteúdo do vetor `ufB.partition` após a execução de cada um dos seguintes métodos, executados em sequência pela ordem indicada.

   `ufB.partition:`

−3	0	0	2	−3	4	4	6	4
0	1	2	3	4	5	6	7	8

   Depois da execução de `ufB.find(3)`:

   `ufB.partition:`

0	1	2	3	4	5	6	7	8

   Depois da execução de `ufB.union(0, 4)`:

   `ufB.partition:`

0	1	2	3	4	5	6	7	8

2. [3.5 valores] A classe *QueueIn2Stacks* implementa filas com disciplina FIFO, de elementos do tipo E, com duas pilhas. Esta classe usa a interface *Stack* e a classe *StackInLinkedList*, que implementa uma pilha com uma lista ligada.

```
public class QueueIn2Stacks<E> {

    private Stack<E> inS;  // Stack with the most recent elements
    private Stack<E> outS; // Stack with the oldest elements

    public QueueIn2Stacks( ) {
        inS = new StackInLinkedList<E>();
        outS = new StackInLinkedList<E>();
    }

    public boolean isEmpty( ) {
        return inS.isEmpty() && outS.isEmpty();
    }

    public void enqueue( E element ) {
        inS.push(element);
    }

    public E dequeue( ) throws EmptyQueueException {
        if ( this.isEmpty() )
            throw new EmptyQueueException();

        if ( outS.isEmpty() )
            do
                outS.push( inS.pop() );
            while ( !inS.isEmpty() );

        return outS.pop();
    }

}
```

Considere a função $\Phi(F)$, que atribui a cada fila F da classe *QueueIn2Stacks* o número de elementos guardados na pilha F`.inS`:

$$\Phi(F) = F\texttt{.inS.size()}\,.$$

Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *isEmpty*, *enqueue* e *dequeue*, justificando. Assuma que os métodos *isEmpty*, *push* e *pop* da classe *StackInLinkedList* têm complexidade constante. No estudo da complexidade amortizada do método *dequeue*, assuma que não é levantada a exceção, mas analise separadamente os casos em que a pilha `outS`: não está vazia; está vazia.

3. [3.5 valores] Considere a seguinte função recursiva, $f_X(i,j)$, onde:

   - $X = (x_0, x_1, \ldots, x_n)$ é uma sequência de inteiros (com $n \geq 1$) cujo primeiro elemento é inferior ou igual aos restantes;
   - i é um inteiro entre 0 e n;
   - j é 0 ou 1.

$$f_X(i,j) = \begin{cases} 1, & \text{se } i = 0 \text{ e } 0 \leq j \leq 1; \\ 1 + \max\limits_{0 \leq k < i \,\wedge\, x_k \leq x_i} f_X(k,0), & \text{se } i \geq 1 \text{ e } j = 0; \\ \max\Big(f_X(i,0),\ f_X(i-1,1) \Big), & \text{se } i \geq 1 \text{ e } j = 1. \end{cases}$$

   Repare que $f_{(1,9,4,4)}(3,0) = 1 + \max\Big(f_{(1,9,4,4)}(0,0),\ f_{(1,9,4,4)}(2,0) \Big)$. Optou-se por escrever $f_X(i,j)$, em vez de $f(X,i,j)$, porque X não varia entre chamadas recursivas.

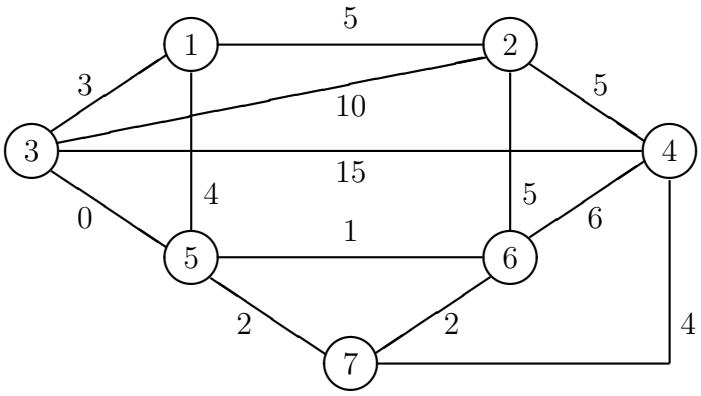
   Apresente um algoritmo iterativo, desenhado segundo a técnica da programação dinâmica, que recebe uma sequência $X = (x_0, x_1, \ldots, x_n)$ de inteiros (com $n \geq 1$) cujo primeiro elemento é inferior ou igual aos restantes e calcula o valor de $f_X(n,1)$. Estude (justificando) as complexidades temporal e espacial do seu algoritmo.

4. [3.5 valores] O **Problema do Carteiro** formula-se da seguinte forma.
   Dados:

   - um grafo $G = (V, A)$ não orientado e pesado, cujos arcos têm custo não negativo,
   - um subconjunto não vazio X de V e
   - um inteiro positivo M,

   existe uma sequência $v_1\, v_2 \cdots v_n$ de vértices (com $|X| \leq n \leq |A|$) que verifica as duas seguintes propriedades?

   - $v_1\, v_2 \ldots v_n\, v_1$ é um caminho em G cujo comprimento pesado não excede M.
   - Todos os vértices de X ocorrem em $v_1\, v_2 \cdots v_n$ (ou seja, $X \subseteq \{v_1, v_2, \ldots, v_n\}$).

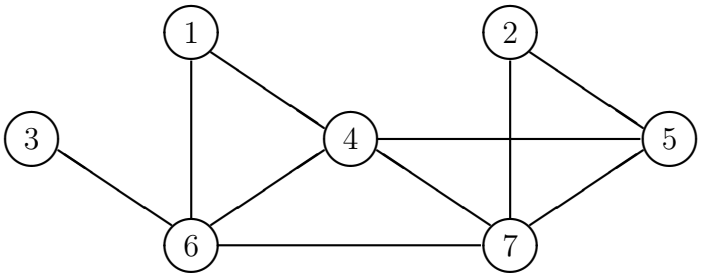


   Considere a instância $(G',\ \{2,4,5,7\},\ 23)$, onde G' é o grafo esquematizado na figura. Por exemplo, a sequência 5 7 6 4 2 6 verifica as propriedades pretendidas:

   - 5 7 6 4 2 6 5 é um caminho em G cujo comprimento pesado é 21.
   - Os vértices 2, 4, 5 e 7 ocorrem em 5 7 6 4 2 6.

   Prove que o Problema do Carteiro é NP-completo.

5. [3.5 valores] Seja G um grafo não orientado, não pesado e conexo. Dados dois vértices v e w, a *distância* entre v e w é o comprimento dos caminhos mais curtos de v para w. O *diâmetro* de G é a maior distância entre dois vértices do grafo.



No grafo esquematizado na figura, a distância entre os vértices 3 e 5 é 3 (um dos caminhos mais curtos é 3 6 4 5). O diâmetro do grafo também é 3 porque (facilmente se verifica que) a distância entre dois quaisquer vértices é inferior ou igual a 3.

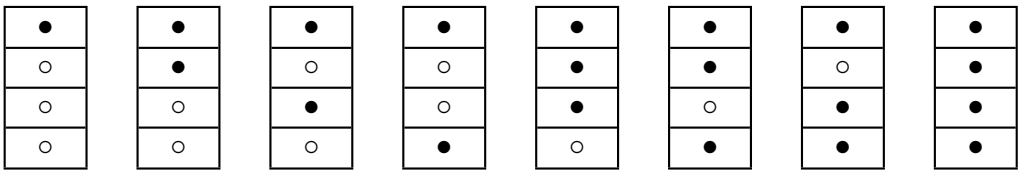
Apresente uma função (em pseudo-código) que recebe:

um grafo não orientado, não pesado e conexo (com pelo menos dois vértices)

e retorna o diâmetro do grafo. Se quiser recorrer a algum algoritmo de grafos estudado, exatamente como ele foi definido, não precisa de o copiar: chame a função respetiva. Se quiser fazer alterações, escreva o código todo. Indique as estruturas de dados que usaria para implementar o grafo e calcule (justificando) a complexidade temporal do seu algoritmo, no pior caso.

6. [3 valores] Considere uma cómoda com gavetas. Cada gaveta tem uma fechadura, que pode estar **aberta** ou **fechada** (mas não pode estar aberta e fechada). Sabe-se que a gaveta de cima da cómoda está fechada e, como é a gaveta do topo, nunca conseguimos aceder ao seu conteúdo. Portanto, a gaveta de cima da cómoda está sempre **trancada**. Qualquer uma das outras gavetas só está **trancada** se estiver fechada e se a gaveta imediatamente acima também estiver fechada. Pretende-se saber em quantas configurações uma cómoda com n gavetas tem t gavetas trancadas.

A figura apresenta as 8 configurações possíveis de uma cómoda com 4 gavetas, onde "•" indica que a gaveta está fechada e "○" indica que a gaveta está aberta. Por baixo de cada configuração está o número de gavetas trancadas. Conclui-se que há 3 configurações em que uma cómoda com $n = 4$ gavetas tem $t = 2$ gavetas trancadas.



Apresente **uma função matemática recursiva** que, com base em dois números inteiros positivos,

$$n \text{ e } t \qquad (\text{com } n \geq t),$$

calcula o número de configurações em que uma cómoda com n gavetas tem t gavetas trancadas (sabendo-se que a gaveta de cima está fechada e trancada). Indique claramente o que representa cada uma das variáveis que utilizar e explicite a chamada inicial (a chamada que resolve o problema).