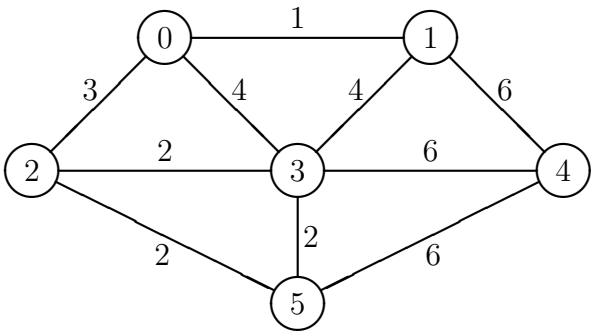
Exame de Análise e Desenho de Algoritmos

Departamento de Informática

Universidade Nova de Lisboa

26 de Junho de 2015

1. Suponha que se executa o algoritmo de Prim com o grafo G esquematizado na figura.



Assumindo que a origem é o vértice 3 (ou seja, que o método $G.aNode()$ retorna 3), indique:

   (a) [2 valores] uma ordem pela qual os arcos podem ser inseridos no resultado (i.e., no vetor *mst*);

   (b) [1 valor] o número total de vezes que o método *decreaseKey* é executado.

2. [2.5 valores] O vetor *uf.partition* contém o estado da partição *uf* (i.e., o estado da instância *uf* da classe *UnionFindInArray*). Assuma que se adotou união por nível e representante com compressão do caminho.

uf.partition:	−4	0	0	2	0	4	4	6	−3	8	8	10
	0	1	2	3	4	5	6	7	8	9	10	11

Indique o estado da partição (ou seja, o conteúdo do vetor *uf.partition*) após a execução **de cada um** dos seguintes métodos, pela ordem indicada:

$$\textit{uf.find}(7), \qquad \textit{uf.union}(8,\, 0) \qquad \text{e} \qquad \textit{uf.find}(10)\ .$$

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 não vazia de inteiros positivos, i é um inteiro entre 0 e n, e j é um inteiro não negativo.

$$f_X(i,j) = \begin{cases} 1, & \text{se } i = 0 \text{ e } j \leq x_0; \\ 0, & \text{se } i = 0 \text{ e } j > x_0; \\ 1, & \text{se } i > 0 \text{ e } j = 0; \\ f_X(i-1,j) + f_X(i,j-1), & \text{se } i > 0 \text{ e } 0 < j \leq x_i; \\ f_X(i-1,j) + f_X(i,j-1) - f_X(i-1,j-x_i-1), & \text{se } i > 0 \text{ e } j > x_i. \end{cases}$$

Apresente um algoritmo iterativo, desenhado segundo a técnica da programação dinâmica, que recebe:

uma sequência não vazia $X = (x_0, x_1, \ldots, x_n)$ de inteiros positivos e um inteiro positivo m

e calcula o valor de $f_X(n, m)$. Estude (justificando) as complexidades temporal e espacial do seu algoritmo.

4. [3.5 valores] Considere a classe *OrderedStack*, de filas ordenadas com disciplina LIFO, de elementos do tipo E. A operação *orderedPush*(*elem*) desempilha todos os elementos menores que *elem* e empilha o elemento *elem* no topo da pilha. A operação *pop*() desempilha e retorna o elemento no topo da pilha.

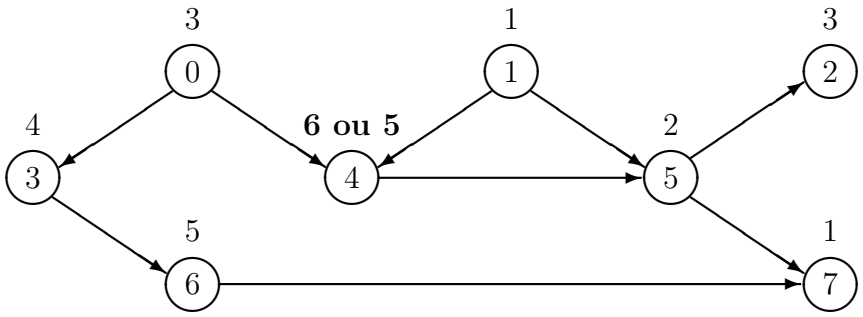
```
public class OrderedStack<E extends Comparable<? super E>> {

    private Deque<E> stack;

    public OrderedStack( ) {
        stack = new LinkedList<E>();
    }

    public int size( ) {
        return stack.size();
    }

    public void orderedPush( E element ) {
        while ( !stack.isEmpty() && stack.getFirst().compareTo(element) < 0 )
            stack.removeFirst();
        stack.addFirst(element);
    }

    public E pop( ) throws RuntimeException {
        if ( stack.isEmpty() )
            throw new RuntimeException();
        return stack.removeFirst();
    }

    @SuppressWarnings("unchecked")
    public E[] getElements( ) {
        E[] elements = (E[]) new Comparable[this.size()];
        int pos = 0;
        for ( E elem : stack )
            elements[pos++] = elem;
        return elements;
    }

}
```

Considere a função $\Phi(S)$, que atribui a cada objeto S da classe *OrderedStack* o número de elementos guardados na pilha:

$$\Phi(S) = S.\text{size}() \,.$$

Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *size*, *orderedPush*, *pop* e *getElements*, justificando. No estudo da complexidade amortizada do método *pop*, assuma que não é levantada a exceção. Assuma que o método *compareTo* da classe dos elementos do tipo E tem complexidade constante (e que as operações efetuadas sobre o atributo *stack* têm a complexidade habitual em listas duplamente ligadas com cabeça e cauda).

5. [4 valores] Considere um grafo orientado, pesado e acíclico, onde cada vértice representa uma tarefa de um projeto. A existência de um arco do vértice v para o vértice w indica que a tarefa v terá de estar concluída antes de se iniciar a tarefa w. O peso (positivo) de um vértice é a prioridade da tarefa; quanto menor for o peso, mais prioritária é a tarefa. As tarefas vão ser executadas sequencialmente (i.e., em cada momento, apenas uma tarefa estará a ser executada). Se houver várias tarefas que podem ser executadas, porque as que as precedem já foram concluídas, executa-se uma das tarefas mais prioritárias. Pretende-se descobrir se a ordem de execução das tarefas é única.



Para exemplificar, considere os grafos esquematizados na figura, onde a única diferença é o peso do vértice 4.

- No grafo em que o peso do vértice 4 é **6**, a ordem de execução das tarefas tem de ser 1, 0, 3, 6, 4, 5, 7, 2 (como se justifica resumidamente a seguir). Logo, a ordem é única.
  - Tarefa 1: porque é mais prioritária que a tarefa 0.
  - Tarefa 0: porque é a única que pode ser executada.
  - Tarefa 3: porque é mais prioritária que a tarefa 4.
  - Tarefa 6: porque é mais prioritária que a tarefa 4.
  - Tarefa 4: porque é a única que pode ser executada.
  - Tarefa 5: porque é a única que pode ser executada.
  - Tarefa 7: porque é mais prioritária que a tarefa 2.
  - Tarefa 2: porque é a única que pode ser executada.
- No grafo em que o peso do vértice 4 é **5**, a ordem de execução das tarefas pode ser 1, 0, 3, 6, 4, 5, 7, 2 (como no caso anterior) ou 1, 0, 3, 4, 5, 2, 6, 7 (pela justificação que se segue). Logo, a ordem não é única.
  - Tarefas 1, 0 e 3: como no caso anterior.
  - Tarefa 4: é uma das alternativas, porque é tão prioritária como a tarefa 6.
  - Tarefa 5: porque é mais prioritária que a tarefa 6.
  - Tarefa 2: porque é mais prioritária que a tarefa 6.
  - Tarefa 6: porque é a única que pode ser executada.
  - Tarefa 7: porque é a única que pode ser executada.

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

- um grafo G orientado e acíclico, com a informação sobre as tarefas do projeto, e
- um vetor *priority* de inteiros positivos, tal que *priority*$[v]$ tem a prioridade da tarefa v,

e retorna *true* se, e só se, a ordem de execução das tarefas for única. Estude (justificando) a complexidade temporal do seu algoritmo, no pior caso.

6. [3.5 valores] Um *palíndromo* é uma sequência de caracteres, $c_1\, c_2\, \cdots\, c_{n-1}\, c_n$ (com $n \geq 1$), que é igual à sua inversa, ou seja, $c_1\, c_2\, \cdots\, c_{n-1}\, c_n = c_n\, c_{n-1}\, \cdots\, c_2\, c_1$.

   Dada uma sequência não vazia de caracteres, S, pretende-se determinar o comprimento das maiores subsequências palíndromas de S.

   Para exemplificar, considere a sequência $A =$ `amxoraromhaah`. A sequência `amorroma` é uma subsequência palíndroma de A de comprimento 8. Repare que `amorroma` pode ser obtida pela concatenação dos caracteres sublinhados de `amxoraromhaah`. O comprimento das maiores subsequências palíndromas de A é 9 (porque a maior subsequência palíndroma de A é `amoraroma`).

   Apresente **uma função recursiva** que, com base:

   - numa sequência $S = x_1\, x_2\, \cdots\, x_n$ de caracteres, com $n \geq 1$,

   calcula o comprimento das maiores subsequências palíndromas de S. Indique claramente o que representa cada uma das variáveis que utilizar e explicite a chamada inicial (a chamada que resolve o problema).