Exame de Análise e Desenho de Algoritmos

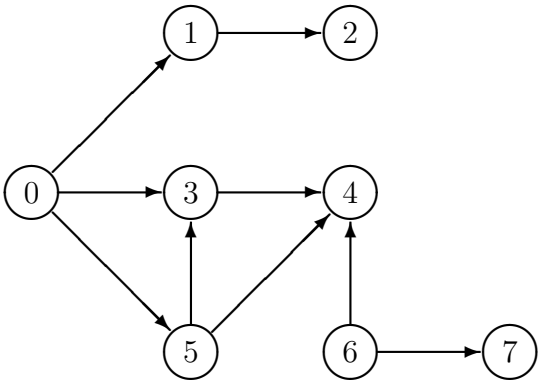
Departamento de Informática da FCT NOVA

23 de Junho de 2016

Responda a **perguntas** diferentes em **folhas** diferentes.

Se precisar de folhas, peça ao docente.

1. [3 valores] Suponha que se executa o algoritmo *topologicalSort* com o grafo G esquematizado na figura depois de se ter alterado a disciplina do saco *ready* para *first-in first-out*.



   Assuma que os métodos *nodes* e *outAdjacentNodes* iteram sempre os vértices por ordem crescente. Por exemplo, $G.outAdjacentNodes(0)$ produz os vértices 1, 3 e 5 (por esta ordem). Indique:

   - A permutação de vértices computada (ou seja, o conteúdo do vetor retornado).
   - O maior número de vértices presentes simultaneamente na fila (ou seja, o maior valor de *ready.size*() durante a execução do algoritmo).

2. [3.5 valores] Uma *pirâmide de base* n (com $n \geq 1$) é uma tabela com n linhas e n colunas, onde a linha k só tem informação útil nas colunas $0, 1, \ldots, k$ (para qualquer $k = 0, 1, \ldots, n-1$).

   A tabela da direita é uma pirâmide de base 5.

	0	1	2	3	4
0	7				
1	3	9			
2	8	1	-1		
3	2	7	4	4	
4	0	5	2	-6	5

   Considere a seguinte função recursiva, $f_T(i, j)$, onde: T é uma pirâmide de números inteiros de base $n \geq 1$; i é um inteiro entre 0 e $n-1$; j é um inteiro entre 0 e i.

$$f_T(i,j) = \begin{cases} T[i][j], & \text{se } i = n-1 \text{ e } 0 \leq j \leq i; \\ T[i][j] + \max(\ f_T(i+1, j),\ f_T(i+1, j+1)\ ), & \text{se } i < n-1 \text{ e } 0 \leq j \leq i. \end{cases}$$

   Optou-se por escrever $f_T(i, j)$ em vez de $f(T, i, j)$, porque T não varia entre chamadas recursivas.

   Apresente um algoritmo iterativo, desenhado segundo a técnica da programação dinâmica e implementado em Java, que recebe:

   uma pirâmide de números inteiros de base n (com $n \geq 1$)

   e calcula o valor de $f_T(0, 0)$. Estude (justificando) as complexidades temporal e espacial do seu algoritmo.

```
import java.util.Random;

public class IntIterator  {
    private int[] sequence;    // Elements to return are in the first currentSize positions.
    private int currentSize;   // Number of elements to return.
    private int halfLength;    // Half of sequence.length.

    public IntIterator( int exponent )  {
        int length = (int) Math.pow(2, exponent);
        sequence = new int[length];
        for ( int i = 0;  i < length;  i++ )
            sequence[i] = i;
        this.permute();
        currentSize = length;
        halfLength = length / 2;
    }

    private void permute( )  {
        Random generator = new Random();
        for ( int i = 0;  i < sequence.length - 1;  i++ )  {
            int pos = i + generator.nextInt(sequence.length - i);
            // Swap elements at positions i and pos.
            int aux = sequence[i];
            sequence[i] = sequence[pos];
            sequence[pos] = aux;
        }
    }

    public boolean hasNext( )  {
        return currentSize > 0;
    }

    public int next( ) throws NoSuchElementException  {
        if ( !this.hasNext() )
            throw new NoSuchElementException();

        currentSize--;
        int element = sequence[currentSize];
        if ( currentSize == halfLength  &&  currentSize >= 2 )
            this.shrinkArray();
        return element;
    }

    private void shrinkArray( )  {
        int[] newArray = new int[halfLength];
        for ( int i = 0;  i < halfLength;  i++ )
            newArray[i] = sequence[i];
        sequence = newArray;
        halfLength = halfLength / 2;
    }

}
```

3. [3.5 valores] A página anterior contém a classe *IntIterator*, que permite iterar os números $0, 1, 2, 3, \ldots, 2^k - 1$ (onde $k \geq 0$), por uma ordem aleatória.

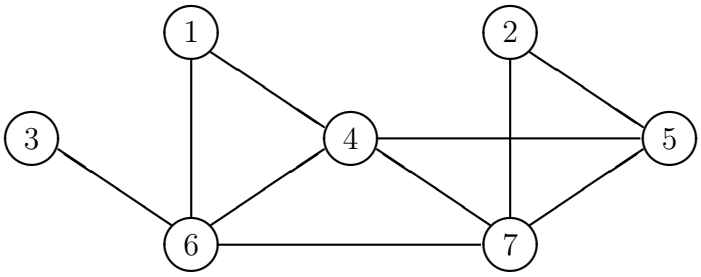
   Considere a função $\Phi(I)$, que atribui a cada objeto I da classe *IntIterator* o número de "posições livres" no vetor *sequence*:

   $$\Phi(I) = I.\text{sequence.length} - I.\text{currentSize}\,.$$

   Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *hasNext* e *next*, justificando. No estudo da complexidade amortizada do método *next*, assuma que não é levantada a exceção, mas analise separadamente os casos em que a condição do segundo *if*: é falsa (e o método *shrinkArray* não é executado); é verdadeira (e o método *shrinkArray* é executado).

4. [3.5 valores] Seja G um grafo não orientado e conexo. Um *caminho Hamiltoniano* em G é uma permutação dos vértices do grafo que é um caminho no grafo.

   Por exemplo, 1 4 5 2 7 6 3 é um caminho Hamiltoniano no grafo esquematizado na figura.



   O **Problema do Caminho Hamiltoniano** formula-se da seguinte forma.
   Dado um grafo G, não orientado e conexo, existe um caminho Hamiltoniano em G?

   Prove que o Problema do Caminho Hamiltoniano é NP-completo.

5. [3.5 valores] Considere um labirinto constituído por salas ligadas por corredores (de sentido único). No início, o jogador tem pontuação zero (i.e., tem zero pontos). O percurso começa na sala assinalada com ENTRADA e o objetivo é chegar à sala assinalada com SAÍDA com a maior pontuação possível.

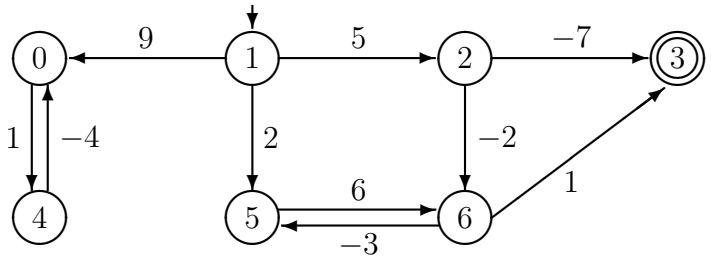
   A pontuação do jogador é alterada quando ele atravessa um corredor. Cada corredor tem um número pintado na parede (chamado o número de pontos do corredor). A pontuação à saída do corredor é a soma desse número com a pontuação à entrada do corredor. O número de pontos de um corredor pode ser positivo ou negativo.

   Algumas salas são **ratoeiras**: são salas a partir das quais não é possível chegar à saída. Mas todas as salas são acessíveis a partir da entrada.

   O jogo termina logo que o jogador chega à saída ou a uma ratoeira. Se o jogador chegar à saída, é considerado: *vencedor*, se a sua pontuação for positiva; *desenrascado*, se a sua pontuação for zero; *perdedor*, se a sua pontuação for negativa. Se o jogador chegar a uma ratoeira, é imediatamente expulso do labirinto (ignorando-se a sua pontuação atual).

   Pretende-se saber se é possível haver perdedores (jogadores que, com azar ou por nabice, chegam à saída com uma pontuação negativa) com base na descrição do labirinto.

   No labirinto desenhado na figura, há 7 salas e 10 corredores, 1 é a sala de entrada e 3 é a sala de saída. Repare que as salas 0 e 4 são ratoeiras. Neste caso, é possível haver perdedores, porque a pontuação final de um jogador que atravesse os corredores $(1, 2)$ e $(2, 3)$ é -2.



   Note que, se o número de pontos do corredor $(2, 3)$ fosse -5, não seria possível haver perdedores.

   Apresente uma função booleana (em pseudo-código) que recebe a descrição do labirinto através das 4 seguintes variáveis:

   - *nRooms*, que é o número de salas.
     (As salas são automaticamente identificadas pelos números $0, 1, \ldots, nRooms - 1$.)
   - *corridors*, que é a sequência de corredores, por uma ordem arbitrária.
     Cada corredor é um terno (r_1, r_2, p), que indica que o corredor parte da sala r_1, chega à sala r_2 e o seu número de pontos é p.
   - *entrance*, que é a sala de entrada.
   - *exit*, que é a sala de saída.

   A função deve retornar *true* se, e só se, puder haver perdedores no labirinto.

   O corpo da sua função deve respeitar as regras enunciadas na próxima página.

O corpo da sua função deve:

1. Chamar uma função que constrói um grafo.
   Não programe esta função mas, para ilustrar o que ela faz, desenhe o grafo que seria construído com o labirinto do exemplo (ou indique que é o grafo do enunciado, se não quiser fazer alterações).
2. Chamar um ou vários algoritmos de grafos estudados, **como se eles estivessem numa biblioteca**, mesmo que esses algoritmos sejam menos eficientes do que poderiam ser para este caso. Há apenas uma exceção: se precisar de aceder a informação que foi computada e não é retornada, pode assumir que o algoritmo a retorna.
3. Se entre duas chamadas a algoritmos de grafos estudados quiser fazer alterações ao grafo, desenhe apenas o novo grafo para o labirinto do exemplo, mas não programe as alterações.
4. Calcular e retornar o resultado (que é um booleano).

6. [3 valores] Apesar do Rui ainda ser uma criança, já revela uma forte aptidão para o *design*. Ultimamente, tem-se divertido a criar *composições* com duas sequências de números, X e Y, respeitando as seguintes regras:

   - Os elementos da sequência X são colocados na primeira linha, preservando a ordem em que ocorrem em X.
   - Os elementos da sequência Y são colocados na segunda linha, também pela ordem em que ocorrem em Y.
   - Sempre que há dois números numa mesma coluna (um elemento de X em cima e um elemento de Y em baixo), o número de baixo é maior ou igual ao de cima.

   Veja as 6 composições possíveis com as sequências (5 7 3) e (5):

   ```
   5 7 3     5 7 3     5 7 3       5 7   3     5   7 3       5 7 3
   5           5         5           5       5       5
   ```

   Como há 19 composições possíveis com as sequências (5 7 3) e (5 5), mostram-se apenas três:

   ```
   5 7 3         5 7 3         5 7 3
   5   5       5       5         5 5
   ```

   O Rui escolhe as duas sequências e depois cria composições com elas. Ele diz que "se as sequências não forem muito pequeninas, há imensas composições possíveis". Pode ajudar o Rui a descobrir quantas são?

   Apresente **uma função matemática recursiva** que, com base em duas sequências não vazias de números inteiros positivos,

   $$X = (x_1\ x_2\ \cdots\ x_m) \quad \text{e} \quad Y = (y_1\ y_2\ \cdots\ y_n) \qquad (\text{com } m, n \geq 1),$$

   calcula o número de composições possíveis com X e Y. Indique claramente o que representa cada uma das variáveis que utilizar e explicite a chamada inicial (a chamada que resolve o problema).