

Exame de Análise e Desenho de Algoritmos

Departamento de Informática, FCT NOVA

2 de Julho de 2018

Duração: 3 horas.

O exame tem **6** páginas e **6** perguntas.

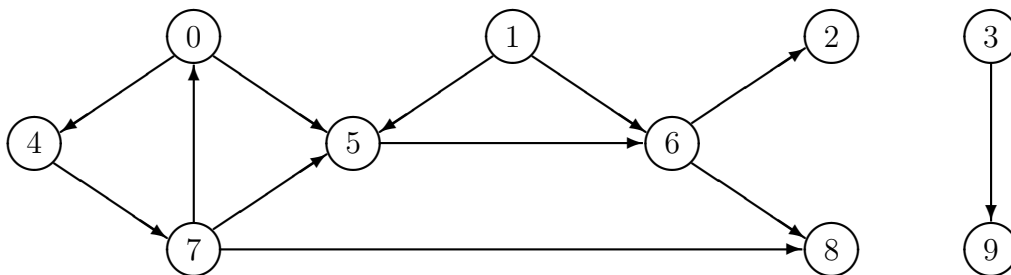
Responda a **perguntas** diferentes em **folhas** diferentes.

Se precisar de folhas, peça ao docente.

Nº de **folhas** entregues
(excluindo o enunciado)

Número: _____ Nome: _____

1. [3 valores] Suponha que se executa a **versão recursiva** do algoritmo *dfsTraversal* (que percorre os vértices de um grafo em profundidade) com o grafo G esquematizado na figura.



Assuma que os métodos *nodes* e *outAdjacentNodes* iteram sempre os vértices por ordem crescente. Por exemplo, $G.outAdjacentNodes(0)$ produz os vértices 4 e 5 (por esta ordem).

- (a) Indique a ordem pela qual os vértices são processados. Um vértice v é processado quando é executada a atribuição `processed[v] = true`.

- (b) Qual é altura da maior árvore encontrada? _____

2. [3.5 valores] Considere a seguinte função recursiva, $f_M(j)$, onde:

- M é uma matriz com 2 linhas e $n \geq 2$ colunas de números inteiros;
- j é um inteiro entre 0 e n .

$$f_M(j) = \begin{cases} 0, & \text{se } j = 0; \\ \max(M[0][0], M[1][0]), & \text{se } j = 1; \\ \max\left(f_M(j-1), \max(M[0][j-1], M[1][j-1]) + f_M(j-2)\right), & \text{se } j \geq 2. \end{cases}$$

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

Apresente um algoritmo iterativo, desenhado segundo a técnica da programação dinâmica e implementado em Java, que recebe uma matriz com 2 linhas e $n \geq 2$ colunas de números inteiros e calcula o valor de $f_M(n)$. Estude (justificando) as complexidades temporal e espacial do seu algoritmo.

3. [3.5 valores] A classe *QueueMinIn2Stacks* implementa filas de inteiros com disciplina FIFO, que oferecem uma operação para obter o menor elemento da fila, recorrendo a duas pilhas. Esta classe usa a interface *Stack* e a classe *StackInLinkedList*, que implementa uma pilha com uma lista ligada.

Considere a função $\Phi(Q)$, que atribui a cada objeto Q da classe *QueueMinIn2Stacks* o número de elementos guardados na pilha $Q.\text{input}$:

$$\Phi(Q) = Q.\text{input}.\text{size()}.$$

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

```
public class QueueMinIn2Stacks {  
  
    // The most recent elements are stored in stack input.  
    private Stack<Integer> input;  
  
    // The oldest elements are stored in stack output  
    // at the first position of the pair.  
    private Stack<Pair<Integer, Integer>> output;  
  
    // Minimum element in stack input.  
    private int minInput;
```

```

public QueueMinIn2Stacks( ) {
    input = new StackInLinkedList<>();
    output = new StackInLinkedList<>();
    minInput = Integer.MAX_VALUE;
}

public boolean isEmpty( ) {
    return input.isEmpty() && output.isEmpty();
}

public int minimum( ) {
    if ( this.isEmpty() )
        throw new EmptyQueueException();

    if ( !output.isEmpty() ) {
        int minOutput = output.top().getSecond();
        if ( minOutput < minInput )
            return minOutput;
    }
    return minInput;
}

public void enqueue( int element ) {
    input.push(element);
    if ( element < minInput )
        minInput = element;
}

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

    if ( output.isEmpty() )
        this.transfer();
    return output.pop().getFirst();
}

private void transfer( ) {
    int min = Integer.MAX_VALUE;
    do {
        int element = input.pop();
        if ( element < min )
            min = element;
        output.push( new Pair<>(element , min) );
    } while ( !input.isEmpty() );
    minInput = Integer.MAX_VALUE;
}
}

```

4. [3.5 valores] O **Problema da Árvore de Cobertura com Grau Limitado** formula-se da seguinte forma. Dados:

- um grafo $G = (V, A)$ não orientado (e não pesado) e
- um inteiro positivo L ,

existe um subconjunto de arcos $A' \subseteq A$ que verifica as três seguintes propriedades?

- (1) $|A'| = |V| - 1$ (A' tem exatamente $|V| - 1$ arcos).
- (2) O grafo $G' = (V, A')$ é acíclico.
- (3) Para todo o vértice $v \in V$, o número de arcos de A' incidentes em v não excede L (i.e., o maior grau dos vértices em G' é inferior ou igual a L).

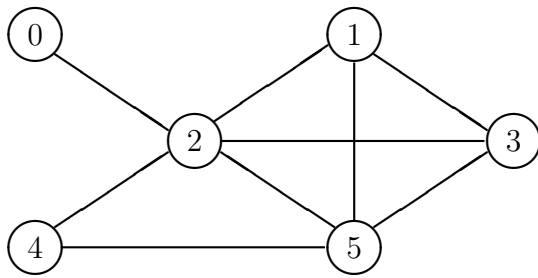


Figura 1: Grafo $G_e = (V_e, A_e)$

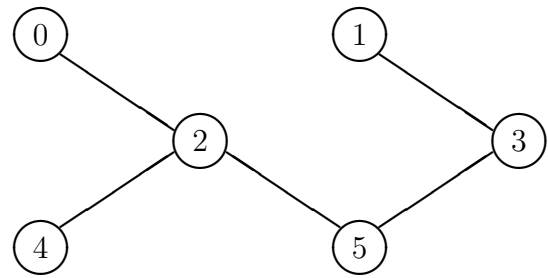


Figura 2: Grafo $G'_e = (V_e, A'_e)$

Considere a instância $(G_e, 3)$, onde $G_e = (V_e, A_e)$ é o grafo esquematizado na Figura 1. Por exemplo, o subconjunto de arcos $A'_e = \{(0, 2), (1, 3), (2, 4), (2, 5), (3, 5)\}$ verifica as três propriedades pretendidas:

- (1) A'_e tem 5 arcos.
- (2) O grafo $G'_e = (V_e, A'_e)$ é acíclico, como se pode verificar facilmente na Figura 2.
- (3) O maior grau dos vértices em G'_e não excede 3: os vértices 0, 1 e 4 têm grau 1; os vértices 3 e 5 têm grau 2; o vértice 2 tem grau 3.

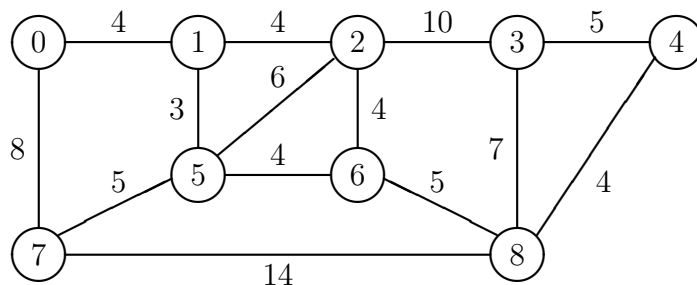
Prove que o Problema da Árvore de Cobertura com Grau Limitado é NP-completo. Pode assumir que tem à sua disposição uma função booleana, $isAcyclic((V, A))$, que recebe um grafo não orientado (V, A) , retorna *true* se, e só se, o grafo for acíclico e cuja complexidade temporal é $O(|V| + |A|)$.

5. [3.5 valores] Há muitos, muitos anos, quando as pessoas se deslocavam a cavalo, era frequente as viagens demorarem vários dias. Mas havia hospedarias onde podiam pernoitar. Alguns viajantes não se importavam de fazer longos percursos em cada dia, se isso permitisse reduzir o número de dias em viagem. Mas outros, mais frágeis fisicamente, preferiam **percursos diários curtos**, mesmo que demorassem muitos dias a chegar ao destino.

Considere que os locais (localidades ou simples hospedarias) e as estradas (que podiam ser percorridas em ambos os sentidos) são modelados por um grafo não orientado, conexo e pesado. O peso (positivo) de um arco entre os locais x e y , denotado por $t(x, y)$, é o tempo médio (em horas) para percorrer a estrada entre x e y . Para um viajante frágil, qualquer caminho com $n \geq 1$ arcos, $v_0, v_1, v_2, \dots, v_n$, seria percorrido em n dias. No primeiro dia faria uma viagem de $t(v_0, v_1)$ horas, no segundo dia faria uma viagem de $t(v_1, v_2)$ horas, e assim sucessivamente, até ao último dia em que faria uma viagem de $t(v_{n-1}, v_n)$ horas. Para esses viajantes, o **caminho** $v_0, v_1, v_2, \dots, v_n$ **é perfeito** se o peso máximo dos arcos do caminho,

$$\max_{0 \leq i < n} t(v_i, v_{i+1}),$$

é o menor possível, considerando todos os caminhos de v_0 para v_n .



Para exemplificar, considere o grafo esquematizado na figura, que tem 9 locais e 14 estradas.

- O caminho 2 1 5 seria feito em dois dias: 4 horas de viagem no primeiro dia e 3 horas no segundo. Como não há caminhos do local 2 para o local 5 em que o peso máximo dos arcos desses caminhos seja inferior a 4, o caminho 2 1 5 é perfeito. Repare que o caminho 2 6 5 também é perfeito, porque os dois arcos têm peso 4.
- Os caminhos 0 1 2 6 8 e 0 1 5 6 8 também são perfeitos. Em ambos, o peso máximo dos arcos é 5.
- Do local 0 para o local 7, o caminho 0 1 5 7 é perfeito (o peso máximo dos arcos é 5).

Considerando todos os caminhos perfeitos entre dois locais distintos, qual é o peso máximo dos arcos envolvidos nesses caminhos? Esse valor é o que se pretende calcular. No caso do grafo do exemplo, o peso máximo dos arcos dos caminhos perfeitos é 5.

Apresente uma função (em pseudo-código) que recebe um grafo G não orientado, conexo e pesado (onde os pesos são números positivos inferiores a 24). A função deve retornar o peso máximo dos arcos dos caminhos perfeitos. **O corpo da sua função deve chamar** um ou vários algoritmos de grafos estudados, como se eles estivessem numa biblioteca (mesmo que esses algoritmos retornem resultados que não interessam para resolver este problema e sejam menos eficientes do que poderiam ser para este caso).

6. [3 valores] MATRIXINT é uma paciência jogada com uma tabela T de números inteiros não negativos, que tem duas linhas e um número positivo de colunas. O objetivo é selecionar uma sequência estritamente crescente de colunas, $c_1, c_2, \dots, c_k$, de forma a maximizar a soma dos números que se encontram na primeira linha e nessas colunas da tabela, ou seja, nas posições $T[1][c_1], T[1][c_2], \dots, T[1][c_k]$. O problema é que, ao selecionar uma coluna j da tabela, é-se impedido de selecionar as $T[2][j]$ colunas à direita de j .

	1	2	3	4	5	6
1	10	20	50	40	30	10
2	2	0	2	0	2	2

Vejamos um exemplo com a tabela acima. Na primeira jogada, não há impedimentos: pode-se selecionar qualquer uma das 6 colunas.

- Se, na primeira jogada, se selecionar a coluna **1**, ganham-se 10 pontos, porque $T[1][1] = 10$. Mas já não é possível selecionar as colunas **2** e **3**, porque se está impedido de selecionar as $T[2][1] = 2$ colunas imediatamente à direita da coluna **1**.
- Se, na primeira jogada, se selecionar a coluna **2**, ganham-se 20 pontos, porque $T[1][2] = 20$. Neste caso, como $T[2][2] = 0$, na segunda jogada pode-se selecionar a coluna **3**, ou a **4**, ou a **5** ou a **6**.

Com esta tabela, a sequência ótima de jogadas (que maximiza o número total de pontos ganhos) é a que seleciona as colunas **2**, **4** e **5**, que permite obter 90 pontos.

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

numa tabela T de números inteiros não negativos (com 2 linhas e $n \geq 1$ colunas),

calcula a maior pontuação que é possível obter no jogo com a tabela T . Indique claramente o que representa cada uma das variáveis que utilizar e explicita a chamada inicial (a chamada que resolve o problema).