# Exame de Análise e Desenho de Algoritmos

Departamento de Informática, FCT NOVA
20 de Junho de 2020
Duração: 3 horas

Caderno 1 (3 folhas e 3 perguntas)

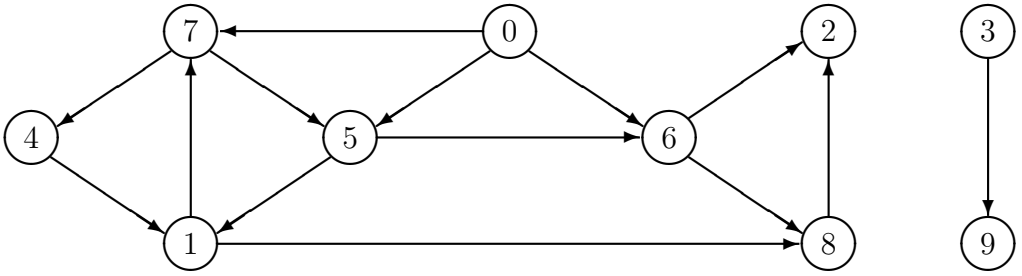
**Tem de entregar os 2 cadernos.**

**Os cadernos não podem ser desagrafados.**

**Identifique os cadernos com o seu número e o seu nome.**

**Número:** _____ **Nome:** _____

**Pergunta 1** Suponha que se executa a **versão iterativa** 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 5, 6 e 7 (por esta ordem).

**Atenção:** A implementação do método *dfsExplore* é **iterativa** (não há chamadas recursivas).

(a) [2 valores] Indique a ordem pela qual os vértices são processados. Um vértice $v$ é processado quando é executada a atribuição `processed[`$v$`] = true`.

0 7 5 6 8 2 1 4 3 9

(b) [0.5 valores] Quantas árvores são geradas (quantas vezes o método *dfsExplore* é executado)?

2

(c) [0.5 valores] Quantos níveis tem a maior árvore gerada? Note que a raiz está no primeiro nível, os filhos da raiz estão no segundo nível, os netos da raiz estão no terceiro nível, etc.

6

**Pergunta 2** Considere a seguinte função recursiva, $f_{S,k}(i,j)$, onde:

- $S = (x_0, x_1, \ldots, x_{n-1})$ é uma sequência de números inteiros (com $n \geq 1$);
- $k$ é um número inteiro;
- $i$ e $j$ são dois números inteiros entre 0 e $n-1$ $(0 \leq i < n \text{ e } 0 \leq j < n)$.

$$f_{S,k}(i,j) = \begin{cases} x_j, & \text{se } i = 0 \text{ e } 0 \leq j < n; \\ x_i, & \text{se } j = 0 \text{ e } 1 \leq i < n; \\ f_{S,k}(i-1, j-1), & \text{se } i \geq 1 \text{ e } j \geq 1 \text{ e } f_{S,k}(i, j-1) = f_{S,k}(i-1, j); \\ k + f_{S,k}(i-1, j-1), & \text{se } i \geq 1 \text{ e } j \geq 1 \text{ e } f_{S,k}(i, j-1) \neq f_{S,k}(i-1, j). \end{cases}$$

Note que a sequência $S$ e o número $k$ não variam entre chamadas recursivas. Por esse motivo, optou-se por escrever $f_{S,k}(i,j)$ em vez de $f(S,k,i,j)$.

(a) [2.5 valores] Apresente um algoritmo iterativo, desenhado segundo a técnica da programação dinâmica e implementado em Java, que receba uma sequência de $n$ números inteiros (guardada num vetor do tipo `int[]` com comprimento $n$) e um número inteiro:

$$S = (x_0, x_1, \ldots, x_{n-1}) \text{ e } k \qquad (\text{com } n \geq 1)$$

e calcule o valor de $f_{S,k}(n-1, n-1)$.

```
int funF( int[] x, int k ) {
   int[][] m = new int[x.length][x.length];
   for ( int j = 0; j < x.length; j++ )
      m[0][j] = x[j];
   for ( int i = 1; i < x.length; i++ )
      m[i][0] = x[i];
   for ( int i = 1; i < x.length; i++ )
      for ( int j = 1; j < x.length; j++ )
         if ( m[i][j-1] == m[i-1][j] )
            m[i][j] = m[i-1][j-1];
         else
            m[i][j] = k + m[i-1][j-1];
   return m[x.length-1][x.length-1];
}
```

**Nota:** Embora a solução apresentada não tenha a menor complexidade espacial possível, em exame, é considerada totalmente correta.

(b) [0.5 valores] Qual é a complexidade espacial do seu algoritmo? Justifique a sua resposta.

A complexidade espacial é $\Theta(n^2)$, onde $n$ é o comprimento do vetor de entrada, porque a matriz `m` tem $n$ linhas, cada uma com $n$ inteiros.

(c) [0.5 valores] Qual é a complexidade temporal do seu algoritmo? Justifique a sua resposta.

A complexidade temporal é $\Theta(n^2)$, onde $n$ é o comprimento do vetor de entrada, porque:

- o primeiro ciclo *for* tem $\Theta(n)$ passos
- o segundo ciclo *for* tem $\Theta(n)$ passos
- o terceiro e o quarto ciclos *for*, no seu conjunto, têm $\Theta(n^2)$ passos

e cada um dos passos referidos acima é constante.

**Pergunta 3** Neste exercício pretende-se implementar o algoritmo de Dijkstra em Java, usando a classe *PriorityQueue*, disponível na biblioteca. Uma *PriorityQueue* é uma fila com prioridade implementada em *heap* binário (que não oferece a operação *decreaseKey*).

(a) [1.5 valores] Implemente o método *dijkstra*, completando os corpos dos dois ciclos assinalados na página seguinte. Considere que, para além dos métodos *isEmpty* e *size*, só há mais três métodos oferecidos por uma *PriorityQueue* (de elementos do tipo E):

- **boolean** *add*( E e ); // Inserts the specified element in this queue.
- E *remove*( ) **throws** NoSuchElementException;
  // Retrieves and removes the least element from this queue (ties are broken arbitrarily).
- **boolean** *remove*( Object o );
  // Removes a single instance of the specified element from this queue, if it is present.

```
class Edge {
    private int destin, weight;  // Destination node and weight

    public Edge( int destination, int theWeight ) {
        destin = destination;
        weight = theWeight;
    }

    public int getDestin( ) { return destin; }

    public int getWeight( ) { return weight; }
}

class Entry implements Comparable<Entry> {
    private int key, val;

    public Entry( int theKey, int value ) {
        key = theKey;
        val = value;
    }

    public int getKey( ) { return key; }

    public int getValue( ) { return val; }

    public void setKey( int newKey ) { key = newKey; }

    public void setValue( int newValue ) { val = newValue; }

    public boolean equals( Object object ) {
        if ( object instanceof Entry ) {
            Entry entry = (Entry) object;
            return key == entry.key && val == entry.val;
        }
        return false;
    }

    public int compareTo( Entry entry ) {
        int compResult = key - entry.key;
        if ( compResult != 0 )
            return compResult;
        return val - entry.val;
    }
}
```

```
import java.util.List;      import java.util.LinkedList;
import java.util.Queue;     import java.util.PriorityQueue;

public class Problem {
    private int numNodes;         // Number of nodes
    private List<Edge>[] edges; // The adjacency linked lists

    @SuppressWarnings("unchecked")
    public Problem( int numberOfNodes ) {
        numNodes = numberOfNodes;
        edges = new List[numNodes];
        for ( int i = 0; i < numNodes; i++ )
            edges[i] = new LinkedList<>();
    }

    public void addEdge( int tail, int head, int dist ) {
        edges[tail].add( new Edge(head, dist) );
    }

    public int[] dijkstra( int origin ) {
        boolean[] selected = new boolean[numNodes];
        int[] length = new int[numNodes];
        Queue<Entry> connected = new PriorityQueue<>(); // Capacidade omitida
        for ( int i = 0; i < numNodes; i++ )             // propositadamente.
            length[i] = Integer.MAX_VALUE;
        length[origin] = 0;
        connected.add( new Entry(0, origin) );
        do {
                int node = connected.remove().getValue();
                // Complete o corpo deste ciclo

                if ( !selected[node] ) {
                    selected[node] = true;
                    this.exploreNode(node, selected, length, connected);
                }

            }
        while ( !connected.isEmpty() );
        return length;
    }

    private void exploreNode( int source, boolean[] selected, int[] length,
                              Queue<Entry> connected ) {
        for ( Edge e : edges[source] ) {
            int node = e.getDestin();
            // Complete o corpo deste ciclo

            if ( !selected[node] ) {
                int newLength = length[source] + e.getWeight();
                if ( newLength < length[node] ) {
                    length[node] = newLength;
                    connected.add( new Entry(newLength, node) );
                }
            }



        }
    }

}
```

(b) [1 valor] Qual é a complexidade temporal do seu método *dijkstra*? Justifique a sua resposta, denotando por $(V, A)$ o grafo guardado nas variáveis *numNodes* e *edges*.

Em termos de complexidades, as diferenças em relação ao algoritmo estudado advêm do facto do número de elementos na fila poder ser da ordem do número de arcos. Cada vértice continua a ser explorado, no máximo, uma vez.

| | |
|---|---|
| criação de 2 vetores | $\Theta(1)$ |
| criação da fila com prioridade | $\Theta(1)$ |
| inicialização do vetor length | $\Theta(\|V\|)$ |
| inserção da origem na fila | $\Theta(1)$ |
| $\leq \|A\|$ remoção do mínimo da fila | $O(\|A\| \times \log \|A\|)$ |
| $\leq \|V\|$ obtenção dos arcos incidentes | $O(\|A\|)$ |
| $\leq \|A\|$ inserção de uma entrada na fila | $O(\|A\| \times \log \|A\|)$ |
| **Total** | $O(\|V\| + \|A\| \times \log \|A\|)$ |

Como $|A| < |V|^2$, a complexidade do algoritmo é $O(|V| + |A| \times \log |V|)$.

**Nota para facilitar a compreensão:** Quando se descobre um caminho mais curto para um vértice, em vez de se retirar a entrada antiga e inserir a entrada nova (que é equivalente a baixar a chave da entrada, em termos de funcionalidade), só se insere a nova entrada. Consequentemente, a fila pode ter várias entradas com o mesmo vértice (no máximo, tantas quanto o grau de entrada do vértice), mas a primeira entrada a ser removida é a que tem a menor chave. Essa chave é a que a fila teria se se tivessem retirado as entradas antigas. Nessa altura, o vértice é explorado, mas só é explorado uma vez, porque se testa se o vértice já foi selecionado sempre que se remove uma entrada da fila.

(c) [1 valor] Qual é a complexidade espacial do seu método *dijkstra*? Justifique a sua resposta, denotando por $(V, A)$ o grafo guardado nas variáveis *numNodes* e *edges*.

A complexidade espacial é $O(|V| + |A|)$ porque:

- o vetor *selected* tem $\Theta(|V|)$ booleanos,
- o vetor *length* tem $\Theta(|V|)$ inteiros,
- a fila *connected* tem $O(|A|)$ entradas (e cada entrada tem dois inteiros).

# Exame de Análise e Desenho de Algoritmos

Departamento de Informática, FCT NOVA
20 de Junho de 2020
Duração: 3 horas

Caderno 2 (4 folhas e 3 perguntas)

**Tem de entregar os 2 cadernos.**

**Os cadernos não podem ser desagrafados.**

**Identifique os cadernos com o seu número e o seu nome.**

**Número:** _____ **Nome:** _____

**Pergunta 4** A classe *SlackStack* implementa filas com disciplina LIFO, de elementos do tipo E, com uma pilha implementada em vetor. Essa pilha (chamada `lifo`) tem o dobro da capacidade indicada no construtor da *SlackStack*. Quando se empilha um elemento numa *SlackStack* e a sua `lifo` está cheia, remove-se metade dos elementos da `lifo` (a metade mais antiga), fazendo uso da pilha `aux`.

```
import dataStructures.Stack;
import dataStructures.StackInArray;

public class SlackStack<E> {

    private Stack<E> lifo;
    private Stack<E> aux;
    private int trueCapacity;

    public SlackStack( int capacity ) {
        trueCapacity = 2 * capacity;
        lifo = new StackInArray<>(trueCapacity);
        aux = new StackInArray<>(capacity);
    }

    public void push( E elem ) {
        if ( lifo.size() == trueCapacity )
            this.deleteOldHalf();
        lifo.push(elem);
    }

    private void deleteOldHalf( ) {
        int half = trueCapacity / 2;
        for ( int i = 0; i < half; i++ )
            aux.push( lifo.pop() );
        for ( int i = 0; i < half; i++ )
            lifo.pop();
        for ( int i = 0; i < half; i++ )
            lifo.push( aux.pop() );
    }

    public E pop( ) throws RuntimeException {
        if ( lifo.size() == 0 )
            throw new RuntimeException("The stack is empty");

        return lifo.pop();
    }

}
```

Considere a função $\Phi(S)$, que atribui a cada objeto $S$ da classe *SlackStack* o dobro do número de elementos guardados na pilha $S$.`lifo`:

$$\Phi(S) = 2 \times S\texttt{.lifo.size()}\,.$$

(a) [0.6 valores] Prove que $\Phi$ é uma função potencial válida.

$\Phi(S_0) = 0$, porque a pilha `lifo` tem 0 elementos quando $S$ é criada (e $2 \times 0 = 0$).

$\Phi(S) \geq 0$, porque o número de elementos na pilha `lifo` nunca é negativo.

(b) [2.9 valores] Calcule as complexidades amortizadas dos métodos *push* e *pop* da classe *SlackStack*, justificando. Assuma que os métodos *size*, *push* e *pop* da classe *StackInArray* têm complexidade constante. No estudo da complexidade amortizada do método *push*, analise separadamente os casos em que a condição do **if** é: falsa; verdadeira. No estudo da complexidade amortizada do método *pop*, assuma que não é levantada a exceção.

Recorde que não se usam constantes multiplicativas nos custos reais. Por exemplo, se um custo real $c$ for de ordem $2n$, onde $n$ é uma variável que denota uma quantidade positiva, $c = n$.

| Operação | | $c$ | $\Phi(S') - \Phi(S)$ | $\widehat{c} = c + \Delta\Phi$ | |
|---|---|---|---|---|---|
| push | if F | 1 | $2s' - 2s = 2(s+1) - 2s = 2$ | $1 + 2 = 3$ | $O(1)$ |
| | if T | $s$ | $2s' - 2s = 2(\frac{s}{2} + 1) - 2s = 2 - s$ | $s + (2 - s) = 2$ | |
| pop | | 1 | $2s' - 2s = 2(s-1) - 2s = -2$ | $1 + (-2) = -1$ | $O(1)$ |

**Notas para facilitar a compreensão:**

- O potencial da estrutura é o dobro do número de elementos na pilha `lifo`.
- $s$ denota o número de elementos na pilha `lifo` antes da operação.
- $s'$ denota o número de elementos na pilha `lifo` depois da operação.
- O custo real do método *push* quando a condição é verdadeira é $s$ (e não $\frac{3}{2}s$) porque não se usam constantes multiplicativas nos custos reais.

**Pergunta 5** A paciência MinReverses joga-se invertendo arcos num grafo, tendo como objetivo "construir" um caminho entre dois vértices com o menor número possível de inversões.

No início do jogo, o jogador recebe um grafo e dois vértices distintos, $x$ e $y$. O grafo é orientado, não pesado e fracamente conexo (há caminho entre dois quaisquer vértices, se se ignorar o sentido dos arcos). Em cada jogada, o jogador altera o grafo, invertendo o sentido de um dos arcos. Em qualquer momento, o jogador pode terminar o jogo, assinalando que não quer fazer mais alterações. Considera-se que o jogo foi ganho se há algum caminho de $x$ para $y$ no último grafo obtido e o número de jogadas foi o menor possível. A ordem das jogadas é irrelevante.

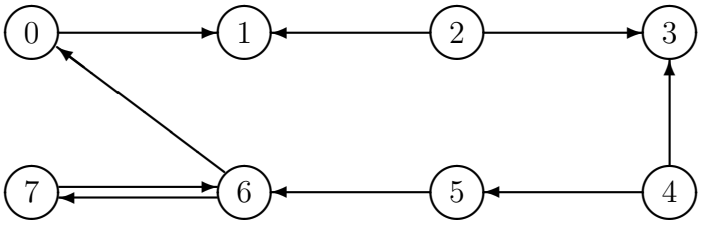



**Figura 1:** Grafo $G_e = (V_e, A_e)$

Vejamos alguns exemplos, todos com o grafo $G_e$ desenhado na Figura 1.

- **Com $x = 1$ e $y = 4$:** ganha-se invertendo os arcos $(2, 1)$ e $(4, 3)$.
  O caminho é 1 2 3 4. O número mínimo de jogadas é **2**.

  Repare que um jogador distraído poderia construir o caminho 1 0 6 5 4, invertendo os arcos $(0, 1)$, $(6, 0)$, $(5, 6)$ e $(4, 5)$. Mas perderia o jogo porque teria invertido **4** arcos (e há uma solução com menos inversões).

- **Com $x = 2$ e $y = 7$:** ganha-se invertendo o arco $(4, 3)$.
  O caminho é 2 3 4 5 6 7. O número mínimo de jogadas é **1**.

- **Com $x = 4$ e $y = 1$:** ganha-se não invertendo arcos.
  O caminho é 4 5 6 0 1. O número mínimo de jogadas é **0**.

- **Com $x = 7$ e $y = 6$:** ganha-se não invertendo arcos.
  O caminho é 7 6. O número mínimo de jogadas é **0**.

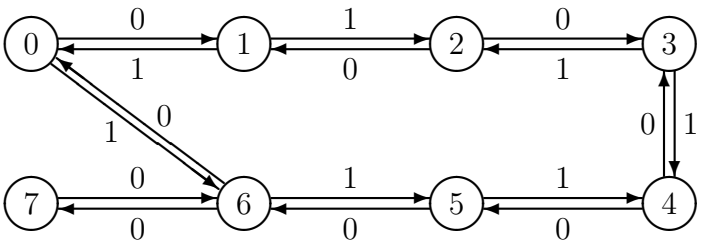
Neste problema, pretende-se que implemente uma função, *minReverses*($G$, $x$, $y$), que recebe:

- um grafo $G$, orientado, não pesado e fracamente conexo, e
- dois vértices distintos, $x$ e $y$.

A função deve retornar o número mínimo de jogadas para haver um caminho de $x$ para $y$ no último grafo obtido. **O corpo da função *minReverses* deve chamar:**

- uma ou várias funções que constroem grafos;
- 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).

(a) [1 valor] **Desenhe o(s) grafo(s)** que seria(m) construído(s) durante a execução de *minReverses*($G_e$, 1, 4), onde $G_e$ é o grafo do exemplo. Desenhar um grafo é representá-lo da forma habitual (como na Figura 1).




(b) [2.5 valores] Implemente a função *minReverses* (em pseudo-código). Chame a ou as várias funções que constroem grafos, mas não a(s) implemente (porque já ilustrou o(s) grafo(s) criado(s) na alínea anterior). Não implemente nenhum algoritmo de grafos; recorra aos algoritmos estudados, chamando as respetivas funções, sem as alterar.

```
int minReverses( Digraph graph, Node x, Node y ) {
    Digraph<Integer> g = createGraph(graph);
    Integer[] len = dijkstra(g, x).getFirst();
    return len[y];
}
```

**Pergunta 6** O Sr. Yan, dono de um grande restaurante chinês, quer passar a dar três pauzinhos a cada cliente: para além do par habitual, constituído por dois pauzinhos do mesmo tamanho, haverá um terceiro pauzinho mais comprido. O problema é que, para usar apenas os pauzinhos existentes no restaurante, as regras sobre os tamanhos relativos dos pauzinhos têm de ser relaxadas.

Cada cliente receberá um conjunto $C$ com exatamente três pauzinhos, para seu uso exclusivo. Se os comprimentos desses pauzinhos forem:

$$x,\ y\ \text{ e }\ z\quad (\text{com } x \leq y \leq z),$$

a *imperfeição do conjunto* $C$ é $(y-x)^2$, porque os pauzinhos mais curtos deveriam ser do mesmo tamanho e o pauzinho extra será sempre o mais comprido (ou um dos mais compridos).

Nestes tempos de pandemia, em que o número máximo $k$ de clientes é rígido, o Sr. Yan quer formar $k$ conjuntos de pauzinhos, $C_1, C_2, \ldots, C_k$, tais que a soma das imperfeições desses conjuntos seja a menor possível. Se chamarmos *desequilíbrio* à soma das imperfeições dos conjuntos, o Sr. Yan quer $k$ conjuntos de pauzinhos (disjuntos dois a dois e cada um com três pauzinhos) que minimizem o desequilíbrio.

Suponha que o número máximo de clientes é 2 e que há 8 pauzinhos no restaurante, denotados pelos números $1, 2, \ldots, 8$ e com os comprimentos indicados na tabela seguinte.

| Pauzinho | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Comprimento | 14 | 20 | 35 | 38 | 38 | 39 | 39 | 43 |

Há muitas maneiras de formar 2 conjuntos (disjuntos) de 3 pauzinhos cada, mas nenhuma permite obter um desequilíbrio inferior a 9. Veja os seguintes exemplos (recordando que a imperfeição de um conjunto é sempre calculada com os dois pauzinhos mais curtos):

| Conjuntos de Pauzinhos | Desequilíbrio dos Conjuntos |
|---|---|
| $\{1,2,6\}$ e $\{4,5,8\}$ | $(20-14)^2 + (38-38)^2 = 36$ |
| $\{3,4,8\}$ e $\{5,6,7\}$ | $(38-35)^2 + (39-38)^2 = 10$ |
| $\{3,4,5\}$ e $\{6,7,8\}$ | $(38-35)^2 + (39-39)^2 = 9$ |

Pretende-se definir **uma função matemática recursiva** que, com base:

- no número máximo $k$ de clientes (que é um inteiro positivo) e
- numa sequência de números inteiros positivos, que correspondem aos comprimentos dos pauzinhos existentes no restaurante,

$$S = (c_1\ c_2\ \cdots\ c_n) \qquad (\text{com } n \geq 3k),$$

calcule o menor desequilíbrio possível de $k$ conjuntos de pauzinhos (disjuntos dois a dois e cada um com três pauzinhos). Pode assumir que a sequência $S$ está ordenada, por ordem crescente ou decrescente (podendo haver elementos iguais).

Para o exemplo dado, em que $k = 2$, $S = (14\ \ 20\ \ 35\ \ 38\ \ 38\ \ 39\ \ 39\ \ 43)$ e $n = 8$, o valor da função pretendida é 9.

(a) [2.7 valores] Defina a função pretendida e indique claramente o que representa cada uma das variáveis que utilizar. Se assumir que a sequência $S$ está ordenada, indique se a ordenação é crescente ou decrescente.

Assume-se que a sequência está ordenada por ordem decrescente.

$\mathcal{D}_S(i,j)$ é o menor desequilíbrio possível de $i$ conjuntos de pauzinhos quando a sequência de comprimentos é:

$$c_1\ c_2\ \cdots\ c_j$$

para $i = 1, 2, \ldots, k$ e $j = 3i, 3i+1, \ldots, n$.

$$\mathcal{D}_S(i,j) = \begin{cases} (c_3 - c_2)^2 & \text{se } i = 1 \text{ e } j = 3 \\ \min(\ (c_j - c_{j-1})^2,\ \mathcal{D}_S(i, j-1)\ ) & \text{se } i = 1 \text{ e } j > 3 \\ (c_j - c_{j-1})^2 + \mathcal{D}_S(i-1, j-2) & \text{se } i > 1 \text{ e } j = 3i \\ \min(\ (c_j - c_{j-1})^2 + \mathcal{D}_S(i-1, j-2),\ \mathcal{D}_S(i, j-1)\ ) & \text{se } i > 1 \text{ e } j > 3i \end{cases}$$

**Notas para facilitar a compreensão:**

- Caso $i = 1$ e $j = 3$: O único conjunto tem os 3 pauzinhos existentes.
- Caso $i = 1$ e $j > 3$:
  - Se o conjunto tiver o pauzinho de menor comprimento, o menor desequilíbrio possível é $(c_j - c_{j-1})^2$.
  - Se o conjunto não tiver o pauzinho de menor comprimento, o menor desequilíbrio possível é $\mathcal{D}_S(i, j-1)$.
- Caso $i > 1$ e $j = 3i$: Como todos os pauzinhos existentes pertencem a algum conjunto, o pauzinho de menor comprimento vai "emparelhar com o anterior". Os restantes conjuntos serão formados da melhor maneira possível com os restantes pauzinhos. Como a ordenação é decrescente, o pauzinho que sobrar tem comprimento maior ou igual aos comprimentos dos pauzinhos $j$ e $j-1$.
- Caso $i > 1$ e $j > 3i$:
  - Se algum dos conjuntos tiver o pauzinho de menor comprimento, o menor desequilíbrio possível é obtido "emparelhando-o com o anterior" e formando os restantes conjuntos da melhor maneira possível com os restantes pauzinhos. Como a ordenação é decrescente, os pauzinhos que sobrarem têm comprimento maior ou igual aos comprimentos dos pauzinhos $j$ e $j-1$.
  - Se nenhum conjunto tiver o pauzinho de menor comprimento, o menor desequilíbrio possível é $\mathcal{D}_S(i, j-1)$.

(b) [0.3 valores] Indique a chamada inicial (a chamada que resolve o problema).

$\mathcal{D}_S(k,n)$ ou $\mathcal{D}_S(2,8)$