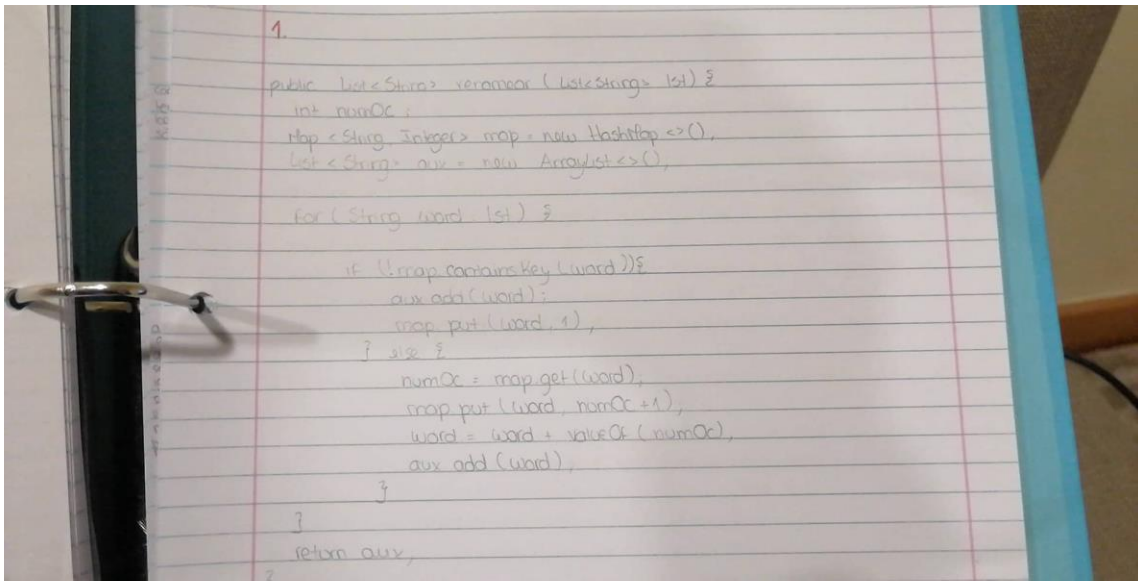
Exames ESINF - Sugestões de correção

Normal 2018



1.

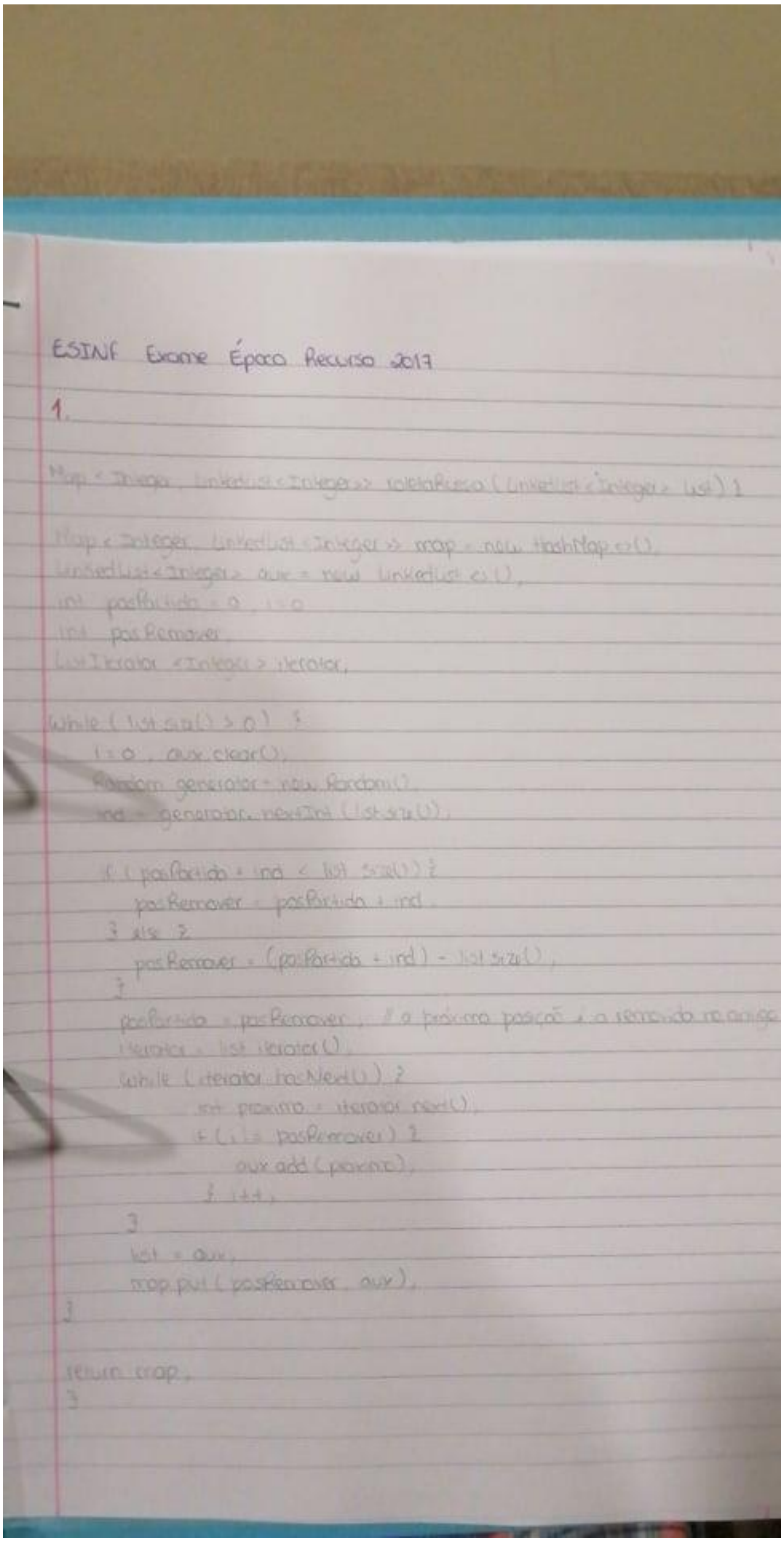
```
public List<String> renomear ( List<String> lst) {
  int numOc;
  Map <String, Integer> map = new HashMap <>(),
  List <String> aux = new ArrayList <>(),

  for ( String word : lst) {

        if (!map.containsKey (word)){
            aux.add(word);
            map.put (word, 1),
        } else {
            numOc = map.get(word);
            map.put (word, numOc +1),
            word = word + valueOf ( numOc),
            aux.add (word),
        }
  }
  return aux,
}
```

b) O método é do tipo determinístico
A sua complexidade temporal é de n.log(n)
O primeiro ciclo é iterado 'n' vezes, consoante o número recebido por parâmetro. Intrinsecamente, tem outro ciclo a ser iterado, logo a complexidade da função é de n. (complexidade do ciclo)
A cada interação, o valor de 'j' do loop interno é multiplicado por 2, logo, o 2 é multiplicado por si próprio K vezes (inicialmente j=2). Assim, $n = 2^K$, logo, a complexidade é de $\log_2(n)$.
Assim, a complexidade do método é de $n \log_2(n)$.

```
3. public Node<E> lca (Node<E> root, Node<E> n1, Node<E> n2){

        if (root == null) { return null; }
        if ( root.getElement() > n1.getElement() &
             root.getElement() < n2.getElement() ou
             root.getElement() < n1.getElement() &
             root.getElement() > n2.getElement() ){
             return root; }

         if (root.getElement() > n1.getElement() &
             root.getElement() > n2.getElement()) {
             return lca(root.getLeft(), n1, n2 )}

         if (root.getElement() < n1.getElement() &
             root.getElement() < n2.getElement() ) {
              return lca (root.getRight(), n1, n2); }

}
```

```
public int ramosEntre2Nos ( Node<E> n1, Node<E> n2, int dist) {

        if ( n1 getElement() == n2 getElement() ) {
            return dist, }

         if ( n1 getElement() > n2 getElement()) {
             return (n1 getLeft(), n2, dist+1), }

         if ( n1 getElement() < n2 getElement()) {
             return (n1 getRight(), n2, dist+1), }

 }

public int DistBetRamos ( Node<E> n1, Node<E> n2) {
        int dist1, dist2,
        Node<E> parenteComum = lca (this root, n1, n2),

         dist1 = ramosEntre2Nos (parenteComum, n1, 0),
         dist2 = ramosEntre 2Nos (parenteComum, n2, 0),

         return (dist1 + dist2),
 }

 5
public <K,V extends Comparable<V>> int ordemServico (HPQ<K,V> heap, V value) {

       int i,
       int size = heap size(),
       for ( i=0  i< size , i++) {

            if ( heap [i, j]. getValue() compareTo (value) == 0 ) {
                return i,
            }
        }
        return -1,
 }
```

Recurso 2017



ESINF Exame Época Recurso 2017

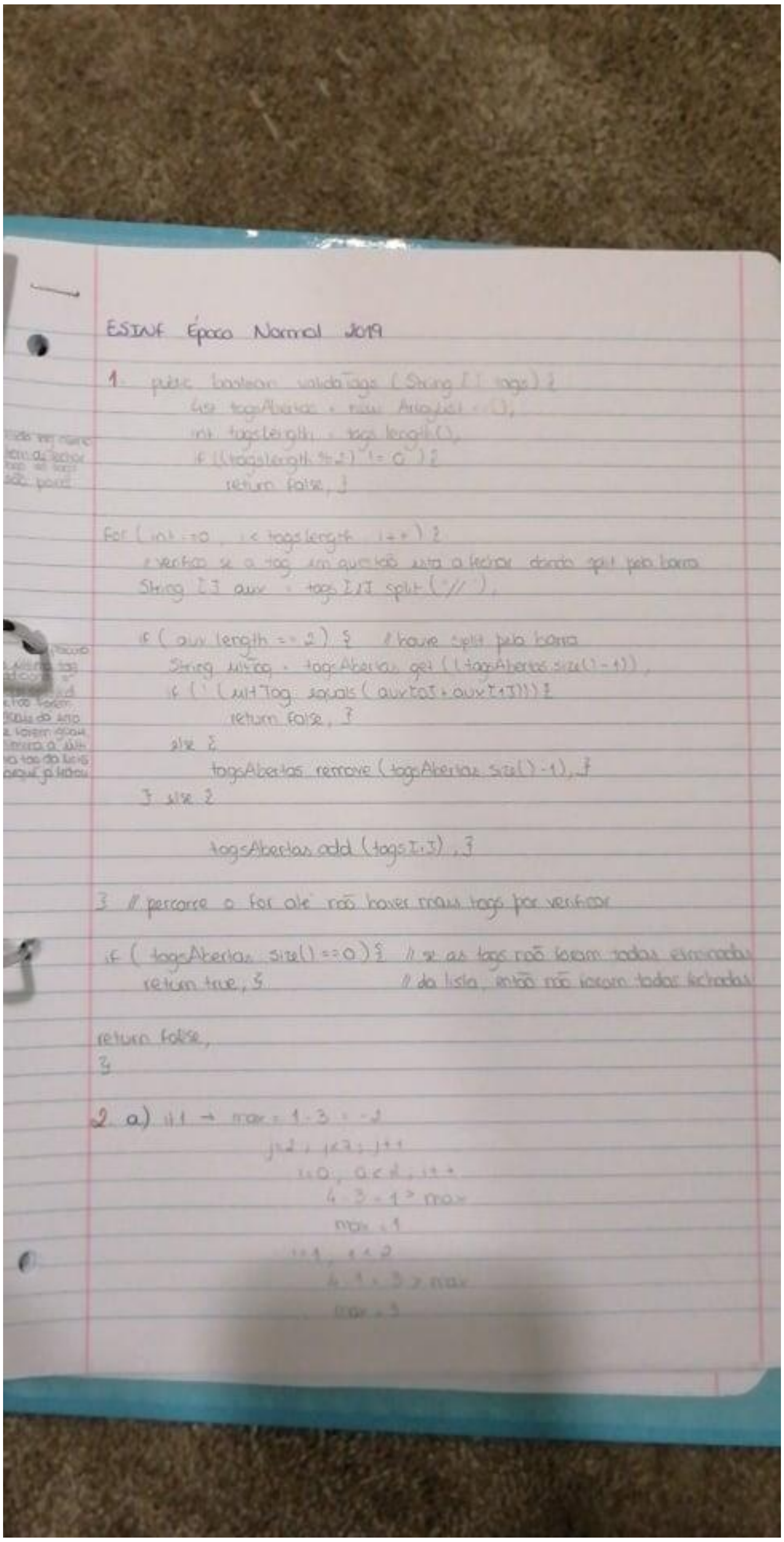
1.

```
Map<Integer, LinkedList<Integer>> roletaRussa(LinkedList<Integer> list) {

Map<Integer, LinkedList<Integer>> map = new HashMap<>();
LinkedList<Integer> aux = new LinkedList<>();
int posPartida = 0, i=0
int posRemover;
ListIterator <Integer> iterator;

While (list.size() > 0) {
    i=0, aux.clear();
    Random generator = new Random();
    ind = generator.nextInt (list.size());

    if ( posPartida + ind < list.size()) {
       posRemover = posPartida + ind;
    } else {
       posRemover = (posPartida + ind) - list.size();
    }
    posPartida = posRemover; // a próxima posição é a removida na antiga
    iterator = list.iterator();
    While (iterator.hasNext()) {
            int proximo = iterator.next();
            if (i != posRemover) {
               aux.add(proximo);
            } i++;
    }
    list = aux;
    map.put(posRemover, aux);
}

return map;
}
```

```
3 public List<E> caminho (Node<E> root, Node<E> dest,
                           List<E> path) {
      if (root == dest) {
          return path;
      }

      if (root.getElement() < dest.getElement()) {
          path.add(root.getRight().getElement());
          return caminho (root.getRight(), dest, path);
      }

      if (root.getElement() > dest.getElement()) {
          path.add( root.getLeft().getElement());
          return caminho (root.getLeft(), dest, path);
      }
  }

  public int caminhoProf ( Node<E> dest) {

      List <E> path = new ArrayList<>();
      path.add(this.root.getElement());

      path = caminho (this.root, dest, path);

      int travessias = path.size() -1;
      int i, prof = 0;

      for (i= travessias ; i>0 ; i--) {
          prof += i;
      }

      return prof;
  }
```

/* a profundidade pode ser calculada com o somatório de todos os elementos inferiores (incluindo o próprio) ao número de nós
e.g. [63, 55, 80, 83] n=4-1=3 prof = 3+2+1=6
[1,2,3,4,5,6] n=6-1=5 prof = 5+4+3+2+1 = 15

Normal 2019



ESINF Época Normal 2019

```
1. public boolean validaTags (String [] tags) {
        List tagsAbertas = new ArrayList<>();
        int tagsLength = tags.length();
        if ((tagsLength % 2) != 0 ) {
            return false; }

for (int i=0 ; i < tagsLength ; i++) {
    // verifica se a tag em questão está a fechar dando split pela barra
    String [] aux = tags[i].split("/");

    if ( aux.length == 2) {   // houve split pela barra
        String ultTag = tagsAbertas.get ((tagsAbertas.size() - 1));
        if (!(ultTag.equals( aux[0] + aux[1]))) {
            return false; }
        else {
            tagsAbertas.remove (tagsAbertas.size() - 1); }
    } else {

            tagsAbertas.add (tags[i]); }

} // percorre o for até não haver mais tags por verificar

if ( tagsAbertas.size() == 0 ) {   // se as tags não foram todas eliminadas
    return true; }                 // da lista, então não foram todas fechadas

return false;
}
```

```
2. a) i=1 → max = 1 - 3 = -2
              j=2 ; j<3 ; j++
                i=0 ; 0<2 ; i++
                  4 - 3 = 1 > max
                  max = 1
                i=1 ; 1<2
                  4 - 1 = 3 > max
                  max = 3
```

[illegible]

```
j=3 , j<7 , j++
    i=0 , i<3 , i++
        1 - 3 = -2 > max  x
    i=1 , i<3 , i++
        1 - 1 = 0 > max  x
    i=2 , i<3 , i++
        1 - 4 = -3 > max  x
```

O método mistery percorre todas as posições do array e, para cada posição, verifica qual a diferença do seu valor e de cada um dos respetivos valores anteriores. O objetivo é, no final, verificar qual a diferença máxima encontrada entre um elemento m e n (em que m está sempre numa posição superior a n).

O valor devolvido é 8, dado que 9-1 é a subtração de maior valor resultante, estando o "9" na posição 5 e o "1" na posição 1

b) A função é não determinística, já que, dependendo do array inserido, poderá ter complexidades distintas.

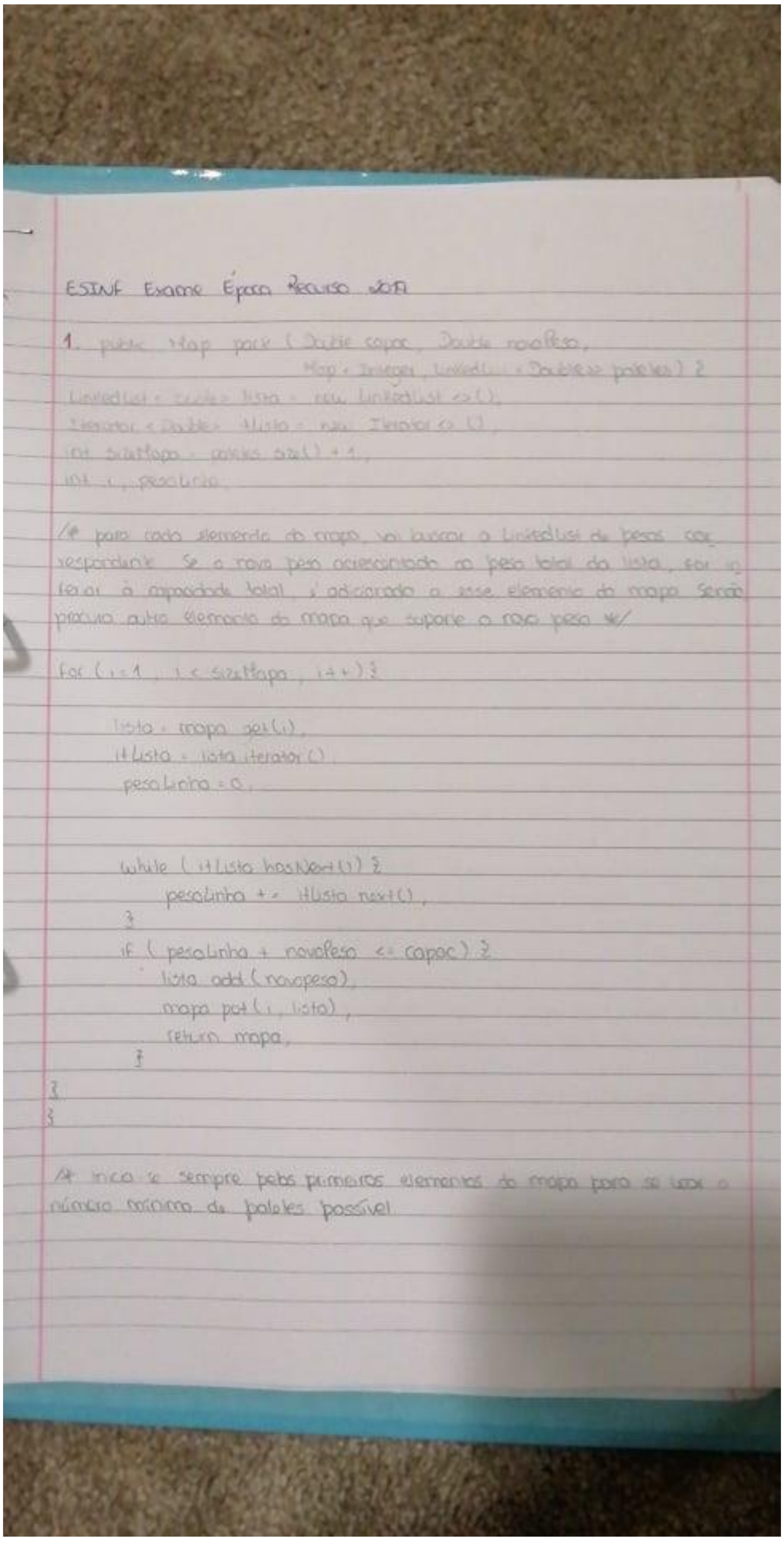
Caso o array "a" só tenha dois elementos, então é realizada a subtração e é retornado o valor (não entra no primeiro ciclo porque, nessa situação, $j == a.length$) Assim, a sua complexidade será $O(1)$

Caso o array "a" tenha mais de 2 elementos, então a complexidade será $O(m) \times O(n) = O(m \times n)$. O primeiro ciclo é iterado m vezes, consoante o número de elementos do array ($m = a.length - 2$) e o seu ciclo intrínseco é iterado n vezes (n = número de elementos antes do elemento m).

```
4  public Node<E> lowestCommonAncestor (Node<E> n1, Node<E> n2,
                                          Node<E> root) {

       if (root getElement() == null) { return null; }

       if ( root getElement() < n1 getElement() &&
           root getElement() > n2 getElement()   ||
           root getElement() > n1 getElement() &&
           root getElement() < n2 getElement() ) {
           return root;
       }

       if (root getElement() < n1 getElement() &&
           root getElement() < n2 getElement() ) {
           return lowestCommonAncestor (n1,n2, root getRight());
       }

       if (root getElement() > n2 getElement() &&
           root getElement() > n2 getElement() ) {
           return lowestCommonAncestor (n1,n2, root getLeft());
       }

   public Map<Integer, List<E>> subArvore (Node<E> root, Node<E> n1,
                                           Node<E> n) {

      Node<E> anc = lowestCommonAncestor (n1, n2, root);
      List<E> lista = new ArrayList<>();
      Map <Integer, lista> map = new HashMap <>();
      findTree (anc, map, 0);
      return map;

      // procura o common ancestor
      / instancia a lista e o mapa
      / chama o findTree que adiciona ao mapa a raiz e os nós
```

```
public void findTree (Node<E> root, Map<Integer, List<E>> map, Integer level){

    if ( map.containsKey (level)) {
        List<E> aux = map.get (level);
        aux.add (root.getElement());
        map.put ( level, aux);
    } else {
        List<E> aux = new ArrayList<>();
        aux.add ( root.getElement());
        map.put (level , aux);
    }

    if ( root.getLeft() != null) {
        findTree (root.getLeft(), map, level+1); }

    if ( root.getRight() != null) {
        findTree( root.getRight(), map, level+1); }

5. public List<V> getElemsPath (int idx) {

    List<V> path = new ArrayList<>()

    // o index em questão pertence sempre à lista
    path.add ( heap[idx].getValue());
    // é calculado o index do primeiro parente (unidade a frente)
    int parent = (idx+1) / 2; // divisão inteira

    while ( parent != 1 || parent != 1.5) {
        // retira-se ao index a unidade somada no cálculo de parent
        path.add ( heap [parent-1].getValue());
        parent = parent/2; // divisão inteira
    }

    // adiciona a raiz à lista
    path.add ( heap [0].getValue());

    return path;
}
```

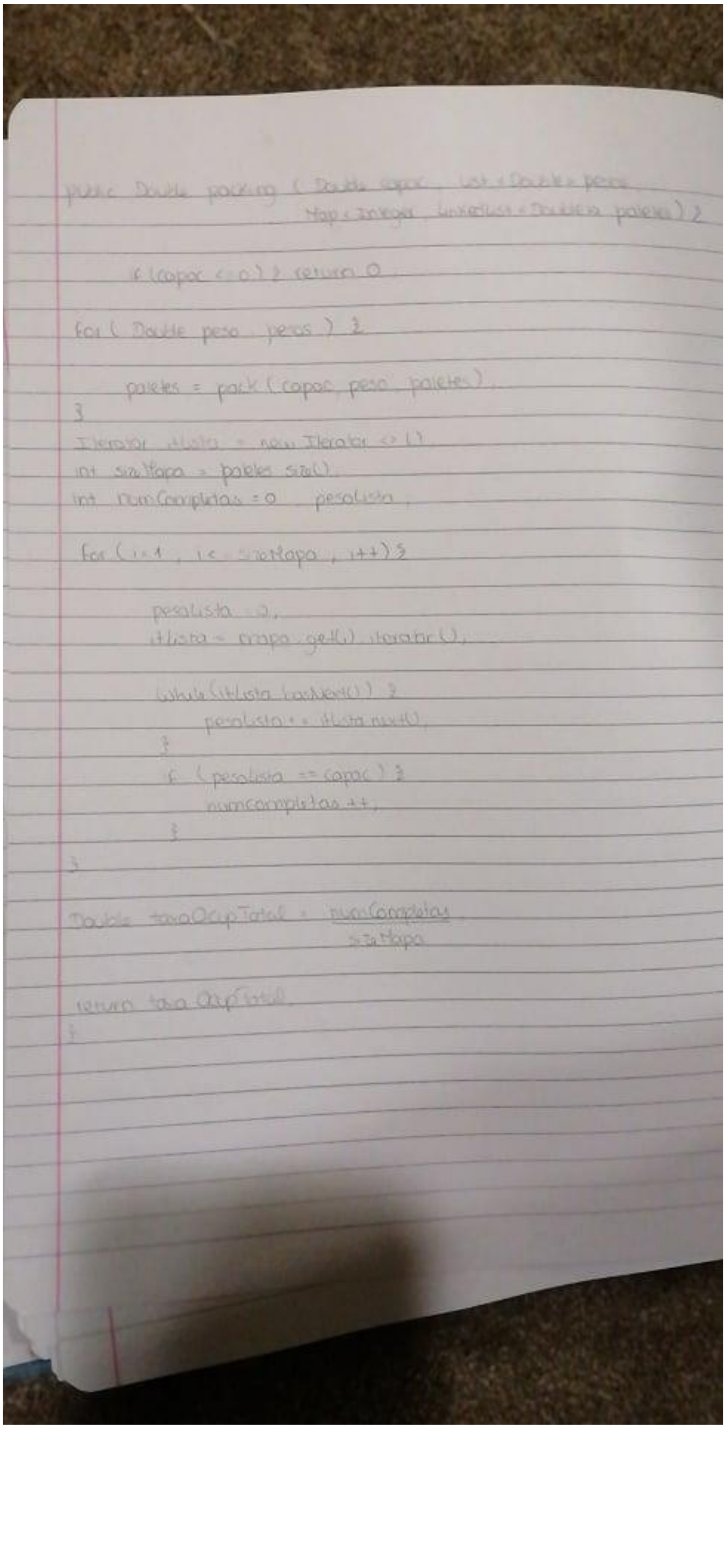
Recurso 2019



ESINF Exame Época Recurso 2019

```
1. public Map pack (Double capac, Double novoPeso,
                   Map<Integer, LinkedList<Double>> paletes) {
LinkedList<Double> lista = new LinkedList<>();
Iterator<Double> itLista = new Iterator<>();
int sizeMapa = paletes.size() + 1;
int i, pesoLinha;

/* para cada elemento do mapa, vai buscar a LinkedList de pesos cor-
respondente. Se o novo peso acrescentado ao peso total da lista, for in-
ferior à capacidade total, é adicionado a esse elemento do mapa. Senão
procura outro elemento do mapa que suporte o novo peso */

for (i = 1 ; i < sizeMapa ; i++) {

      lista = mapa.get(i);
      itLista = lista.iterator();
       pesoLinha = 0;


      while (itLista.hasNext()) {
            pesoLinha += itLista.next();
      }
      if ( pesoLinha + novoPeso <= capac) {
            lista.add(novoPeso);
            mapa.put(i, lista);
            return mapa;
       }
}
}
```

/* inicia-se sempre pelos primeiros elementos do mapa para se usar o número mínimo de paletes possível



```
public Double packing ( Double capac , List < Double > pesos ,
                       Map < Integer , LinkedList < Double >> paletes ) {

      if (capac <= 0 ) { return 0 ;

for ( Double peso : pesos ) {

      paletes = pack ( capac , peso , paletes ) ;
}
Iterator itLista = new Iterator <> () ;
int sizeMapa = paletes size() ;
int numCompletas = 0 , pesoLista ;

for ( i=1 , i <= sizeMapa , i++ ) {

        pesoLista = 0 ;
        itLista = mapa get(i) iterator() ;

        while (itLista hasNext()) {
             pesoLista += itLista next() ;
        }
        if ( pesoLista == capac ) {
             numcompletas ++ ;
        }
}

Double tasaOcupTotal = numCompletas / sizeMapa ;

return tasaOcupTotal ;
}
```

2.

a) tt = bana
pp = ana

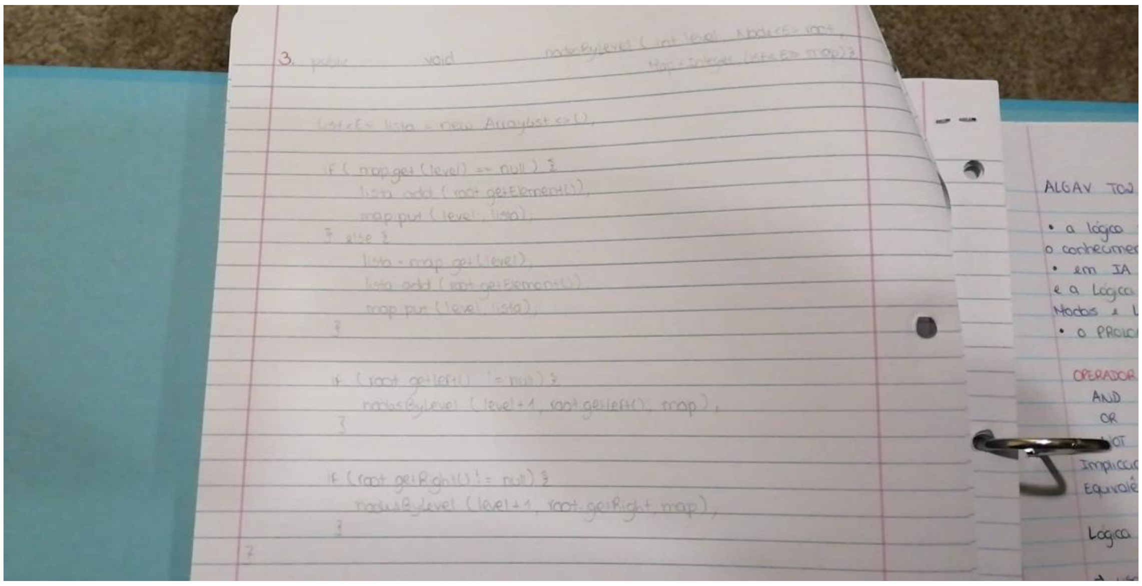
```
i=0 ; i<=4-3 ; i++ ?
    j=0
    j=0 ; j<3 && tt.charAt(0+0) = pp.charAt(0)  X

i=1 ; i<=4-3 ; i++ ?
    j=0
    j=0 ; j<3 && tt.charAt(1+0) == pp.charAt(0)  ✓
    j=1
    j=1 ; j<3 && tt.charAt(1+1) == pp.charAt(1)  ✓

    if ( j=3 == pp.length) ✓
        return i
```

$$\underbrace{tt.charAt(1+0)}_{a} == \underbrace{pp.charAt(0)}_{a}$$

$$\underbrace{tt.charAt(1+1)}_{n} == \underbrace{pp.charAt(1)}_{n}$$

O metodo mistery verifica se pp é substring de tt e, caso seja retorna o caracter de tt a partir do qual começa a substring

b) O metodo é não deterministico, já que pode ter complexidades alternativas consoante os dados introduzidos por parametro.
Na melhor hipotese, a complexidade é O(n), em que são percorridos os 'n' caracteres da lista 'tt' que são possiveis de ser o inicio da substring, não existindo substring e não se entrando nunca no ciclo "while".
Na pior hipotese, a complexidade é O(n)xO(m), em que, para cada 'n' do ciclo 'for', são percorridos 'm' elementos da substring 'pp' no ciclo "while".

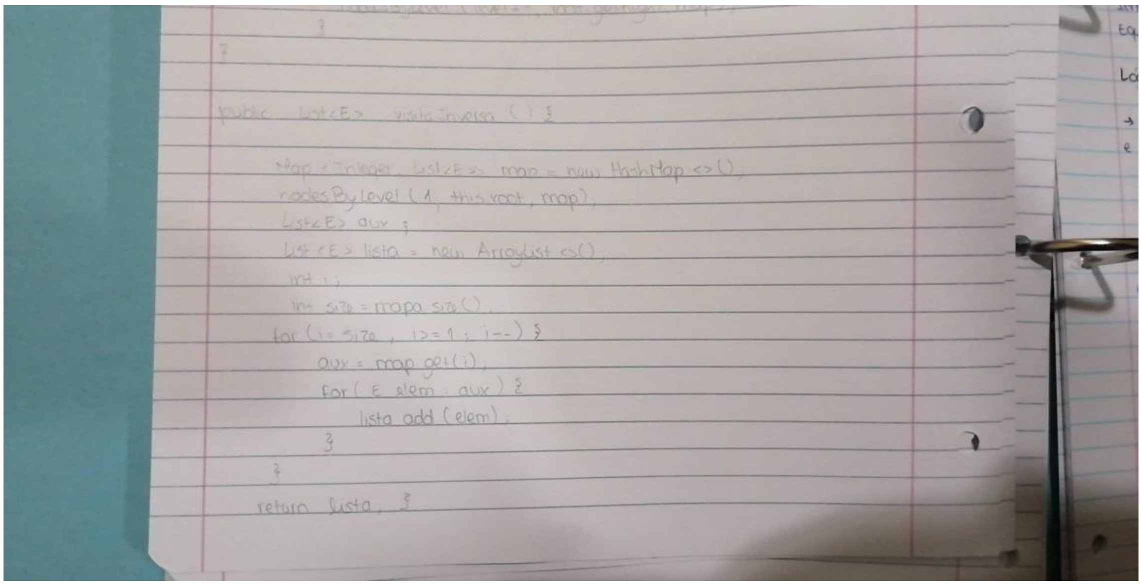

```
3. public        void        nodesByLevel ( int level, Node<E> root,
                                         Map<Integer, List<E>> map) {

   List<E> lista = new ArrayList <>();

   if ( map.get (level) == null ) {
       lista.add ( root.getElement());
       map.put ( level, lista);
   } else {
       lista = map.get(level);
       lista.add ( root.getElement());
       map.put (level, lista);
    }

    if (root.getLeft()  != null) {
        nodesByLevel ( level+1, root.getLeft(), map);
    }

    if (root.getRight()!= null) {
        nodesByLevel (level+1, root.getRight, map);
    }
}
```

ALGAV TO2

• a lógica
o conhecimen
• em IA
e a Lógica
Modos e L
• o PROL

OPERADOR
AND
OR
NOT
Implica
Equivalê

Lógica


```
public  List<E>  visitaInversa ( ) {

    Map<Integer, List<E>> map = new HashMap <>();
    nodesByLevel ( 1, this.root, map);
    List<E> aux ;
    List <E> lista = new ArrayList <>();
    int i;
    int size = mapa.size();
   for (i= size ,  i>= 1 ; i--) {
       aux = map.get(i);
       for( E elem : aux ) {
           lista.add (elem);
       }
   }
 return lista; }
```