

# Relatório do Trabalho de Estruturas de Dados Básicas

Técnicas de Programação Avançada — Ifes — Campus Serra

Alunos: David Vilaça, Harã Heique, Larissa Motta.

Prof. Jefferson O. Andrade

2019/2

## Sumário

<b>1</b>	<b>Introdução</b>	<b>6</b>
<b>2</b>	<b>Problemas</b>	<b>7</b>
2.1	Warm up . . . . .	7
2.1.1	UVa 12247 – Jollo . . . . .	7
2.2	Vetores . . . . .	10
2.2.1	UVa 10038 – Jolly Jumpers . . . . .	10
2.2.2	UVa 11340 – Newspaper . . . . .	13
2.3	Matrizes (Arrays 2D) . . . . .	19
2.3.1	UVa 10920 – Spiral Tap . . . . .	19
2.3.2	UVa 11581 – Grid successors . . . . .	22
2.4	Ordenação . . . . .	25
2.4.1	UVa 10107 – What is the Median? . . . . .	25
2.4.2	UVa 10258 – Contest Scoreboard . . . . .	27
2.5	Manipulação de bits . . . . .	32
2.5.1	UVa 10264 – The Most Potent Corner . . . . .	32
2.5.2	UVa 11926 – Multitasking . . . . .	35
2.6	Lista Encadeada . . . . .	41
2.6.1	UVa 11988 – Broken Keyboard . . . . .	41
2.7	Pilhas . . . . .	44
2.7.1	UVa 00514 – Rails . . . . .	44
2.7.2	UVa 01062 – Containers . . . . .	46
2.8	Filas . . . . .	49
2.8.1	UVa 10172 – The Lonesome Cargo . . . . .	49
2.8.2	UVa 10901 – Ferry Loading III . . . . .	53
2.9	Árvore Binária de Pesquisa (balanceada) . . . . .	57
2.9.1	UVa 00939 – Genes . . . . .	57
2.9.2	UVa 10132 – File Fragmentation . . . . .	61

2.10	Conjuntos . . . . .	65
2.10.1	UVa 00978 – Lemmings Battle . . . . .	65
2.10.2	UVa 11849 – CD . . . . .	71

## Lista de Códigos Fonte

1	Solução para o problema <i>UVa 12247 – Jollo</i> (pt 1).	8
2	Solução para o problema <i>UVa 12247 – Jollo</i> (pt 2).	9
3	Solução para o problema <i>UVa 10038 – Jolly Jumpers</i> (pt. 1).	11
4	Solução para o problema <i>UVa 10038 – Jolly Jumpers</i> (pt. 2).	12
5	Conjunto de dados de entrada fornecida pelo <i>uDebug</i> (pt. 1).	14
6	Conjunto de dados de entrada fornecida pelo <i>uDebug</i> (pt. 2).	15
7	Saída esperada dos dados de entrada fornecido pelo <i>uDebug</i> .	15
8	Solução para o problema <i>UVa 11340 – Newspaper</i> (pt. 1).	17
9	Solução para o problema <i>UVa 11340 – Newspaper</i> (pt. 2).	18
10	Solução para o problema <i>UVa 10920 – Spiral Tap</i> (pt. 1).	20
11	Solução para o problema <i>UVa 10920 – Spiral Tap</i> (pt. 2).	21
12	Solução para o problema <i>UVa 11581 – Grid successors</i> (pt. 1).	23
13	Solução para o problema <i>UVa 11581 – Grid successors</i> (pt. 2).	24
14	Solução para o problema <i>UVa 10107 – What is the Median?</i> .	26
15	Solução para o problema <i>UVa 10258 – Contest Scoreboard</i> (pt. 1).	28
16	Solução para o problema <i>UVa 10258 – Contest Scoreboard</i> (pt. 2).	29
17	Solução para o problema <i>UVa 10258 – Contest Scoreboard</i> (pt. 3).	30
18	Solução para o problema <i>UVa 10258 – Contest Scoreboard</i> (pt. 4).	31
19	Solução para o problema <i>UVa 10264 – The Most Potent Corner</i> (pt. 1).	33
20	Solução para o problema <i>UVa 10264 – The Most Potent Corner</i> (pt. 2).	34
21	Solução para o problema <i>UVa 11926 – Multitasking</i> (pt. 1).	36
22	Solução para o problema <i>UVa 11926 – Multitasking</i> (pt. 2).	37
23	Solução para o problema <i>UVa 11926 – Multitasking</i> (pt. 3).	38
24	Solução para o problema <i>UVa 11926 – Multitasking</i> (pt. 4).	39
25	Solução para o problema <i>UVa 11926 – Multitasking</i> (pt. 5).	40
26	Solução para o problema <i>UVa 11988 – Broken Keyboard</i> (pt. 1).	42
27	Solução para o problema <i>UVa 11988 – Broken Keyboard</i> (pt. 2).	43
28	Solução para o problema <i>UVa 00514 – Rails</i> (pt.1).	45
29	Solução para o problema <i>UVa 00514 – Rails</i> (pt.2).	46
30	Solução para o problema <i>UVa 01062 – Containers</i> (pt. 1).	47
31	Solução para o problema <i>UVa 01062 – Containers</i> (pt.2).	48
32	Solução para o problema <i>UVa 10172 – The Lonesome Cargo</i> (pt. 1).	50
33	Solução para o problema <i>UVa 10172 – The Lonesome Cargo</i> (pt.2).	51
34	Solução para o problema <i>UVa 10172 – The Lonesome Cargo</i> (pt.3).	52
35	Solução para o problema <i>UVa 10901 – Ferry Loading III</i> (pt.1).	54
36	Solução para o problema <i>UVa 10901 – Ferry Loading III</i> (pt.2).	55
37	Solução para o problema <i>UVa 10901 – Ferry Loading III</i> (pt.3).	56
38	Solução para o problema <i>UVa 00939 – Genes</i> (pt. 1).	58
39	Solução para o problema <i>UVa 00939 – Genes</i> (pt. 2).	59
40	Solução para o problema <i>UVa 00939 – Genes</i> (pt. 3).	60
41	Solução para o problema <i>UVa 10132 – File Fragmentation</i> (pt.1).	62
42	Solução para o problema <i>UVa 10132 – File Fragmentation</i> (pt.2).	63
43	Solução para o problema <i>UVa 10132 – File Fragmentation</i> (pt.3).	64
44	Solução para o problema <i>UVa 00978 – Lemmings Battle</i> (pt.1).	66
45	Solução para o problema <i>UVa 00978 – Lemmings Battle</i> (pt.2).	67

46	Solução para o problema <i>UVa 00978 – Lemmings Battle</i> (pt.3).	68
47	Solução para o problema <i>UVa 00978 – Lemmings Battle</i> (pt.4).	69
48	Solução para o problema <i>UVa 00978 – Lemmings Battle</i> (pt.5).	70
49	Solução para o problema <i>UVa 11849 – CD</i> .	72

## Lista de Figuras

1	Email de aceitação da solução <i>12247-Jollo</i> . . . . .	7
2	Email de aceitação da solução <i>UVa 10038 – Jolly Jumpers</i> . . . . .	10
3	Email de tempo limite excedido da solução <i>UVa 11340 – Newspaper</i> . . . . .	13
4	Resultados de saída do console <i>UVa 11430 – Newspaper</i> . . . . .	16
5	Email de aceitação da solução <i>UVa 10920 – Spiral Tap</i> . . . . .	19
6	Email de aceitação da solução <i>UVa 11581 – Grid successors</i> . . . . .	22
7	Email de aceitação da solução <i>UVa 10107 – What is the Median?</i> . . . . .	25
8	Email de aceitação da solução <i>UVa 10258 – Contest Scoreboard</i> . . . . .	27
9	Email de aceitação da solução <i>UVa 10264 – The Most Potent Corner</i> . . . . .	32
10	Email de aceitação da solução <i>UVa 10264 – The Most Potent Corner</i> . . . . .	35
11	Email de aceitação da solução <i>UVa 11988 – Broken Keyboard</i> . . . . .	41
12	Email de aceitação da solução <i>UVa 00514 – Rails</i> . . . . .	44
13	Email de aceitação da solução <i>UVa 01062 – Containers</i> . . . . .	46
14	Email de aceitação da solução <i>UVa 10172 – The Lonesome Cargo</i> . . . . .	49
15	Email de aceitação da solução <i>UVa 10901 – Ferry Loading III</i> . . . . .	53
16	Email de aceitação da solução <i>UVa 00939 – Genes</i> . . . . .	57
17	Email de aceitação da solução <i>UVa 10132 – File Fragmentation</i> . . . . .	61
18	Email de aceitação da solução <i>UVa 00978 – Lemmings Battle</i> . . . . .	65
19	Email de aceitação da solução <i>UVa 11849 – CD</i> . . . . .	71

# 1 Introdução

Estrutura de Dados é uma implementação concreta de um tipo abstrato de dado ou um tipo de dado básico ou primitivo. Neste trabalho é proposto a resolução de um conjunto de problemas envolvendo as estruturas de dados básicas vistas na disciplina, onde todos os problemas propostos estão disponíveis no site UVA Online Judge.

OS problemas foram resolvidos usando a linguagem de programação *Java* (versão JDK 11.0.3+) nos Ambientes Integrados de Desenvolvimento (IDE) *Eclipse* e o *Apache Netbeans*. As resoluções estão dispostos na seção 2 deste documento.

Todos os códigos e imagens presentes nesse trabalho estão disponíveis no GitHub.

## 2 Problemas

### 2.1 Warm up

#### 2.1.1 UVa 12247 – Jollo

De acordo com o Código Fonte 1 inicialmente é feito a leitura dos dados de entrada, verificando se a linha é contida toda de 0, caso seja é terminado a execução do programa e caso contrário faz-se a distribuição dos dados de acordo com o problema proposto, sendo as três primeiras cartas da princesa e as 2 últimas do príncipe, como é mostrado entre as linhas 24 e 27.

Para se descobrir se é possível o príncipe ganhar e qual seria o menor valor da carta é feito um loop presente a partir da linha 30 do método *main*, o qual é verificado todas as possibilidades de valores até achar-se o resultado, onde caso não encontrado retorna-se o valor  $-1$ , que significa que não tem possibilidade de ganhar. Baseado nisso, o método *worst* do Código Fonte 2 é o que desempenha o papel de checar a pior possibilidade de jogada de maneira recursiva.



Figura 1: Email de aceitação da solução 12247-Jollo.

```

1  import java.util.Scanner;
2  import java.util.Arrays;
3  import java.util.List;
4
5  public class Main {
6
7      private static Scanner scanner;
8      private static int[] prince, princess;
9      static boolean[] princePlayed, princessPlayed;
10
11     public static void main(String[] args) {
12         scanner = new Scanner(System.in);
13         String buf = scanner.nextLine();
14         int[] line = getLine(buf);
15         while (!isLineZeros(line)) {
16             int answer = -1;
17             princess = new int[3];
18             prince = new int[3];
19             princePlayed = new boolean[3];
20             princessPlayed = new boolean[3];
21
22             princess[0] = line[0];
23             princess[1] = line[1];
24             princess[2] = line[2];
25             prince[0] = line[3];
26             prince[1] = line[4];
27
28             for (int i = 1; i < 53 && answer == -1; i++) {
29                 boolean isValid = true;
30                 prince[2] = -1;
31                 for (int j = 0; j < 3; j++) {
32                     isValid = isValid && princess[j] != i && prince[j] != i;
33                 }
34                 if (isValid) {
35                     prince[2] = i;
36                     answer = worst(0, 0, -1) ? i : -1;
37                 }
38             }
39
40             System.out.println(answer);
41
42             buf = scanner.nextLine();
43             line = getLine(buf);
44         }
45     }
46 }

```

Código Fonte 1: Solução para o problema *UVa 12247 – Jollo* (pt 1).



```

1      // Recursive get the worst play
2      static boolean worst(int player, int lost, int prev) {
3          if (lost > 1) {
4              return false;
5          }
6          boolean ret = true;
7          for (int i = 0; i < 3; i++) {
8              if (player == 0 && !princePlayed[i]) { // prince player
9                  princePlayed[i] = true;
10                 ret = ret && worst(1, lost, prince[i]);
11                 princePlayed[i] = false;
12             }
13             if (player == 1 && !princessPlayed[i]) { // princess player
14                 princessPlayed[i] = true;
15                 int newLost = prev < princess[i] ? lost + 1 : lost;
16                 ret = ret && worst(0, newLost, -1);
17                 princessPlayed[i] = false;
18             }
19         }
20         return ret;
21     }
22
23     // get all line mapped to int from string input
24     public static int[] getLine(String line) {
25         List<String> res = Arrays.asList(line.split(" "));
26         int[] l = new int[5];
27         for (int i = 0; i < 5; i++) {
28             l[i] = Integer.parseInt(res.get(i));
29         }
30         return l;
31     }
32
33     // verify if all line elements is zero
34     public static boolean isLineZeros(int[] line) {
35         for (Integer element : line) {
36             if (element != 0) {
37                 return false;
38             }
39         }
40         return true;
41     }
42 }

```

Código Fonte 2: Solução para o problema UVa 12247 – Jollo (pt 2).

## 2.2 Vetores

### 2.2.1 UVa 10038 – Jolly Jumpers

Como é possível checar no Código Fonte 3 e Código Fonte 4 Inicialmente é feito a leitura dos dados de entrada, transformando em valores inteiros. Cada linha é entendido como uma sequência. Olhando os exemplos do site de debug do UVa, ficou entendido que o caso base é uma sequência de apenas 1 número, sendo Jolly. Quando a entrada possui uma sequência de mais de 1 número, é calculado a diferença absoluta e armazenado num Vetor de resultados "ArrayBool". Após os dados armazenados o Vetor de resultados é percorrido para saber se os elementos formam uma sequência de 1 até (n-1). Caso a afirmativa seja verdadeira a saída é que a sequência é Jolly.

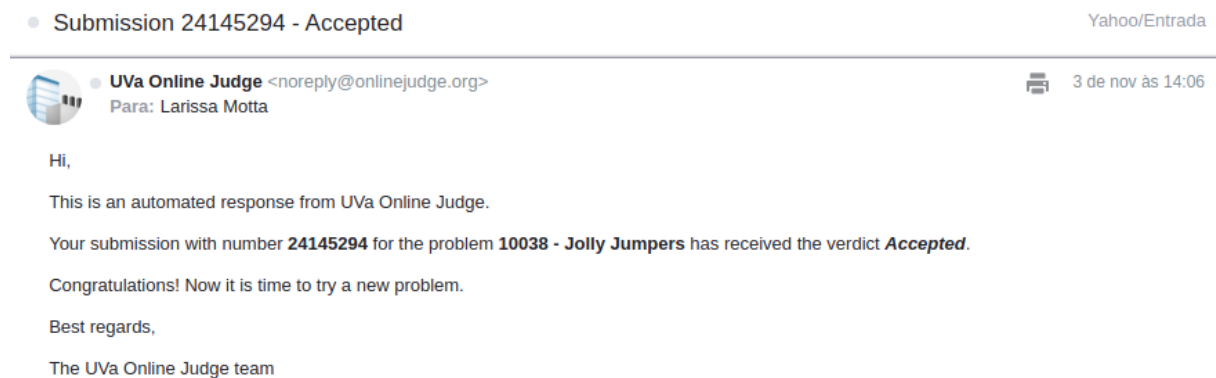


Figura 2: Email de aceitação da solução *UVa 10038 – Jolly Jumpers*.

```

1  import java.util.Scanner;
2  import java.util.Arrays;
3  import java.util.ArrayList;
4
5  public class Main {
6
7      private static Scanner scanner;
8
9      // Main function
10     public static void main(String[] args) {
11         scanner = new Scanner(System.in);
12         while (scanner.hasNextLine()) {
13             String buf = scanner.nextLine();
14             ArrayList<Integer> line = getLine(buf);
15
16             // It is understood through the "debug" site that the base case is
17             ↪ a sequence
18             // with size 1, being Jolly
19             if(line.get(0) == 1){
20                 output(true);
21                 continue;
22             }
23
24             boolean[] arrayBool = new boolean[line.get(0) - 1];
25             // Calculate the difference between sequence and store in
26             ↪ arrayBoll
27             for (int i = 1; i < line.get(0); i++) {
28                 int result = Math.abs(line.get(i) - line.get(i + 1));
29                 if(result > 0 && result < line.get(0)){
30                     arrayBool[result - 1] = true;
31                 }
32             }
33             boolean isJolly = true;
34
35             // Checks if the ArrayList "results" has a sequence from 1 to
36             ↪ (n-1)
37             for (boolean exist : arrayBool) {
38                 isJolly = isJolly && exist;
39             }
40             output(isJolly);
41         }
42     }
43 }

```

Código Fonte 3: Solução para o problema *UVa 10038 – Jolly Jumpers* (pt. 1).

```

1  // Print output
2  private static void output(boolean isJolly) {
3      if (isJolly)
4          System.out.println("Jolly");
5      else
6          System.out.println("Not jolly");
7  }
8
9  // get all line mapped to int from string input
10 public static ArrayList<Integer> getLine(String line) {
11     ArrayList<String> res = new ArrayList<>(Arrays.asList(line.split(" ")));
12     ArrayList<Integer> l = new ArrayList<>(res.size());
13     for (String element : res) {
14         l.add(Integer.parseInt(element));
15     }
16     return l;
17 }

```

Código Fonte 4: Solução para o problema *UVa 10038 – Jolly Jumpers* (pt. 2).

### 2.2.2 UVa 11340 – Newspaper

Como se pode notar no Código Fonte 8 inicialmente é feito a leitura do número de testes que será realizados por artigo. Logo depois é feita a leitura do número de caracteres a ser pago, onde nas linhas 18 e 19 são instanciados duas estruturas do tipo *ArrayList* do Java para armazenar o carácter e seu valor a ser pago quando encontrado no artigo.

No Código Fonte 9 é realizada a leitura do número de linhas que o artigo contém e para cada linha é percorrido carácter por carácter buscando seu índice no *ArrayList paid\_chars*, onde caso encontre (índice diferente de -1) é feita a soma do custo a ser pago pelo publisher.

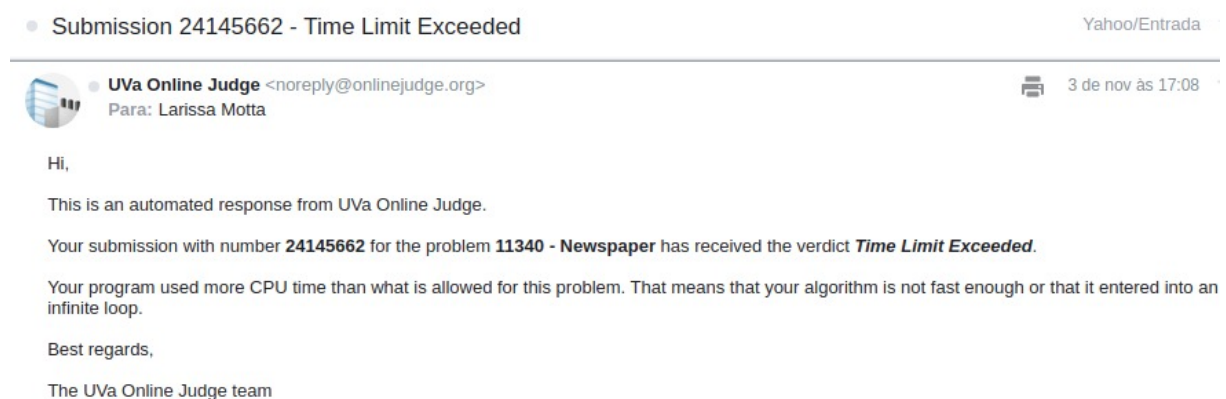


Figura 3: Email de tempo limite excedido da solução *UVa 11340 – Newspaper*.

Note que na Figura 3 o código submetido no UVa não foi aceito, pois entrou no estado de *Time Limit Exceeded* correspondendo que algoritmo não possui o desempenho bom o suficiente para a solução. Entretanto utilizando um conjunto de entradas provenientes do *uDebug* foi possível detectar que o algoritmo desenvolvido resolve o problema, ou seja, é correto, dado que obteve os resultados esperados.

1 5  
2 5  
3 b 33  
4 e 11  
5 c 101  
6 - 9  
7 y 20  
8 8  
9 He an thing rapid these after going drawn or.  
10 Timed she his law the spoil round defer.  
11 In surprise concerns informed betrayed he learning is ye. Ignorant formerly so  
↪ ye blessing.  
12 He as spoke avoid given downs money on we. Of properly carriage shutters  
13 ye as wandered up repeated moreover. Inquietude attachment if ye an solicitude  
↪ to.  
14 Remaining so continued concealed as knowledge happiness. Preference did how exp  
15 reSSION may favourable devon  
16 shire insipidity considered. An length design regret an hardly barton mr  
↪ figure.  
17 4  
18 i 72  
19 . 45  
20 o 799  
21 f 67  
22 5  
23 Cause dried no solid no an small so still widen. Ten weather evident smiling  
↪ bed against she examine its.  
24 Rendered far opinions two yet moderate sex striking. Sufficient motionless  
↪ compliment by stimulated assistance at. Convinced resolving extensive  
25 agreeable in it on as remainder. Cordially say affection met who propriety him.  
↪ Are man she towards private weather pleased.  
26 In more part he lose need so want rank no. At bringing or he sensible pleasure.  
↪ Prevent he parlors do waiting be f  
27 emales an message society.

Código Fonte 5: Conjunto de dados de entrada fornecida pelo *uDebug* (pt. 1).

```

1  2
2  " 999
3  R 72
4  6
5  His having within "saw become ask passed misery giving. Recommend questions get
   ↳ too fulfilled.
6  He fact in we case miss sake. Entrance be throwing he do blessing up.
7  Hearts warmth in genius do garden "advice" mr it garret. Collected preserved
   ↳ are middleton dependent residence but him how.
8  Handsome weddings yet" mrs you has carriage packages. Preferred joy "agreement
   ↳ put continual elsewhere delivered now.
9  Mrs exercise felicity had men speaking met. Rich deal mrs part
10 led pure will but.
11  2
12  a 3
13  z 100
14  2
15  this is a test.
16  aaaaaaaaaaaaaa.
17  1
18  a 3
19  1
20  zero

```

Código Fonte 6: Conjunto de dados de entrada fornecida pelo *uDebug* (pt. 2).

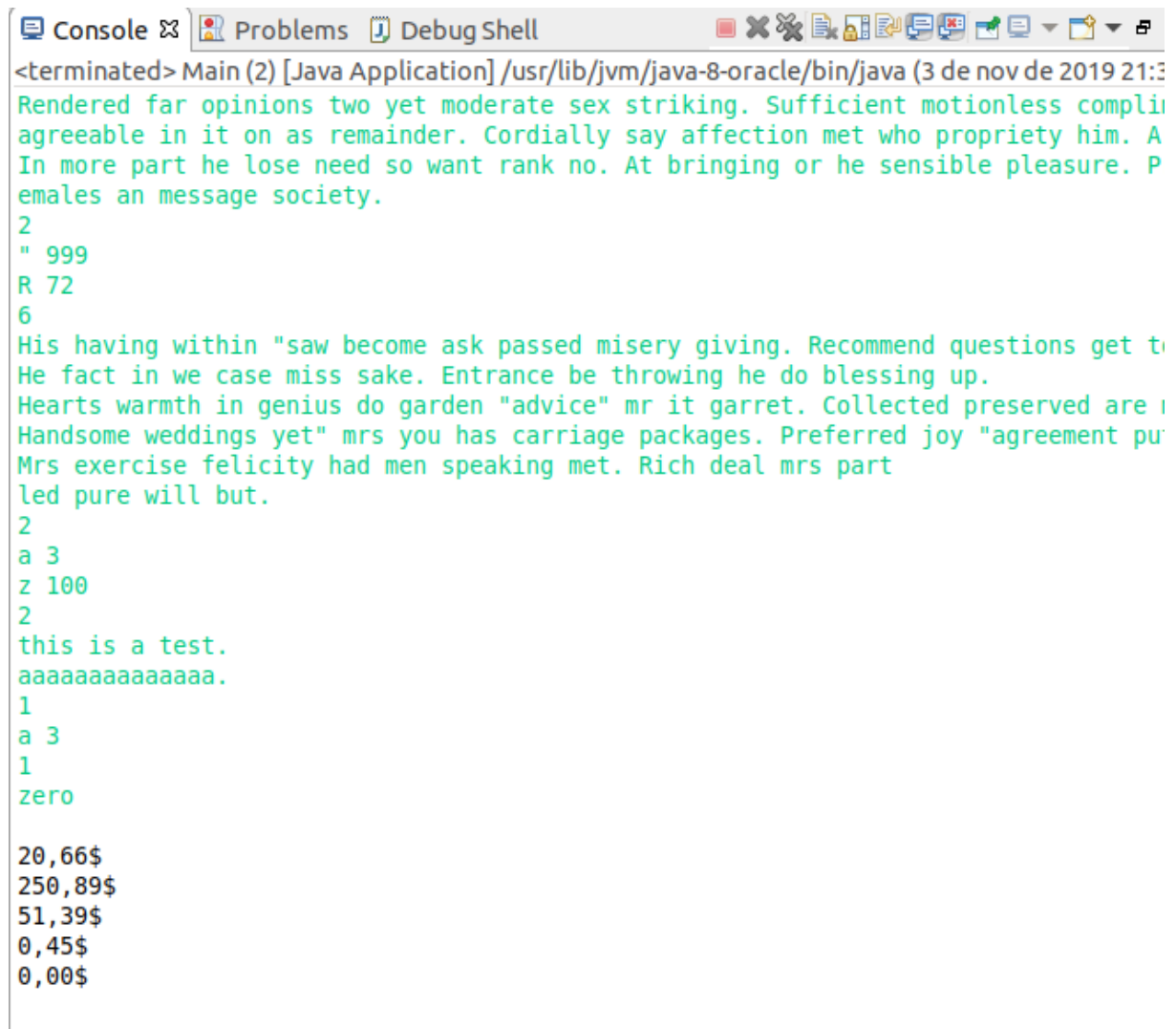
```

1  20.66$
2  250.89$
3  51.39$
4  0.45$
5  0.00$

```

Código Fonte 7: Saída esperada dos dados de entrada fornecido pelo *uDebug*.

Note que são feito realizados cinco testes diferentes, ou seja, esperando-se cinco resultados, onde na Figura 4 corresponde a saída no console da execução do algoritmo para o mesmo conjunto de entrada de dados e obtendo os mesmos valores dos resultados esperados apresentando no Código Fonte 7.



The screenshot shows an IDE's console window with tabs for 'Console', 'Problems', and 'Debug Shell'. The console output is as follows:

```
<terminated> Main (2) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (3 de nov de 2019 21:3
Rendered far opinions two yet moderate sex striking. Sufficient motionless compli
agreeable in it on as remainder. Cordially say affection met who propriety him. A
In more part he lose need so want rank no. At bringing or he sensible pleasure. P
emales an message society.
2
" 999
R 72
6
His having within "saw become ask passed misery giving. Recommend questions get to
He fact in we case miss sake. Entrance be throwing he do blessing up.
Hearts warmth in genius do garden "advice" mr it garret. Collected preserved are
Handsomeness yet" mrs you has carriage packages. Preferred joy "agreement pur
Mrs exercise felicity had men speaking met. Rich deal mrs part
led pure will but.
2
a 3
z 100
2
this is a test.
aaaaaaaaaaaaaaaa.
1
a 3
1
zero

20,66$
250,89$
51,39$
0,45$
0,00$
```

Figura 4: Resultados de saída do console UVa 11430 - Newspaper.



```

1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class Main {
5      private static Scanner scanner = new Scanner(System.in);
6
7      // Main function
8      public static void main(String[] args) {
9          int n = scanner.nextInt();
10         float money_paid = 0.00f;
11
12         // Looping through the number of tests
13         for (int i = 0; i < n; i++) {
14             int k = scanner.nextInt();
15             ArrayList<Character> paid_chars = new ArrayList<>();
16             ArrayList<Integer> paid_char_values = new ArrayList<>();
17
18             scanner.nextLine(); // Avoiding the \n problem in console
19
20             /*Looping through the paid characters and storing inside
21             ↪ arrays to check
22             * how much will pay to the current article */
23             for (int j = 0; j < k; j++) {
24                 String[] split_line = scanner.nextLine().split(" ");
25                 paid_chars.add(split_line[0].charAt(0));
26                 paid_char_values.add(Integer.parseInt(split_line[1]));
27             }
28         }
29     }
30 }

```

Código Fonte 8: Solução para o problema *UVa 11340 – Newspaper* (pt. 1).

```

1      int m = scanner.nextInt();
2      scanner.nextLine(); // Avoiding the \n problem in console
3
4      /* Looping through the article and calculate how much money publisher
   ↪ must
5      * pay for the article */
6      for (int l = 0; l < m; l++) {
7          String line_article = scanner.nextLine();
8
9          // Search for the index of character and realizes the sum if
   ↪ find it
10         for (int c = 0; c < line_article.length(); c++) {
11             int indexOfChar =
   ↪ paid_chars.indexOf(line_article.charAt(c));
12             if (indexOfChar != -1) {
13                 money_paid +=
   ↪ (float)(paid_char_values.get(indexOfChar)/100.00f);
14             }
15         }
16     }
17
18     System.out.printf("%.2f$\n", money_paid);
19     money_paid = 0.0f;
20 }
21 }
22 }

```

Código Fonte 9: Solução para o problema *UVa 11340 – Newspaper* (pt. 2).

## 2.3 Matrizes (Arrays 2D)

### 2.3.1 UVa 10920 – Spiral Tap

Primeiramente a entrada foi tratada como long, porque talvez o range do Integer não daria conta dos números grandes. A primeira solução foi desenvolvida criando um array 2d de todo tabuleiro, para conseguir chegar no valor de "P". Porém de alguma forma o programa dava TLE porque alguns tabuleiros eram grandes demais. Então resolvemos mudar a abordagem.

Como no problema diz, quando sz e p são 0 é porque a última linha foi lida, então essa verificação foi feita no início. Logo após nós identificamos o caso base que é p igual a 1, ou seja, p está exatamente no centro do tabuleiro, não sendo necessário nenhuma iteração para achar as coordenadas. Se p não for 1, então calculamos o limite superior e inferior onde p está com um loop (linha 48), dando voltas em espiral internamente com o cálculo  $(upper + 4 * (2 * count + 1) - 4)$ .

Logo após calcular os limites, x e y são inicializados com a posição central do espiral somando com a quantidade de voltas em espiral que foi necessária para calcular os limites. E com isso conseguimos caminhar em espiral partindo desses limites e indo incrementando x e y até o valor de p ser encontrado.

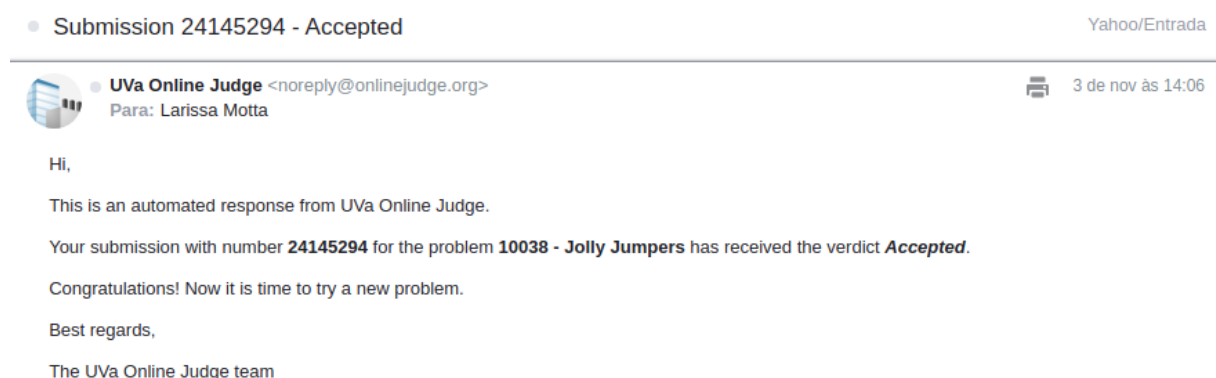


Figura 5: Email de aceitação da solução *UVa 10920 – Spiral Tap*.

```

1  import java.util.Scanner;
2  import java.util.Arrays;
3  import java.util.ArrayList;
4
5  public class Main {
6      private static Scanner scanner;
7
8      // Main function
9      public static void main(String[] args) {
10         scanner = new Scanner(System.in);
11         while (true) {
12             String buf = scanner.nextLine();
13             ArrayList<Long> line = getLine(buf);
14             long sz = line.get(0);
15             long p = line.get(1);
16             long x, y;
17             if (sz == 0 && p == 0) {
18                 break;
19             }
20
21             // p = 1 is the center of matrix
22             if (p == 1) {
23                 x = Math.round((sz / 2)) + 1;
24                 output(x, x);
25                 continue;
26             }
27
28             long lower = 1;
29             long upper = 1;
30             int count = 0;
31
32             // Find lower limit and upper limit
33             while (!(p >= lower && p <= upper)) {
34                 count++;
35                 lower = upper + 1;
36                 upper = upper + 4 * (2 * count + 1) - 4;
37             }
38
39             // x, y represents coordinates of upper bound
40             x = (sz + 1) / 2 + count;
41             y = (sz + 1) / 2 + count;
42
43             // flag and auxiliary to iterate the board in spiral
44             boolean isVertical = true;
45             int i = -1;
46             long actual = upper;
47             int lowerLimit = (int) (sz + 1) / 2 - count;
48             int upperLimit = (int) (sz + 1) / 2 + count;

```

Código Fonte 10: Solução para o problema *UVa 10920 – Spiral Tap* (pt. 1).

```

1      // find the piece "p" iterating the board in spiral mode
2      while (actual != p) {
3
4          if (isVertical) {
5              x += i;
6          } else {
7              y += i;
8          }
9
10         if (isVertical) { // vertical moviment
11             if (i == -1 && x == lowerLimit) {
12                 isVertical = false;
13             } else if (i == 1 && x == upperLimit) {
14                 isVertical = false;
15                 upperLimit--;
16             }
17         } else { // horizontal moviment
18             if (i == -1 && y == lowerLimit) {
19                 isVertical = true;
20                 i = 1;
21                 lowerLimit++;
22             } else if (i == 1 && y == upperLimit) {
23                 isVertical = true;
24                 i = -1;
25             }
26         }
27         actual--;
28     }
29
30     output(x, y);
31 }
32
33
34 // Print output coordinates
35 public static void output(long x, long y) {
36     System.out.printf("Line = %d, column = %d.\n", x, y);
37 }
38
39 // get all line mapped to int from string input
40 public static ArrayList<Long> getLine(String line) {
41     ArrayList<String> res = new ArrayList<>(Arrays.asList(line.split("
↵ ")));
42     ArrayList<Long> l = new ArrayList<>(res.size());
43     for (String element : res) {
44         l.add(Long.parseLong(element));
45     }
46     return l;
47 }
48 }

```

Código Fonte 11: Solução para o problema UVa 10920 – Spiral Tap (pt. 2).

### 2.3.2 UVa 11581 – Grid successors

O problema é sobre uma matriz 3x3 preenchida com números 0 ou 1 e define uma função que transforma cada célula dessa matriz na soma com resto da divisão por 2 (módulo 2) com seu adjacente. Essa função foi definida no código com nome de  $fn$ , como é mostrado no Código Fonte 12, e é aplicada enquanto o grid não está totalmente preenchido com números 0.

Como o caso base é a matriz preenchida apenas com números 0, iniciamos o contador de saída com -1 (linha 23) assim se a entrada já for uma matriz zerada o índice retornado já será -1 como esperado. Ao final do loop o contador conterá o maior índice onde  $k_g(f^{(i)}(g))$  é finito.

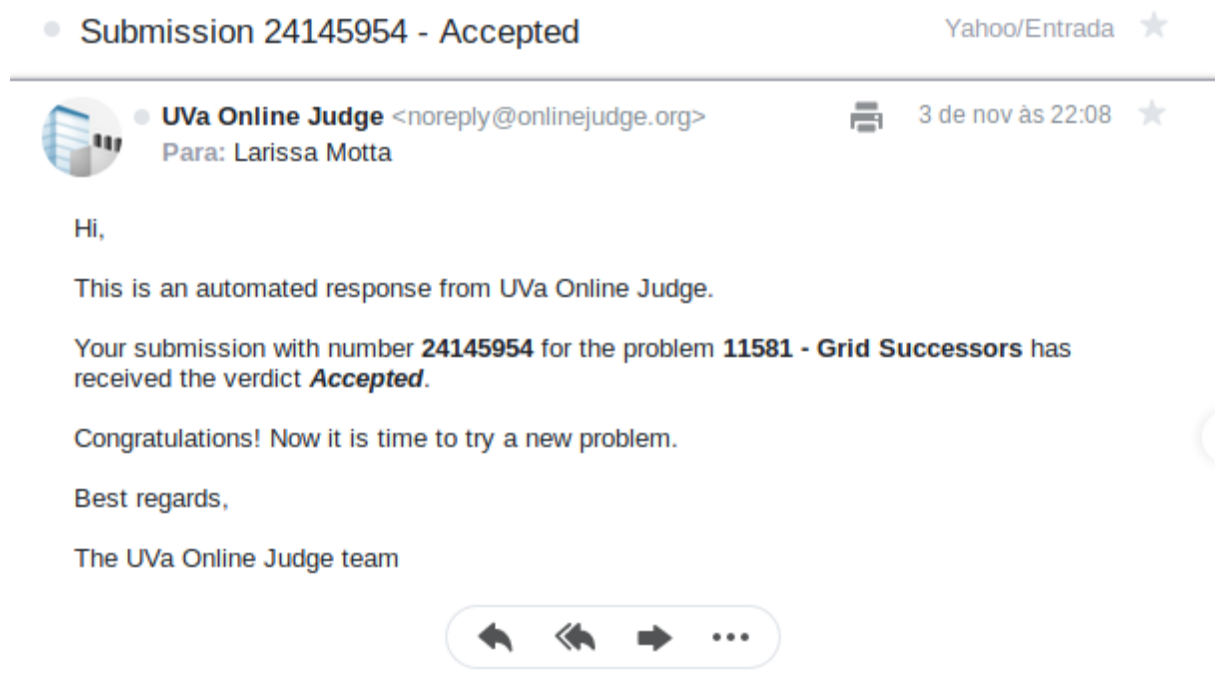


Figura 6: Email de aceitação da solução *UVa 11581 – Grid successors*.

```

1  import java.util.Scanner;
2
3  public class Main {
4
5      private static Scanner scanner;
6
7      // Main function
8      public static void main(String[] args) {
9          scanner = new Scanner(System.in);
10         // total amount of grids
11         int n = Integer.parseInt(scanner.nextLine());
12         for (int i = 0; i < n; i++) {
13             scanner.nextLine(); // blank line
14             int[][] grid = new int[3][3];
15             // fill the grid
16             grid[0] = getLine(scanner.nextLine());
17             grid[1] = getLine(scanner.nextLine());
18             grid[2] = getLine(scanner.nextLine());
19             // debugPrint(grid);
20
21             int output = -1;
22             while (!isZeroGrid(grid)) {
23                 grid = fn(grid);
24                 // debugPrint(grid);
25                 output++;
26             }
27
28             output(output);
29         }
30     }
31     // function described in problem to apply in grid
32     private static int[][] fn(int[][] grid) {
33         int[][] h = new int[3][3];
34         // line 0
35         h[0][0] = (grid[0][1] + grid[1][0]) % 2;
36         h[0][1] = (grid[0][0] + grid[1][1] + grid[0][2]) % 2;
37         h[0][2] = (grid[0][1] + grid[1][2]) % 2;
38         // line 1
39         h[1][0] = (grid[0][0] + grid[1][1] + grid[2][0]) % 2;
40         h[1][1] = (grid[0][1] + grid[1][0] + grid[1][2] + grid[2][1]) % 2;
41         h[1][2] = (grid[1][1] + grid[0][2] + grid[2][2]) % 2;
42         // line 2
43         h[2][0] = (grid[2][1] + grid[1][0]) % 2;
44         h[2][1] = (grid[2][0] + grid[1][1] + grid[2][2]) % 2;
45         h[2][2] = (grid[2][1] + grid[1][2]) % 2;
46
47         return h;
48     }

```

Código Fonte 12: Solução para o problema UVa 11581 – Grid successors (pt. 1).

```

1      // print index output
2      private static void output(int index) {
3          System.out.println(index);
4      }
5
6      // check if the current grid is only filled with 0
7      private static boolean isZeroGrid(int[][] grid) {
8          for (int i = 0; i < 3; i++) {
9              for (int j = 0; j < 3; j++) {
10                 if (grid[i][j] != 0) {
11                     return false;
12                 }
13             }
14         }
15         return true;
16     }
17
18     // get all line mapped to int from string input
19     public static int[] getLine(String line) {
20         char[] res = line.toCharArray();
21         int[] mappedL = new int[3];
22         for (int i = 0; i < 3; i++) {
23             char[] data = { res[i] };
24             mappedL[i] = Integer.parseInt(new String(data));
25         }
26         return mappedL;
27     }
28 }

```

Código Fonte 13: Solução para o problema *UVa 11581 – Grid successors* (pt. 2).



## 2.4 Ordenação

### 2.4.1 UVa 10107 – What is the Median?

No Código Fonte 14 inicialmente é criada uma lista de inteiros onde são armazenados os valores lidos como entrada, enquanto houver dados. Essa lista é ordenada, usando a função nativa do java `Collections.sort()`, e armazena-se a posição central da lista. Verifica-se o tamanho da lista, caso seja par é retornado a divisão inteira da soma dos seus valores centrais por 2, mostrado na linha 23, caso contrário é retornado o valor na posição central, mostrado na linha 26.

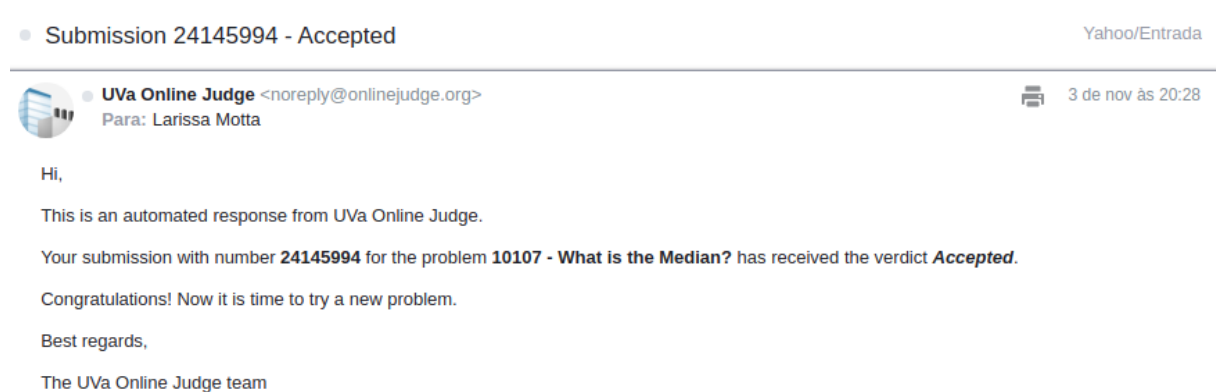


Figura 7: Email de aceitação da solução *UVa 10107 – What is the Median?*.

```

1  import java.util.Scanner;
2  import java.util.ArrayList;
3  import java.util.Collections;
4
5  public class Main {
6
7      private static Scanner scanner;
8
9      // Main function
10     public static void main(String[] args) {
11         scanner = new Scanner(System.in);
12         ArrayList<Integer> lstValues = new ArrayList<>();
13         while (scanner.hasNext()) {
14             int value = scanner.nextInt();
15             lstValues.add(value);
16             Collections.sort(lstValues);
17
18             int half = (lstValues.size() / 2);
19
20             if(lstValues.size() % 2 == 0){
21                 System.out.println((lstValues.get(half - 1) +
22                                     ↪ lstValues.get(half)) / 2);
23             }
24             else{
25                 System.out.println(lstValues.get(half));
26             }
27         }
28     }

```

Código Fonte 14: Solução para o problema *UVa 10107 – What is the Median?*.

### 2.4.2 UVa 10258 – Contest Scoreboard

Como é possível notar no Código Fonte 16 o algoritmo se inicia com a leitura do número de casos do concurso seguido da leitura de cada um dos casos em um loop de enquanto tiver próximo linha na entrada é realizado a leitura (linha 7). Após feito a leitura da "fila de julgamento" (linha 8) é realizada a checagem se o participante existe no vetor (linha 24). Após isso é checado se o participante resolveu o problema, onde caso não tenha resolvido é feita a análise do carácter *L*, cujo significado é de estado da submissão. Caso estado seja incorreto é somado no array de penalidades do problema presente no objeto instanciado da classe *Contestant*. Caso o estado seja correto é registrado na estrutura de dados *map* também presente como atributo do objeto da classe *Contestant*, onde este possui como chave o número do problema e como valor a soma de toda a penalidade do determinado problema resolvido.

Como se pode notar no Código Fonte 18 está presente a classe *Contestant*, cujo objetivo é representar a entidade participante do problema. Outro ponto importante para sua criação é a utilização da interface *Comparable< T >*, onde basta implementar o método de comparação, chamado *CompareTo* presente na linha 28, para o processo de ordenação solicitada no problema.

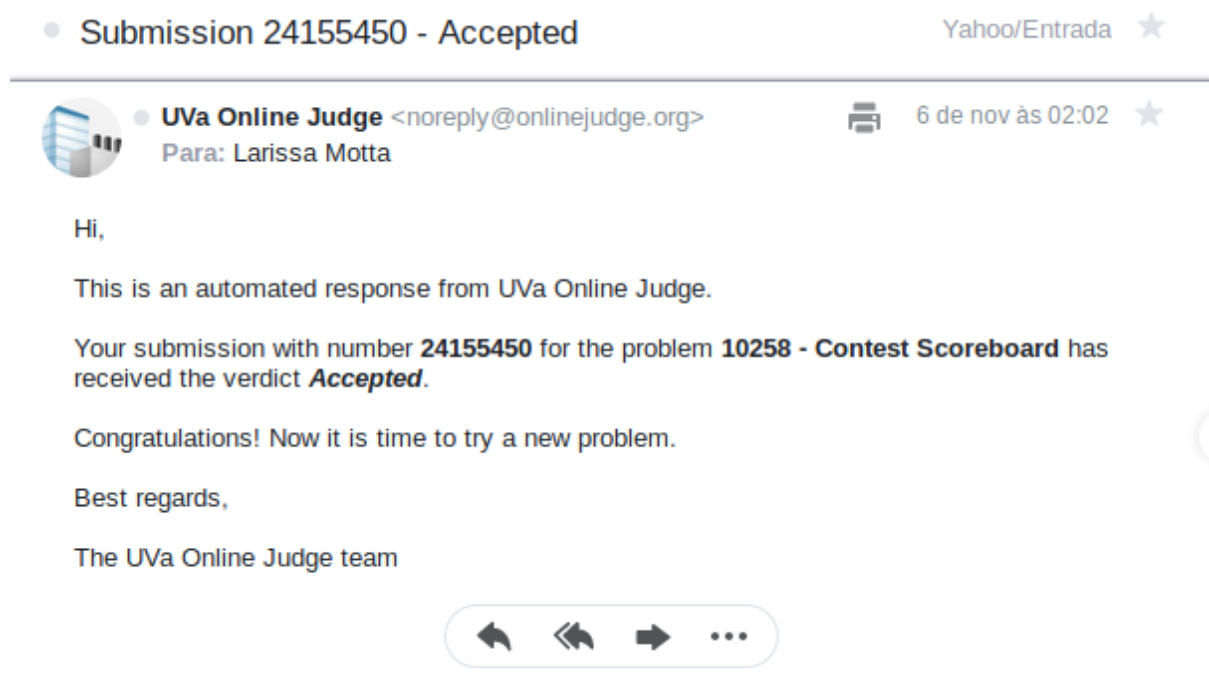


Figura 8: Email de aceitação da solução *UVa 10258 – Contest Scoreboard*.

```
1 import java.util.Scanner;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.HashMap;
5
6 public class Main {
7     private static final Scanner SC = new Scanner(System.in);
8     private static final int INCORRECT_SUBMISSION = 20;
9     private static Contestant[] contestants;
```

Código Fonte 15: Solução para o problema *UVa 10258 – Contest Scoreboard* (pt. 1).

```

1  public static void main(String[] args) {
2      SC.nextInt();
3      ignoreLines(2); // Avoiding the \n problem in terminal and ignoring next
        ↳ blank line
4      contestants = new Contestant[101];
5
6      // While there are lines with the scores will read it
7      while (SC.hasNextLine()) {
8          String lineJudgeQueue = SC.nextLine();
9
10         if (lineJudgeQueue.isEmpty()) {
11             output();
12             System.out.println();
13             contestants = new Contestant[101];
14             continue;
15         }
16
17         String[] tokensJudgeQueue = lineJudgeQueue.split(" ");
18         int numberCon = Integer.parseInt(tokensJudgeQueue[0]);
19         int problem = Integer.parseInt(tokensJudgeQueue[1]);
20         int penalty = Integer.parseInt(tokensJudgeQueue[2]);
21         char L = tokensJudgeQueue[3].charAt(0);
22
23         // Check if the Contestant exists or not
24         if (contestants[numberCon] == null) {
25             contestants[numberCon] = new Contestant(numberCon);
26         }
27
28         Contestant current = contestants[numberCon];
29
30         // If the key already exists is because was already solved
31         if (current.solvedProblems.get(problem) != null) {
32             continue;
33         }
34
35         // Incorrect question so +20 penalty
36         if (L == 'I') {
37             current.incorrectPenalty[problem] += INCORRECT_SUBMISSION;
38         }
39         /* Correct answer so register the penalty on solvedProblems map with
            ↳ the number
40         * of the problem as key
41         */
42         else if (L == 'C') {
43             current.solvedProblems.put(problem, penalty +
            ↳ current.incorrectPenalty[problem]);
44         }
45     }
46     output();
47 }

```

Código Fonte 16: Solução para o problema UVa 10258 – Contest Scoreboard (pt. 2).

```

1      // Ignore the number of lines in terminal
2  public static void ignoreLines(int numberOfLines) {
3      for (int i = 0; i < numberOfLines; i++) {
4          SC.nextLine();
5      }
6  }
7
8  public static void output() {
9      // Filter only the objects no nullables and puts inside a list
10     ArrayList<Contestant> noNullablesContestant = new ArrayList<>();
11
12     for (int i = 0; i < contestants.length; i++) {
13         if (contestants[i] != null) {
14             noNullablesContestant.add(contestants[i]);
15         }
16     }
17
18     // Sorting by the rules specified in the class Contestant
19     Collections.sort(noNullablesContestant);
20
21     // Print the output result
22     for (int i = 0; i < noNullablesContestant.size(); i++) {
23         System.out.println(noNullablesContestant.get(i).toString());
24     }
25 }
26 }

```

Código Fonte 17: Solução para o problema *UVa 10258 – Contest Scoreboard* (pt. 3).

```

1  // Contestants model representation of the contest
2  final class Contestant implements Comparable<Contestant> {
3      public int num;
4      public HashMap<Integer, Integer> solvedProblems; // key:problem &
        ↪ value:penalty
5      public int[] incorrectPenalty;
6
7      public Contestant(int numContestant) {
8          this.num = numContestant;
9          solvedProblems = new HashMap<>();
10         incorrectPenalty = new int[10];
11     }
12
13     public int calculteTotalPenalty() {
14         int result = 0;
15         return this.solvedProblems.values().stream().map((value) ->
            ↪ value).reduce(result, Integer::sum);
16     }
17
18     public int qntProblemsSolved() {
19         return this.solvedProblems.size();
20     }
21
22     @Override
23     public String toString() {
24         return String.format("%s %s %s", this.num, this.qntProblemsSolved(),
            ↪ this.calculteTotalPenalty());
25     }
26
27     @Override
28     public int compareTo(Contestant contestant) {
29         if (contestant == null || this.qntProblemsSolved() >
            ↪ contestant.qntProblemsSolved()) {
30             return -1;
31         }
32
33         if (this.qntProblemsSolved() == contestant.qntProblemsSolved() &&
            ↪ this.calculteTotalPenalty() < contestant.calculteTotalPenalty()) {
34             return -1;
35         }
36
37         if (this.calculteTotalPenalty() == contestant.calculteTotalPenalty() &&
            ↪ this.num < contestant.num) {
38             return -1;
39         }
40
41         return 1;
42     }
43 }

```

Código Fonte 18: Solução para o problema *UVa 10258 – Contest Scoreboard* (pt. 4).

## 2.5 Manipulação de bits

### 2.5.1 UVa 10264 – The Most Potent Corner

De acordo com o Código Fonte 19 primeiramente é lido a dimensão do cubo (linha 12) e chamada a função responsável por fazer a leitura dos pesos de cada "canto", denominado como *corner* no problema. O número de linhas lidas é  $2^N$  indicando o número de *corners/vértices* do cubo, onde cada um terá seu peso.

Na segunda etapa é realizado o cálculo da potência de cada *corner* baseado em seu peso através da função *calculateCornersPotency* presente no Código Fonte 20. Entre as linhas 12 e 16 está presente o loop responsável por calcular a soma das potências de cada vizinho do *corner* atual.

Por fim na terceira etapa é encontrado a maior soma de potência de dois *corners* vizinhos entre todos eles da dimensão do cubo corrente. A estratégia adotada foi armazenar as somas das potências em um ArrayList e depois recuperar o maior elemento entre eles utilizado o método *max* da classe *Collections*. Note também que para conseguir o índice do vizinho é utilizada o operador bit a bit à esquerda, o qual equivale a  $2^j$ , onde este é elevado a  $i$  ( $i^{2^j}$ ).

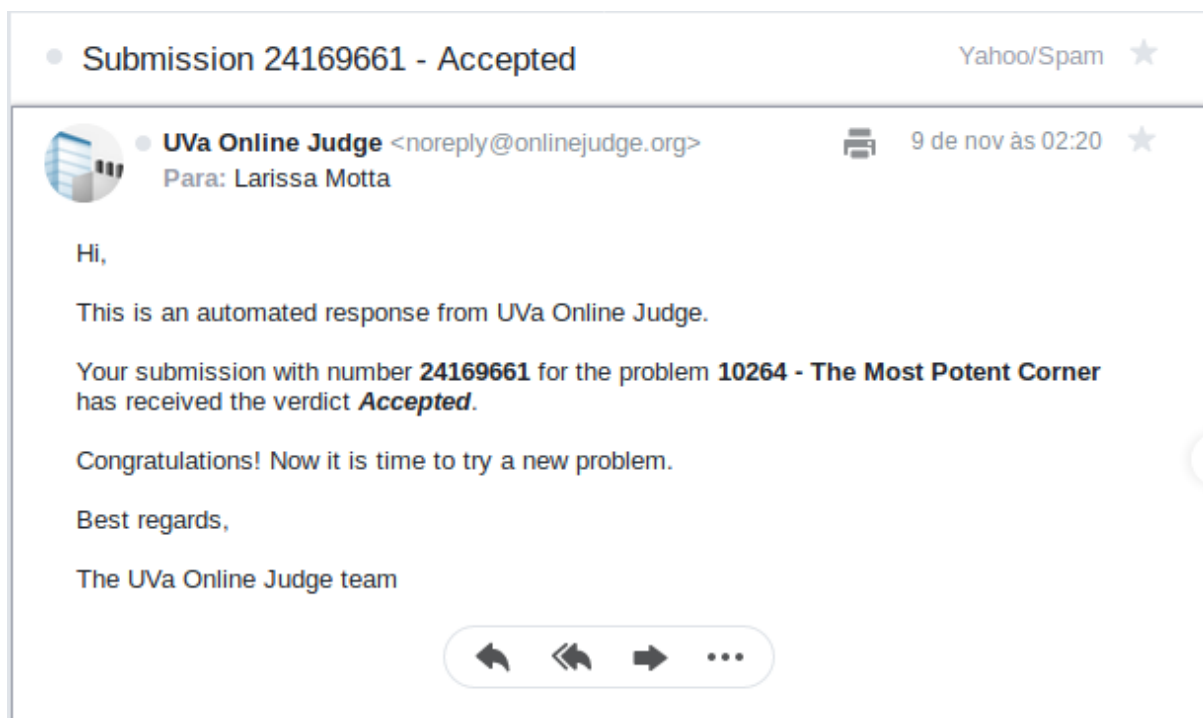


Figura 9: Email de aceitação da solução UVa 10264 – The Most Potent Corner.



```

1  import java.util.Scanner;
2  import java.util.ArrayList;
3  import java.util.Collections;
4  import java.util.List;
5
6  public class Main {
7
8      private static final Scanner SC = new Scanner(System.in);
9
10     public static void main(String[] args) {
11         while (SC.hasNextLine()) {
12             int n = SC.nextInt();
13             ignoreLines(1);
14
15             List<Integer> weightCorners = takeWeightCorners(n);
16             List<Integer> potencyCorners =
17                 ↪ calculateCornersPotency(weightCorners, n);
18             int maxSumPotency = maxPotencySumNeighborCorners(potencyCorners,
19                 ↪ n);
20
21             System.out.println(maxSumPotency);
22         }
23     }
24
25     // Get all weight corners based on dimension of the cube
26     public static List<Integer> takeWeightCorners(int n) {
27         ArrayList<Integer> weightCorners = new ArrayList();
28
29         for (int i = 0; i < Math.pow(2, n); i++) {
30             weightCorners.add(SC.nextInt());
31             ignoreLines(1);
32         }
33
34         return weightCorners;
35     }
36 }

```

Código Fonte 19: Solução para o problema *UVa 10264 – The Most Potent Corner* (pt. 1).

```

1  // Calculate the potency of each corners based on their weights
2  public static List<Integer> calculateCornersPotency(List<Integer>
   ↪ weightCorners, int numberOfNeighbor) {
3      ArrayList<Integer> potencyCorners = new ArrayList();
4
5      // Iterates over the corners weight
6      for (int i = 0; i < weightCorners.size(); i++) {
7          int potenciesSum = 0;
8
9          /* Calculate the weight sum of each corner
10         * i = corner position; j = weight corner neighbor
11         */
12         for (int j = 0; j < numberOfNeighbor; j++) {
13             int indexNeighbor = i ^ (1 << j);
14             potenciesSum += weightCorners.get(indexNeighbor);
15         }
16         potencyCorners.add(potenciesSum);
17     }
18
19     return potencyCorners;
20 }
21
22 // Returns the max sum of potencies of two neighbor corners
23 public static int maxPotencySumNeighborCorners(List<Integer>
   ↪ potencyCorners, int numberOfNeighbor) {
24     ArrayList<Integer> potenciesSumNeighbor = new ArrayList();
25
26     for (int i = 0; i < potencyCorners.size(); i++) {
27         for (int j = 0; j < numberOfNeighbor; j++) {
28             int indexCurrentCorner = i;
29             int indexNeighbor = i ^ (1 << j);
30             potenciesSumNeighbor.add(potencyCorners.get(indexCurrentCorner)
   ↪ + potencyCorners.get(indexNeighbor));
31         }
32     }
33
34     return Collections.max(potenciesSumNeighbor);
35 }
36
37 // Ignore the number of lines in terminal
38 public static void ignoreLines(int numberOfLines) {
39     for (int i = 0; i < numberOfLines; i++) {
40         SC.nextLine();
41     }
42 }
43 }

```

Código Fonte 20: Solução para o problema UVa 10264 – The Most Potent Corner (pt. 2).

### 2.5.2 UVa 11926 – Multitasking

No Código Fonte 21 está presente a função *main* que contém e manipula todo fluxo de chamadas das funções responsável por averiguar se grupos de tarefas de entrada possui conflito ou não. Inicialmente é lido a linha de entrada correspondendo a quantidade de tarefas *one-time* e posteriormente a quantidade de tarefas *repetition*. Logo após é instanciado um vetor de inteiros de tamanho um milhão, correspondente a quantidade de minutos do problema proposto, onde cada posição do vetor é uma unidade de tempo. Lembrando que todos os valores default deste vetor é zero, o qual acaba funcionando como uma espécie de vetor binário, em que zero significa que não há tarefa não alocada e um é que há tarefa alocada, para a determinada unidade de tempo.

Para que seja averiguado se há ou não conflito na linha 35 Código Fonte 21 é realizada a chamada da função responsável por determinar se há conflitos em tarefas do tipo *one-time*. Para isso, como é mostrado no Código Fonte 22, para cada quantidade de tarefas é instanciado um objeto da classe *Task*, cujo este representa uma tarefa, e preenche no vetor de minutos com o valor 1 através do loop interno que inicia no start do tarefa até o seu end. Entretanto caso a posição a ser preenchida já contenha o valor 1 significa um conflito.

Já para averiguar se há conflito em tarefas do tipo *repetition* usa a mesma abordagem que a anteriormente explicada. No Código Fonte 23 realiza esta etapa, onde possui como diferença em relação ao outro a presença do intervalo. A estratégia adotada foi criar um loop externo que roda infinitamente até que a tarefa atual possua seu fim, atributo end, igual ao valor total de minutos (1 milhão) é interrompido o loop. Outra forma também de interromper o loop é caso encontre algum ponto de conflito entre as tarefas. Na linha 25 há a chamada do método *setInterval* do objeto da classe *Task*, cuja função é definir o novo valor tanto do start quanto do end da tarefa levando em consideração o tempo máximo das tarefas.

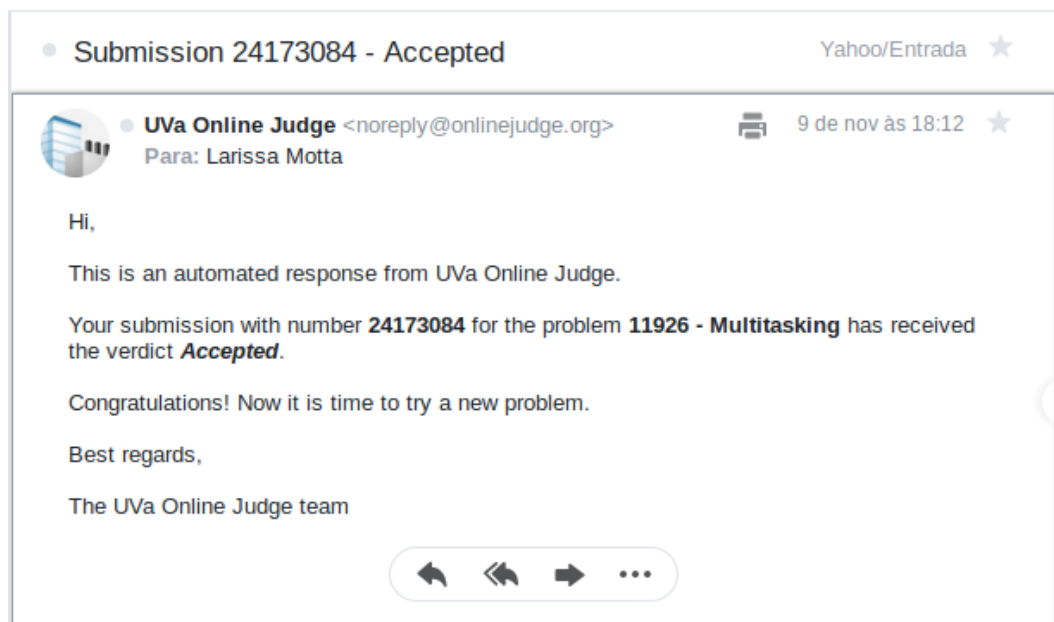


Figura 10: Email de aceitação da solução *UVa 10264 – The Most Potent Corner*.

```

1  import java.util.Scanner;
2
3  public class Main {
4
5      private static final Scanner SC = new Scanner(System.in);
6      private static final int TOTAL_MINUTES = 1000000;
7
8      public static void main(String[] args) {
9          boolean hasAnyConflict = false;
10
11          // Iterates over until finds a 0 0 line input
12          while (SC.hasNextLine()) {
13              String[] numberOfTasksTokens = SC.nextLine().split(" ");
14
15              if (!hasNextTask(numberOfTasksTokens)) {
16                  break;
17              }
18
19              int qntOneTimeTask = Integer.parseInt(numberOfTasksTokens[0]);
20              int qntRepeatingTask = Integer.parseInt(numberOfTasksTokens[1]);
21
22              /* Represents the minutes of the problem. Each position represents
23               * a minute. If the position equals 0 then can be allocated.
24               ↪ Otherwise
25               cannot, wich means there is a conflict
26               */
27              int[] minutes = new int[TOTAL_MINUTES + 10];
28
29              // First check if exists at least one conflict with one time tasks
30              hasAnyConflict = hasConflictOneTimeTasks(qntOneTimeTask, minutes);
31
32              // Second check if exists at least one conflict with reapeating
33              ↪ tasks
34              if (!hasAnyConflict) {
35                  hasAnyConflict = hasConflictRepeatingTasks(qntRepeatingTask,
36                  ↪ minutes);
37              } else {
38                  // If there is a conflict ignore the rest of lines to go to the
39                  ↪ next group tasks
40                  ignoreLines(qntRepeatingTask);
41              }
42
43              // Prints if there is or not a conflict
44              output(hasAnyConflict);
45          }
46      }
47  }

```

Código Fonte 21: Solução para o problema UVa 11926 – Multitasking (pt. 1).

```

1  //
2  private static boolean hasNextTask(String[] nTasksTokens) {
3      return nTasksTokens[0].charAt(0) != '0' || nTasksTokens[1].charAt(0) !=
        ↪ '0';
4  }
5
6  // Check if there is conflict, ie, if there is overlap of one time tasks
7  private static boolean hasConflictOneTimeTasks(int qntTasks, int[]
        ↪ minutesTasksTime) {
8      for (int i = 1; i <= qntTasks; i++) {
9          Task currentTask = getTask(SC.nextLine());
10
11         /* Iterates over time array and puts 1 to each position minute
12          * from start to end task time. If position already has 1 value
13          * means there is a conflict. Otherwise there isn't.
14          */
15         for (int min = currentTask.start; min < currentTask.end; min++) {
16             if (minutesTasksTime[min] == 1) {
17                 // If finds any conflict reads the rest of lines to go to
                ↪ the next task
18                 ignoreLines(qntTasks - i);
19                 return true;
20             }
21
22             minutesTasksTime[min] = 1;
23         }
24     }
25
26     return false;
27 }

```

Código Fonte 22: Solução para o problema *UVa 11926 – Multitasking* (pt. 2).

```

1  // Check if there is conflict, ie, if there is overlap of one time tasks
2  private static boolean hasConflictRepeatingTasks(int qntTasks, int[]
    ↪ minutesTasksTime) {
3      for (int i = 1; i <= qntTasks; i++) {
4          Task currentTask = getTask(SC.nextLine());
5
6          /* Iterates over time array and puts 1 to each position minute
7           * from start to end task time. If position already has 1 value
8           * means there is a conflict. Otherwise there isn't.
9           */
10         while (true) {
11             for (int min = currentTask.start; min < currentTask.end; min++)
12                 ↪ {
13                     if (minutesTasksTime[min] == 1) {
14                         // If finds any conflict reads the rest of lines to go
15                         ↪ to the next task
16                         ignoreLines(qntTasks - i);
17                         return true;
18                     }
19                     minutesTasksTime[min] = 1;
20                 }
21             if (currentTask.end == TOTAL_MINUTES) {
22                 break;
23             }
24             currentTask.setInterval(TOTAL_MINUTES);
25         }
26     }
27 }
28
29 return false;
30 }

```

Código Fonte 23: Solução para o problema *UVa 11926 – Multitasking* (pt. 3).

```

1  // Returns a object Task to represents one time tasks or repeating tasks
2  private static Task getTask(String dataString) {
3      String[] tokensTime = dataString.split(" ");
4      int start = Integer.parseInt(tokensTime[0]);
5      int end = Integer.parseInt(tokensTime[1]);
6
7      if (tokensTime.length == 3) {
8          int interval = Integer.parseInt(tokensTime[2]);
9
10         return new Task(start, end, interval);
11     }
12
13     return new Task(start, end);
14 }
15
16 // Prints out the current result
17 private static void output(boolean hasConflict) {
18     if (hasConflict) {
19         System.out.println("CONFLICT");
20     } else {
21         System.out.println("NO CONFLICT");
22     }
23 }
24
25 // Ignore the number of lines in terminal
26 public static void ignoreLines(int numberOfLines) {
27     for (int i = 0; i < numberOfLines; i++) {
28         SC.nextLine();
29     }
30 }
31 }

```

Código Fonte 24: Solução para o problema *UVa 11926 – Multitasking* (pt. 4).

```

1  final class Task {
2
3      int start;
4      int end;
5      int interval;
6
7      public Task(int start, int end) {
8          this.start = start;
9          this.end = end;
10         this.interval = 0;
11     }
12
13     public Task(int start, int end, int interval) {
14         this.start = start;
15         this.end = end;
16         this.interval = interval;
17     }
18
19     public int spentTime() {
20         return this.end - this.start;
21     }
22
23     // Negative time limit means there is no time limit. Otherwise has a time
24     ↪ limit
25     public void setInterval(int maxlimitTime) {
26         this.start += this.interval;
27         this.end += this.interval;
28
29         if (maxlimitTime > 0) {
30             this.start = (this.start > maxlimitTime) ? maxlimitTime :
31                 ↪ this.start;
32             this.end = (this.end > maxlimitTime) ? maxlimitTime : this.end;
33         }
34     }
35 }

```

Código Fonte 25: Solução para o problema *UVa 11926 – Multitasking* (pt. 5).



## 2.6 Lista Encadeada

### 2.6.1 UVa 11988 – Broken Keyboard

Como é possível notar no Código Fonte 26 Inicialmente é feita a leitura da entrada de dados e criado uma lista encadeada para ser preenchida na ordem correta da saída. Iteramos cada char de cada linha do texto de entrada e é verificado se é a tecla "home" ou "end". Se o caracter for a tecla "end" ou um outro caracter qualquer, o texto continua sendo inserido na lista normalmente ao final. Caso uma tecla "home" é encontrada, uma flag para indicar ser adicionado ao início é marcada e até que encontre uma tecla end os seguintes caracteres é adicionado a partir da posição 0.

Ao final apenas imprimimos a lista encadeada na ordem que ela está. Na primeira submissão foi retornado TLE pelo UVa, pois a lista encadeada era iterada e imprimida caracter por caracter separadamente. Um jeito mais rápido que encontramos foi transformar a lista de caracteres em uma string através da classe StringBuilder, onde isso é realizado na função *output* no Código Fonte 27.

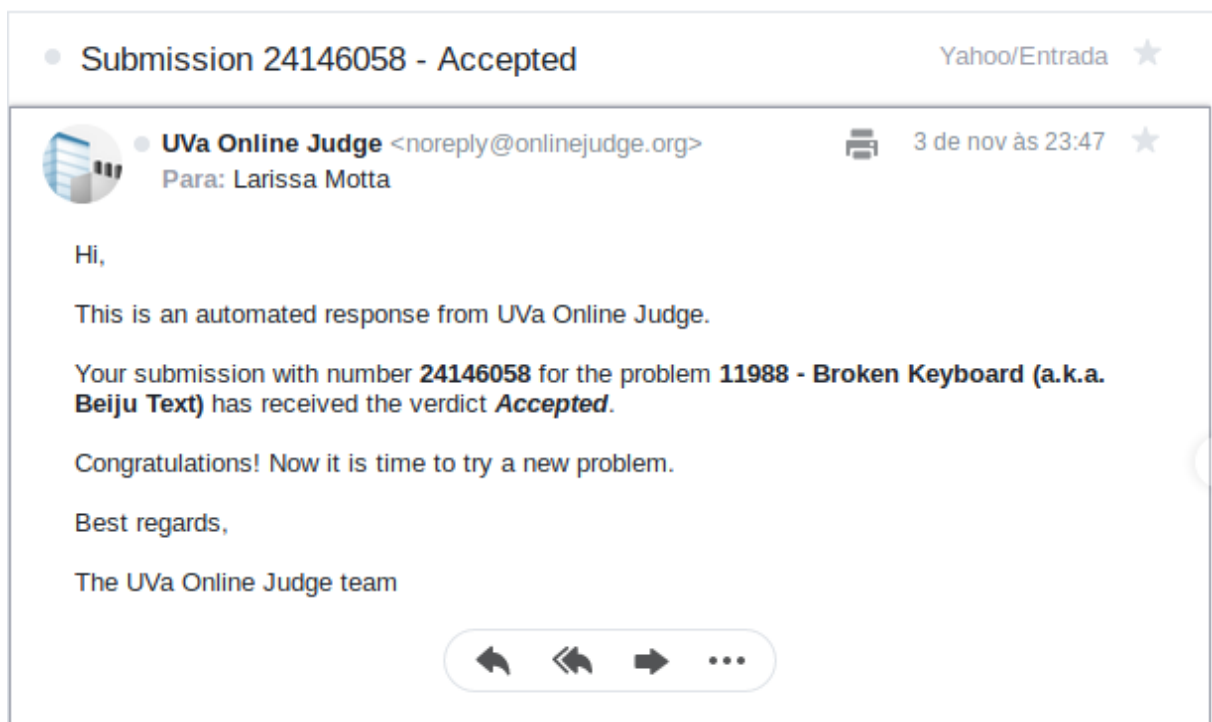


Figura 11: Email de aceitação da solução UVa 11988 – Broken Keyboard.

```

1  import java.util.Scanner;
2  import java.util.LinkedList;
3
4  public class Main {
5
6      private static Scanner scanner;
7
8      // Main function
9      public static void main(String[] args) {
10         scanner = new Scanner(System.in);
11         while (scanner.hasNext()) {
12             String line = scanner.nextLine();
13             LinkedList<Character> lst = new LinkedList<>();
14             // start by adding at the end of the list
15             boolean isFirst = false;
16             int indexFirst = 0;
17             // traverse the entire input sequence of the current line
18             for (int i = 0; i < line.length(); i++) {
19                 char current = line.charAt(i);
20                 if (current == '[') { // if '[' is home key, then add on first
21                     ↪ the next characters
22                     isFirst = true;
23                     indexFirst = 0;
24                 } else if (current == ']') // if ']' is end key, then add on
25                     ↪ last the next characters
26                     isFirst = false;
27                 else {
28                     if (isFirst) {
29                         /**
30                         * When the home key is pressed the first character is
31                         ↪ inserted and the
32                         * following characters right after it and before all
33                         ↪ already entered.
34                         */
35                         lst.add(indexFirst, current);
36                         indexFirst += 1;
37                     } else {
38                         indexFirst = 0;
39                         lst.addLast(current);
40                     }
41                 }
42             }
43             output(lst);
44         }
45     }
46 }

```

Código Fonte 26: Solução para o problema UVa 11988 – Broken Keyboard (pt. 1).

```
1    // print string output
2    private static void output(LinkedList<Character> list) {
3        StringBuilder sb = new StringBuilder();
4        for (char current : list)
5            sb.append(current);
6        System.out.println(sb.toString());
7    }
8 }
```

Código Fonte 27: Solução para o problema *UVa 11988 – Broken Keyboard* (pt. 2).

## 2.7 Pilhas

### 2.7.1 UVa 00514 – Rails

A solução consiste em preencher uma pilha de acordo com a ordem dos vagões que chegam do trem do trilho A, obtidos na entrada de dados, e logo após removê-lo da pilha para ser reorganizado para o trilho B como mostrado no Código Fonte 28. A remoção só pode acontecer se na pilha o número do vagão for igual a ordem, pois são empilhados de forma crescente.

O programa itera a ordem de chegada dos trens e a verificação para saída de que é possível ou não obter, a ordem da entrada é feita eliminando item a item da pilha. Ao final se a pilha estiver vazia a ordem desejada é possível de se obter, caso contrário não.

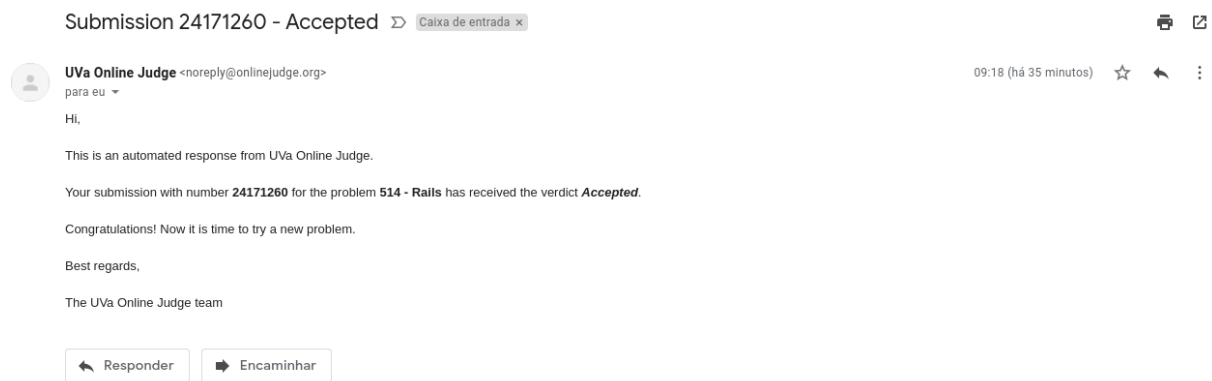


Figura 12: Email de aceitação da solução *UVa 00514 – Rails*.

```

1  import java.util.Scanner;
2  import java.util.Stack;
3
4  public class Main {
5      private static Scanner scanner;
6
7      // Main function
8      public static void main(String[] args) {
9          scanner = new Scanner(System.in);
10         int n = Integer.parseInt(scanner.nextLine());
11
12         // End program with n = 0
13         while (n > 0) {
14
15             // flag to break loop with coach is 0
16             boolean isEnd = false;
17             while (!isEnd) {
18
19                 String[] coaches = scanner.nextLine().split(" ");
20                 Stack<Integer> station = new Stack<>();
21
22                 int count = 0;
23
24                 // for each coach of train
25                 for (int i = 0; i < n; i++) {
26                     int coach = Integer.parseInt(coaches[i]);
27                     if (coach == 0) { // if coach = 0 is end of block
28                         isEnd = true;
29                         break;
30                     }
31
32                     while (count < n && coach != count) {
33                         if (station.size() > 0 && coach == station.peek())
34                             break;
35                         station.push(++count);
36                     }
37
38                     if (station.peek() == coach)
39                         station.pop();
40                 }
41
42                 if (!isEnd) {
43                     output(station);
44                 }
45             }
46             n = Integer.parseInt(scanner.nextLine());
47             System.out.printf("\n"); // fix the presentation error
48         }
49     }

```

Código Fonte 28: Solução para o problema UVa 00514 – Rails (pt.1).

```

1      // output print
2      private static void output(Stack<Integer> s) {
3          System.out.println(s.size() == 0 ? "Yes" : "No");
4      }
5  }

```

Código Fonte 29: Solução para o problema *UVa 00514 – Rails* (pt.2).

### 2.7.2 UVa 01062 – Containers

Foi criado um loop que percorre todos caracteres de cada linha de entrada para achar a pilha em que ele se encaixa. Caso não ache uma pilha é criada uma pilha nova inicialmente com o caracter que não foi possível se encaixar em nenhuma pilha. Como os navios são preenchidos em ordem alfabética, para achar a pilha que o caracter se encaixa foi simples. Apenas buscamos a primeira pilha que contenha uma carga de um navio que será preenchido primeiro, ou seja, a primeira pilha que tiver um caracter com peso maior do que o atual poderá recebe-lo porque o peso menor será preenchido primeiro, logo será desempilhado primeiro.

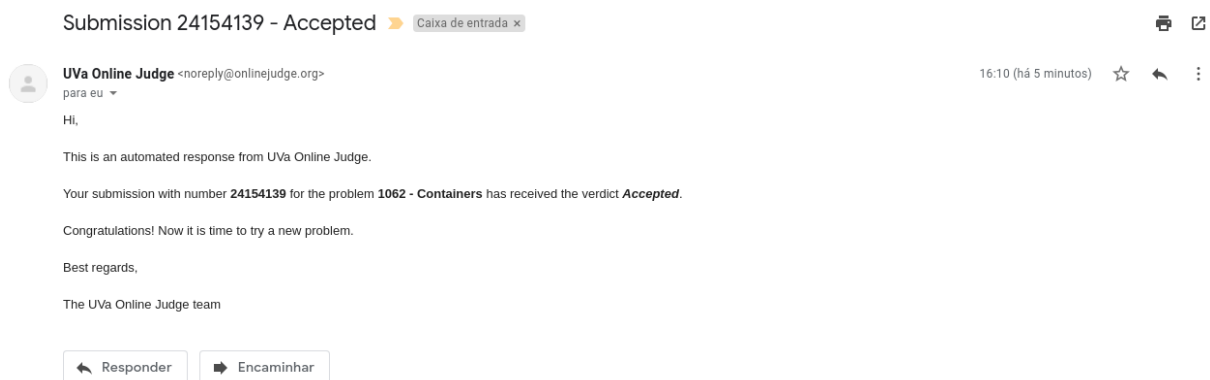


Figura 13: Email de aceitação da solução *UVa 01062 – Containers*.

```

1  import java.util.Scanner;
2  import java.util.Stack;
3  import java.util.Vector;
4
5  public class Main {
6
7      private static Scanner scanner;
8
9      // Main function
10     public static void main(String[] args) {
11         scanner = new Scanner(System.in);
12         String line = scanner.nextLine();
13         int caseCount = 1;
14
15         while (!isLineEnd(line)) {
16
17             // Vector of stacks (java vector get is O(1))
18             Vector<Stack<Character>> stacks = new Vector<>();
19
20             // for each characters in line find the first stack that can be
21             // → stacked
22             for (int i = 0; i < line.length(); i++) {
23                 char container = line.charAt(i);
24                 boolean isStacked = false;
25
26                 int j = 0;
27                 while (j < stacks.size() && !isStacked) {
28                     Stack<Character> currentStack = stacks.get(j);
29                     if (currentStack.peek() >= container) { // the ships is
30                         // → alphabetic order
31                         currentStack.push(container);
32                         isStacked = true;
33                     }
34                     j++;
35                 }
36
37                 if (!isStacked) { // couldn't find when create a new stack
38                     stacks.add(createStackWithElement(container));
39                 }
40
41             }
42
43             // output message
44             System.out.printf("Case %d: %d\n", caseCount, stacks.size());
45
46             line = scanner.nextLine();
47             caseCount++;
48         }
49     }
50 }

```

Código Fonte 30: Solução para o problema UVa 01062 – Containers (pt. 1).

```

1      // create a character stack and push the first element
2      public static Stack<Character> createStackWithElement(char element) {
3          Stack<Character> stack = new Stack<>();
4          stack.push(element);
5          return stack;
6      }
7
8      // verify if line is end token
9      public static boolean isLineEnd(String line) {
10         return line.compareTo("end") == 0;
11     }
12 }

```

Código Fonte 31: Solução para o problema *UVa 01062 – Containers* (pt.2).



## 2.8 Filas

### 2.8.1 UVa 10172 – The Lonesome Cargo

Começamos o desenvolvimento com o pensamento de criar um array para representar a plataforma B, porém para facilitar o uso de índices de  $1 \leq x \leq n$ , optamos por usar um HashMap que não deve ser tão pior assim neste caso e facilitaria um pouco para escrever a solução de acordo com o enunciado, como se pode ver na linha 15 do Código Fonte 32.

Após a entrada de dados inicial, que representa a quantidade total de casos de teste, obtemos os números  $n$ ,  $s$  e  $q$ , número de estações no anel, capacidade da transportadora e número máximo de cargas da fila da plataforma  $b$ , respectivamente. Então, instanciamos uma fila para cada estação dentro do map e a preenchemos com seus destinos. Com essa estrutura, caminhamos de forma circular na plataforma  $b$  e contamos o tempo de acordo com os destinos que são alcançados. Para ganhar um pouco mais de tempo e evitar TLE também é utilizado a classe StringBuilder para criar uma string de saída e realizar apenas uma operação de print ao final do programa.

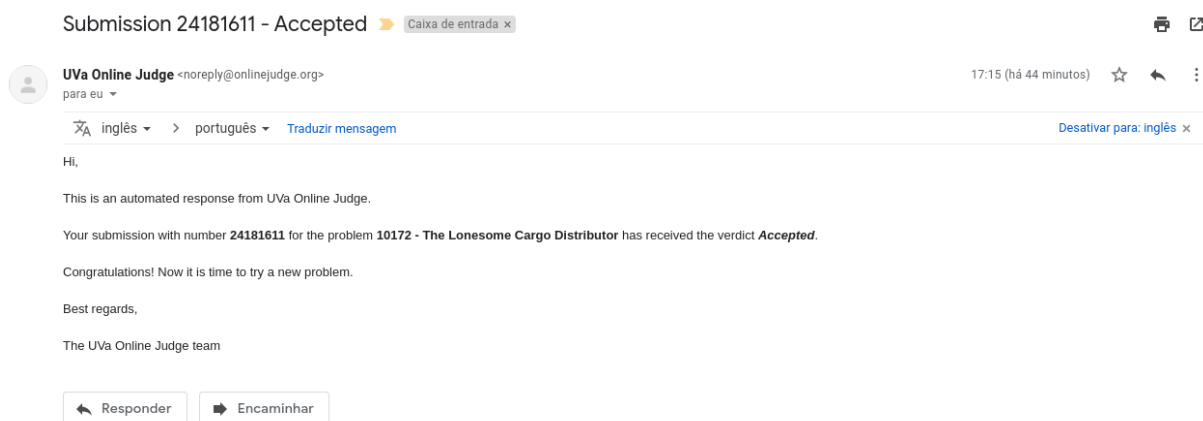


Figura 14: Email de aceitação da solução *UVa 10172 – The Lonesome Cargo*.

```

1  import java.util.HashMap;
2  import java.util.LinkedList;
3  import java.util.Queue;
4  import java.util.Scanner;
5  import java.util.Stack;
6
7  class Main {
8
9      private static Scanner scanner;
10     private static int n;
11
12     public static void main(String[] args) {
13         /**
14          * to start index with 1 I'm using HashMap
15          */
16         HashMap<Integer, Queue<Integer>> b = new HashMap<Integer,
17             ↪ Queue<Integer>>();
18         Queue<Integer> currentQueue;
19
20         scanner = new Scanner(System.in);
21
22         // buffer output to print one time
23         StringBuilder output = new StringBuilder();
24
25         // total cases of set
26         int totalSet = scanner.nextInt();
27
28         // run program with "totalSet" number of output(s)
29         for (int x = 0; x < totalSet; x++) {
30             n = scanner.nextInt(); // number of stations in the ring
31             int s = scanner.nextInt(); // capacity of your cargo carrier
32             int q = scanner.nextInt(); // maximum number of cargoes the queue
33             ↪ in platform "b" can accommodate
34
35             /**
36              * Create the queues in map and put input
37              */
38             for (int i = 1; i <= n; i++) {
39                 int qi = scanner.nextInt();
40                 currentQueue = new LinkedList<>();
41                 b.put(i, currentQueue);
42                 for (int j = 0; j < qi; j++) {
43                     currentQueue.add(scanner.nextInt());
44                 }
45             }
46         }
47     }
48 }

```

Código Fonte 32: Solução para o problema UVa 10172 – The Lonesome Cargo (pt. 1).

```

1  int minutes = 0;
2      Queue<Integer> platform;
3      Stack<Integer> carrier = new Stack<Integer>();
4      int cargo;
5      int i = 1;
6      while (!isEmpty(b) || !carrier.empty()) {
7          minutes += 2;
8          platform = b.get(i);
9
10         /**
11         ↪ unload
12         */
13         while (!carrier.empty() && (carrier.peek() == i ||
14         ↪ platform.size() < q)) {
15             cargo = carrier.pop();
16             if (cargo != i) {
17                 platform.add(cargo);
18             }
19             minutes++;
20         }
21
22         /**
23         ↪ Load carrier if "b" has cargos and carrier has free space
24         */
25         while (platform.size() > 0 && carrier.size() < s) {
26             carrier.push(platform.poll());
27             minutes++;
28         }
29
30         // circular motion
31         i = i % n + 1;
32     }
33     output.append(Integer.toString(minutes == 0 ? minutes : minutes -
34     ↪ 2));
35     output.append("\n");
36 }
37
38 System.out.print(output);

```

Código Fonte 33: Solução para o problema UVa 10172 – The Lonesome Cargo (pt.2).

```

1      /**
2       * return true if platform b is empty
3       *
4       * @param b
5       * @return
6       */
7      public static boolean isEmpty(HashMap<Integer, Queue<Integer>> b) {
8          for (int i = 1; i <= n; i++) {
9              if (b.get(i).size() > 0) {
10                 return false;
11             }
12         }
13         return true;
14     }
15 }

```

Código Fonte 34: Solução para o problema *UVa 10172 – The Lonesome Cargo* (pt.3).

## 2.8.2 UVa 10901 – Ferry Loading III

Inicialmente, após as entradas de dados "n", "t" e "m" é instanciado duas filas de array de inteiros, estes arrays agem como um par ordenado com o tempo do carro e a ordem que ele foi inserido no programa (para ajudar na saída do programa). Essas filas são preenchida com os dados seguintes que são dos carros: qual lado o carro chegou e em qual tempo (em minutos começando de 0). Dessa forma conseguimos uma fila para cada lado e com o tempo que cada um chegou na extremidade da balsa. Após a fila preenchida a função "getOutput" faz a coleta das filas alternando entre as posições e inserindo no array de saída na posição correta de impressão o tempo. O tempo é calculado de acordo com o tempo corrente atual + o tempo de duração de viagem. Assim que a balsa muda de lado a variável que conta o tempo é acumulada com o valor de duração da viagem. No final com o array de saída preenchido basta percorrê-lo, pois a informação do par ordenado já cuidou de inserir todas as saídas na ordem correta.

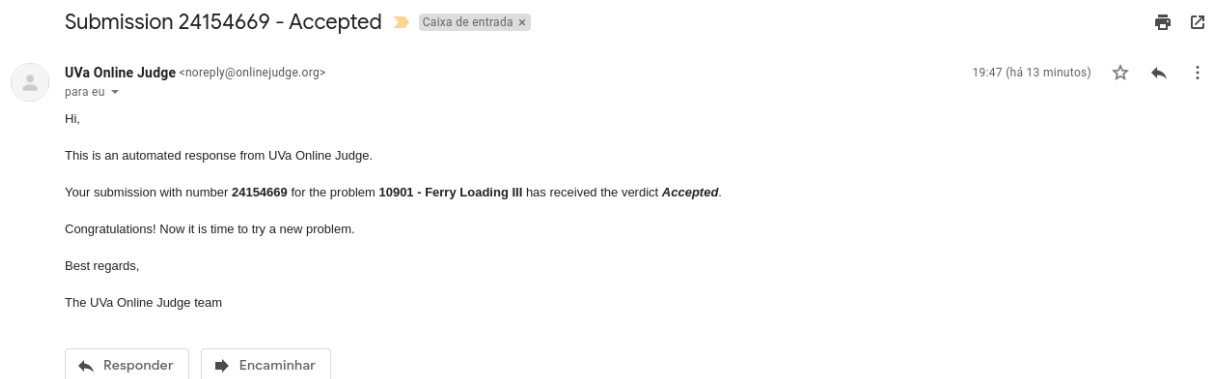


Figura 15: Email de aceitação da solução *UVa 10901 – Ferry Loading III*.

```

1  import java.util.Arrays;
2  import java.util.LinkedList;
3  import java.util.List;
4  import java.util.Queue;
5  import java.util.stream.Collectors;
6  import java.util.Scanner;
7
8  public class Main {
9
10     private final static String LEFT = "left";
11     private final static String RIGHT = "right";
12     private static Scanner scanner;
13
14     // Main function
15     public static void main(String[] args) {
16         scanner = new Scanner(System.in);
17         int casesLength = Integer.parseInt(scanner.nextLine());
18         for (int i = 0; i < casesLength; i++) {
19             List<Integer> firstLine = getLineInt(scanner.nextLine());
20             int n = firstLine.get(0);
21             int t = firstLine.get(1);
22             int m = firstLine.get(2);
23
24             Queue<int []> l = new LinkedList<>();
25             Queue<int []> r = new LinkedList<>();
26
27             fillQueues(m, l, r);
28
29             for (int time : getOutput(n, t, m, l, r)) {
30                 System.out.println(time);
31             }
32             if (i + 1 < casesLength) {
33                 System.out.printf("\n");
34             }
35         }
36     }

```

Código Fonte 35: Solução para o problema *UVa 10901 – Ferry Loading III* (pt.1).

```

1  // Fill left and right queue
2  public static void fillQueues(int m, Queue<int[]> leftQueue, Queue<int[]>
   ↪ rightQueue) {
3      for (int i = 0; i < m; i++) {
4          String[] buf = Main.scanner.nextLine().split(" ");
5          int carTime = Integer.parseInt(buf[0]);
6          String bank = buf[1];
7          int[] car = new int[2];
8          car[0] = carTime;
9          car[1] = i;
10         if (isEquals(bank, RIGHT)) {
11             rightQueue.add(car);
12         } else {
13             leftQueue.add(car);
14         }
15     }
16 }
17
18 // if string a is equals string b apllying trim
19 public static boolean isEquals(String a, String b) {
20     return a.trim().equals(b.trim());
21 }
22
23 // toggle bank state
24 public static String toggleBank(String current) {
25     return current == LEFT ? RIGHT : LEFT;
26 }
27
28 // get scond line and map to Integer
29 public static List<Integer> getLineInt(String line) {
30     List<String> res = Arrays.asList(line.split(" "));
31     return res.stream().map(e ->
   ↪ Integer.parseInt(e)).collect(Collectors.toList());
32 }

```

Código Fonte 36: Solução para o problema *UVa 10901 – Ferry Loading III* (pt.2).

```

1 // calculate time of two queues and returns a output array
2 public static int[] getOutput(int n, int t, int m, Queue<int[]> left,
3   ↪ Queue<int[]> right) {
4     int[] output = new int[m];
5     int time = 0;
6     String currentBank = LEFT;
7     Queue<int[]> aux1, aux2;
8
9     while (!left.isEmpty() || !right.isEmpty()) {
10
11         if (isEqual(currentBank, LEFT)) {
12             aux1 = left;
13             aux2 = right;
14         } else {
15             aux1 = right;
16             aux2 = left;
17         }
18
19         int enqueuedCount = 0;
20
21         while (enqueuedCount < n && !aux1.isEmpty() && aux1.peek()[0] <=
22   ↪ time) {
23             int[] car = aux1.remove();
24             output[car[1]] = time + t;
25             enqueuedCount++;
26         }
27
28         if (enqueuedCount != 0 || !aux2.isEmpty() && aux2.peek()[0] <=
29   ↪ time) {
30             time += t;
31             currentBank = toggleBank(currentBank);
32         } else if (aux1.isEmpty() || !aux2.isEmpty() && aux2.peek()[0] <
33   ↪ aux1.peek()[0]) {
34             time = aux2.peek()[0] + t;
35             currentBank = toggleBank(currentBank);
36         } else {
37             time = aux1.peek()[0];
38         }
39     }
40     return output;
41 }

```

Código Fonte 37: Solução para o problema *UVa 10901 – Ferry Loading III* (pt.3).



## 2.9 Árvore Binária de Pesquisa (balanceada)

### 2.9.1 UVa 00939 – Genes

Inicialmente no Código Fonte 38 é lido a quantidade de linhas de dados a serem processados e inseridos em uma *TreeMap* onde a chave é uma string, representando o nome da pessoa, e o valor um objeto da classe *Person*, onde possui as informações sobre o gene.

Ao iniciar a leitura desses dados é separado as duas substrings (linha 18) utilizando o método *split*, onde a primeira necessariamente é um nome de uma pessoa e a segunda pode ser um nome ou referência ao gene. Caso a segunda string seja um gene, é verificado se existe uma chave com a primeira string na *TreeMap*, caso exista, é atualizado seu gene, caso não é inserida na *TreeMap*. Caso a segunda substring for um nome, indicando assim que é um Filho e a primeira string o Pai, verifica se possui as duas chaves na *TreeMap*, caso não possua alguma é inserida, em seguida é feito a ligação de pai ao filho.

Ao final é feito a impressão dos resultados onde a *TreeMap* é percorrida e faz-se a chamada do método *getGene()*, onde de forma recursiva consegue identificar se a pessoa não possui e se possui qual o tipo de gene.

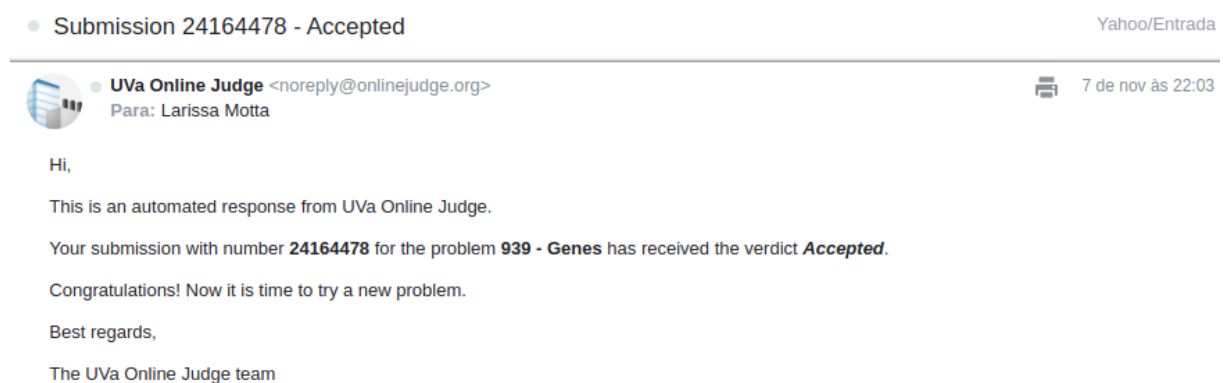


Figura 16: Email de aceitação da solução *UVa 00939 – Genes*.

```

1  import java.util.ArrayList;
2  import java.util.Scanner;
3  import java.util.SortedMap;
4  import java.util.TreeMap;
5
6  public class Main {
7      private static final Scanner SC = new Scanner(System.in);
8      private static final SortedMap<String, Person> RESULT = new TreeMap();
9      private static final String DOMINANT = "dominant";
10     private static final String RECESSIVE = "recessive";
11     private static final String NON_EXISTENT = "non-existent";
12
13     public static void main(String[] args) {
14         //First reads the line that represents the number of lines of the
15         ↪ dataset
16         int n = SC.nextInt();
17         ignoreLines(1);
18         for (int i = 0; i < n; i++) {
19             String[] tokens_line = SC.nextLine().split(" ");
20             String personName = tokens_line[0];
21
22             // Check if the token of second positions is a person or a gene
23             if (isGene(tokens_line[1])) {
24                 if (RESULT.containsKey(personName)) {
25                     RESULT.get(personName).gene = tokens_line[1];
26                 } else {
27                     RESULT.put(personName, new Person(personName,
28                     ↪ tokens_line[1]));
29                 }
30             } else {
31                 String parent = personName;
32                 personName = tokens_line[1];
33
34                 if (!RESULT.containsKey(parent)) {
35                     RESULT.put(parent, new Person(parent));
36                 }
37                 if (!RESULT.containsKey(personName)) {
38                     RESULT.put(personName, new Person(personName));
39                 }
40                 RESULT.get(personName).parents.add(RESULT.get(parent));
41             }
42         }
43     }

```

Código Fonte 38: Solução para o problema *UVa 00939 – Genes* (pt. 1).

```

1      // Prints out the RESULT map values sorted by name
2      public static void output() {
3          RESULT.keySet().forEach((key) -> {
4              System.out.println(key + " " + RESULT.get(key).getGene());
5          });
6      }
7
8      // Verify if a string is a gene
9      public static boolean isGene(String value) {
10         return value.equals(DOMINANT) || value.equals(RECESSIVE) ||
11             ↪ value.equals(NON_EXISTENT);
12     }
13
14     // Ignore the number of lines in terminal
15     public static void ignoreLines(int numberOfLines) {
16         for (int i = 0; i < numberOfLines; i++) {
17             SC.nextLine();
18         }
19     }

```

Código Fonte 39: Solução para o problema *UVa 00939 – Genes* (pt. 2).

```

1  final class Person {
2      private static final String DOMINANT = "dominant";
3      private static final String RECESSIVE = "recessive";
4      private static final String NON_EXISTENT = "non-existent";
5      public String name;
6      public String gene;
7      public ArrayList<Person> parents = new ArrayList();
8
9      public Person(String name, String gene) {
10         this.name = name;
11         this.gene = gene;
12     }
13
14     public Person(String name) {
15         this.name = name;
16         this.gene = null;
17     }
18
19     public String getGene() {
20         if (this.gene == null) {
21             this.gene = identifyGene(this.parents.get(0).getGene(),
22                                     ↪ this.parents.get(1).getGene());
23         }
24         return this.gene;
25     }
26
27     // Identifies wich gene the person has
28     private static String identifyGene(String parentGene1, String parentGene2)
29     ↪ {
30         // If they are equal means that the parents have the same gene
31         if (parentGene2.equals(parentGene1)) {
32             return parentGene1;
33         }
34
35         // Verify if the parents have a gene
36         boolean dominant = parentGene1.equals(DOMINANT) ||
37             ↪ parentGene2.equals(DOMINANT);
38         boolean recessive = parentGene1.equals(RECESSIVE) ||
39             ↪ parentGene2.equals(RECESSIVE);
40
41         // Check the correct gene of the current person based on his parents
42         if (dominant && recessive) {
43             return DOMINANT;
44         }
45         if (dominant && !recessive) {
46             return RECESSIVE;
47         }
48         return NON_EXISTENT;
49     }
50 }

```

Código Fonte 40: Solução para o problema UVa 00939 – Genes (pt. 3).

## 2.9.2 UVa 10132 – File Fragmentation

Como mostrado no Código Fonte 41 é lido a primeira linha da entrada que indica a quantidade de blocos a serem lidos. Para cada bloco é guardado os fragmentos em uma *ArrayList*, mostrado em Código Fonte 42 na função *readBlock()*, em seguida é calculado, a partir do menor e maior fragmento, o tamanho do arquivo final.

Após isso verifica-se os possíveis resultados, onde os que possuem o tamanho esperado é armazenado em uma *TreeMap* junto com a quantidade de vezes que aparece. A saída é dada passando pela *TreeMap* e impresso a chave, conteúdo final dos arquivos, caso apareça mais que a quantidade de fragmentos dividida por 2 (quantidade de arquivos que se fragmentaram).

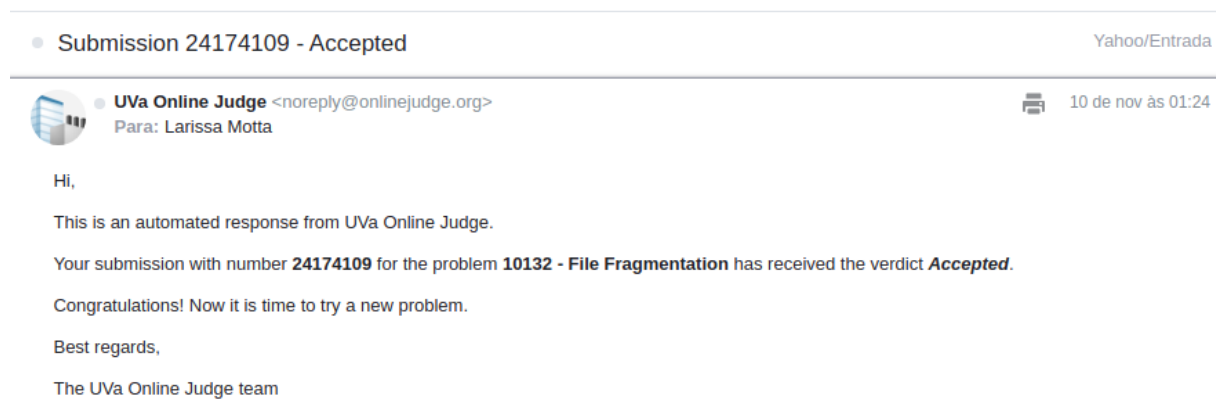


Figura 17: Email de aceitação da solução *UVa 10132 – File Fragmentation*.

```

1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.util.ArrayList;
5  import java.util.SortedMap;
6  import java.util.TreeMap;
7
8  public class Main {
9
10     private static final BufferedReader SC = new BufferedReader(new
        ↳ InputStreamReader(System.in));
11
12     public static void main(String[] args) throws IOException {
13
14         //First reads the line that represents the number of cases of the
        ↳ dataset
15         int n = Integer.parseInt(SC.readLine());
16         ignoreLines(1);
17
18         for (int i = 0; i < n; i++) {
19             //Add the fragments in memory list structure
20             ArrayList<String> listFragments = readBlock();
21
22             //Get the length of final file
23             int lengthOfFinalFile = getLengthOfFinalFile(listFragments);
24
25             //Get possibles solutions
26             SortedMap<String, Integer> possibleResults =
                ↳ getPossiblesSolutions(listFragments, lengthOfFinalFile);
27
28             //Print solution
29             for(String key : possibleResults.keySet()) {
30                 if (possibleResults.get(key) >= (listFragments.size() / 2)) {
31                     System.out.println(key);
32                     break;
33                 }
34             }
35
36             //Print a blank line
37             if (i != n-1) {
38                 System.out.println();
39             }
40         }
41
42         SC.close();
43         System.exit(0);
44     }

```

Código Fonte 41: Solução para o problema *UVa 10132 – File Fragmentation* (pt.1).

```

1  // Ignore the number of lines in terminal
2  public static void ignoreLines(int numberOfLines) throws IOException {
3      for (int i = 0; i < numberOfLines; i++) {
4          SC.readLine();
5      }
6  }
7
8  //Returns the fragments in memory list structure
9  public static ArrayList<String> readBlock() throws IOException {
10     ArrayList<String> listFragments = new ArrayList();
11
12     while (true) {
13         String fragment = SC.readLine();
14
15         if(fragment == null || fragment.isEmpty()){
16             break;
17         }
18         listFragments.add(fragment);
19     }
20     return listFragments;
21 }
22
23 //Returns the length of final file based on the min and max fragment
24 public static int getLengthOfFinalFile(ArrayList<String> listFragments) {
25     int min = listFragments.get(0).length();
26     int max = min;
27
28     for (int i = 1; i < listFragments.size(); i++) {
29         int length = listFragments.get(i).length();
30
31         if (length > max) {
32             max = length;
33         }
34
35         if (min > length) {
36             min = length;
37         }
38     }
39     return min + max;
40 }

```

Código Fonte 42: Solução para o problema *UVa 10132 – File Fragmentation* (pt.2).

```

1      //Returns a TreeMap with the possibles results
2      public static SortedMap<String, Integer>
3      ↪      getPossiblesSolutions(ArrayList<String> listFragments, int
4      ↪      lengthOfFile) {
5          SortedMap<String, Integer> possibleResults = new TreeMap();
6
7          for (int j = 0; j < listFragments.size(); j++) {
8
9              for (int k = 0; k < listFragments.size(); k++) {
10
11                  if (k != j) {
12                      String possibility = listFragments.get(j) +
13                      ↪      listFragments.get(k);
14
15                      //Verify if the possibility has the length of final file
16                      if (possibility.length() == lengthOfFile) {
17                          Integer count = possibleResults.get(possibility);
18
19                          if (count != null) {
20                              possibleResults.put(possibility, count + 1);
21                          } else {
22                              possibleResults.put(possibility, 1);
23                          }
24                      }
25                  }
26              }
27          }

```

Código Fonte 43: Solução para o problema *UVa 10132 – File Fragmentation* (pt.3).



## 2.10 Conjuntos

### 2.10.1 UVa 00978 – Lemmings Battle

No Código Fonte 44 está presente a função *main* que contém todo o fluxo principal de execução do jogo. Inicialmente, na linha 10, é lido o número de casos que ocorrerá no jogo. Após isso para cada caso ocorre uma partida do jogo, ou seja, caso tenha 3 casos são três partidas diferentes.

Entre as linhas 14 e 17 são lidas as informações de número de campos de batalha, quantidade de soldados da raça *green* e *blue* respectivamente. Após isso, nas linhas 20 e 23 é chamada a função responsável por criar as instâncias dos soldados da classe *Lemming*, que são armazenados dentro da estrutura de dados chamada fila de prioridade (*PriorityQueue*). Este tipo de estrutura armazena os dados baseado na prioridade, onde no problema proposto é o poder (*power*), dado que quanto maior o poder maior a prioridade do objeto na fila, como é mostrado no Código Fonte 48, onde a classe *Lemming* implementa a interface *Comparable* sendo necessário implementar o método *compareTo* para ser utilizado como comparador no momento de adicionar o objeto na fila de prioridade.

Logo após é chamada a função *battle*, responsável por emular a batalha entre os exércitos das duas raças distintas até que um dos exércitos ou nenhum deles possuem soldados, onde estes estão presentes na fila de prioridade. No Código Fonte 46 está a implementação da função de batalha, onde em cada campo de batalha ocorre um duelo entre um soldado de cada raça. Nesta batalha ganha o soldado que possuir maior poder (*power*), matando seu adversário. Este tem seu poder subtraído pelo poder de seu oponente e armazenado em uma lista para a próxima rodada, ou seja, não podendo duelar duas vezes seguidas. Após isso todos os duelistas sobreviventes são adicionados novamente, na estrutura de fila de prioridade, no seu correspondente exército para que lute em futuros duelos.

Por fim Código Fonte 47 é impresso o ganhador da guerra, com o poder dos seus respectivos soldados vivos em ordem decrescente, ou que ambos perderam a guerra, pois não há nenhum soldado restante.

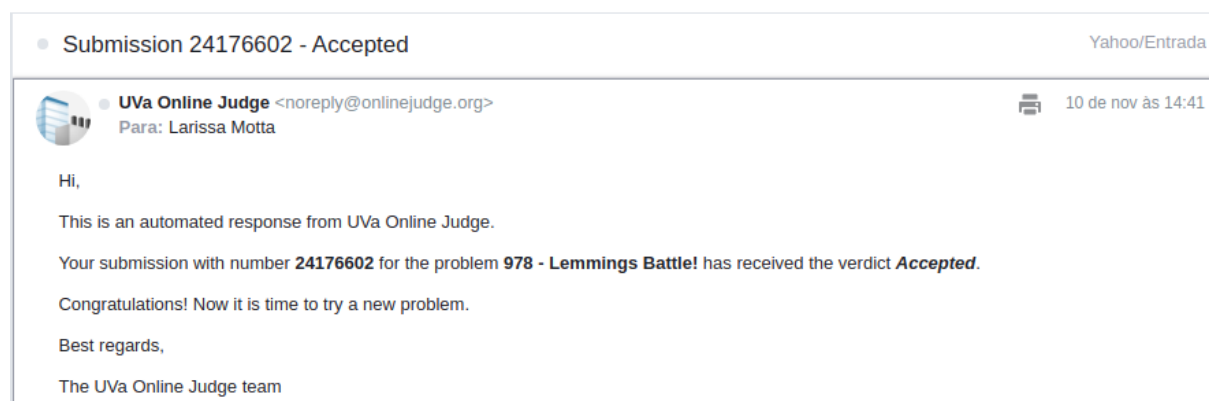


Figura 18: Email de aceitação da solução UVa 00978 – Lemmings Battle.

```

1  import java.util.LinkedList;
2  import java.util.PriorityQueue;
3  import java.util.Scanner;
4
5  public class Main {
6
7      private static final Scanner SC = new Scanner(System.in);
8
9      public static void main(String[] args) {
10         int nCases = SC.nextInt();
11         ignoreLines(1);
12
13         for (int i = 1; i <= nCases; i++) {
14             String[] tokensInfoGame = SC.nextLine().split(" ");
15             int nBattlefield = Integer.parseInt(tokensInfoGame[0]);
16             int nLemmingGreen = Integer.parseInt(tokensInfoGame[1]);
17             int nLemmingBlue = Integer.parseInt(tokensInfoGame[2]);
18
19             // Creating a queue to each Lemming Green Army soldier
20             PriorityQueue<Lemming> greenArmy =
21                 ↪ createLemmingArmy(nLemmingGreen);
22
23             // Creating a queue to each Lemming Blue Army soldier
24             PriorityQueue<Lemming> blueArmy = createLemmingArmy(nLemmingBlue);
25
26             /* Perfomances a battle until the end, wich means just one or any
27             * army has elements
28             */
29             battle(greenArmy, blueArmy, nBattlefield);
30
31             // Show the winner of the current war case
32             showWinner(greenArmy, blueArmy);
33
34             if (i < nCases) {
35                 System.out.println();
36             }
37         }
38     }

```

Código Fonte 44: Solução para o problema *UVa 00978 – Lemmings Battle* (pt.1).

```

1  // Ignore the number of lines in terminal
2  private static void ignoreLines(int numberOfLines) {
3      for (int i = 0; i < numberOfLines; i++) {
4          SC.nextLine();
5      }
6  }
7
8  // Creates a Lemming Army
9  private static PriorityQueue<Lemming> createLemmingArmy(int qntLemmingArmy) {
10     PriorityQueue<Lemming> army = new PriorityQueue();
11
12     while (qntLemmingArmy-- > 0) {
13         int currentPower = SC.nextInt();
14         ignoreLines(1);
15
16         /* Creates a Leeming and add into a queue based in his priority,
17         * wich is the greater power
18         */
19         army.add(new Lemming(currentPower));
20     }
21
22     return army;
23 }

```

Código Fonte 45: Solução para o problema *UVa 00978 – Lemmings Battle* (pt.2).

```

1  //
2  private static void battle(PriorityQueue<Lemming> greenArmy,
    ↪ PriorityQueue<Lemming> blueArmy, int nBattles) {
3      if (nBattles <= 0) {
4          return;
5      }
6
7      // Fight until one of them or both are dead
8      while (!greenArmy.isEmpty() && !blueArmy.isEmpty()) {
9          int nBattlesCurrent = nBattles;
10         TreeSet<Lemming> greenBattlefieldArmy = new TreeSet();
11         TreeSet<Lemming> blueBattlefieldArmy = new TreeSet();
12
13         while (nBattlesCurrent-- > 0) {
14             // Retrieves the next soldiers for the battle
15             Lemming greenSoldier = greenArmy.remove();
16             Lemming blueSoldier = blueArmy.remove();
17
18             // Fight with each other and one of them or both will be in
    ↪ dead state
19             greenSoldier.fight(blueSoldier);
20
21             // Adds the green soldier with the difference power
22             if (!greenSoldier.isDead) {
23                 greenBattlefieldArmy.add(greenSoldier);
24             }
25             // Adds the blue soldier with the difference power
26             if (!blueSoldier.isDead) {
27                 blueBattlefieldArmy.add(blueSoldier);
28             }
29
30             // Check if at least one of the armies is all down while
    ↪ battlefield exists
31             if (greenArmy.isEmpty() || blueArmy.isEmpty()) {
32                 break;
33             }
34         }
35
36         // Adds the soldiers in the real army after the battlefield ends
37         if (!greenBattlefieldArmy.isEmpty()) {
38             greenArmy.addAll(greenBattlefieldArmy);
39         }
40
41         if (!blueBattlefieldArmy.isEmpty()) {
42             blueArmy.addAll(blueBattlefieldArmy);
43         }
44     }
45 }

```

Código Fonte 46: Solução para o problema UVa 00978 – Lemmings Battle (pt.3).

```

1  private static void showWinner(PriorityQueue<Lemming> greenArmy,
   ↪  PriorityQueue<Lemming> blueArmy) {
2      if (greenArmy.isEmpty() && blueArmy.isEmpty()) {
3          System.out.println("green and blue died");
4          return;
5      }
6
7      if (blueArmy.isEmpty()) {
8          System.out.println("green wins");
9
10         while (!greenArmy.isEmpty()) {
11             System.out.println(greenArmy.remove().toString());
12         }
13         return;
14     }
15
16     if (greenArmy.isEmpty()) {
17         System.out.println("blue wins");
18
19         while (!blueArmy.isEmpty()) {
20             System.out.println(blueArmy.remove().toString());
21         }
22     }
23 }
24 }

```

Código Fonte 47: Solução para o problema *UVa 00978 – Lemmings Battle* (pt.4).

```

1  final class Lemming implements Comparable<Lemming> {
2
3      int power;
4      boolean isDead;
5
6      public Lemming(int power) {
7          this.power = power;
8      }
9
10     public void fight(Lemming enemy) {
11         int myPower = this.power - enemy.power;
12         int enemyPower = enemy.power - this.power;
13
14         if (myPower < 0) {
15             this.isDead = true;
16         } else if (myPower > 0) {
17             enemy.isDead = true;
18         } else {
19             this.isDead = enemy.isDead = true;
20         }
21
22         this.power = myPower;
23         enemy.power = enemyPower;
24     }
25
26     @Override
27     public int compareTo(Lemming l) {
28         if (this.power > l.power) {
29             return -1;
30         }
31
32         return 1;
33     }
34
35     @Override
36     public String toString() {
37         return "" + this.power;
38     }
39 }

```

Código Fonte 48: Solução para o problema *UVa 00978 – Lemmings Battle* (pt.5).

### 2.10.2 UVa 11849 – CD

Através do Código Fonte 49, inicialmente um loop infinito foi definido (linha 16), pois não temos a quantidade total de testes. Assim que encontramos N e M iguais a abortamos o loop, pois acabaram os casos de teste (linhas 20 e 21).

Para a solução do problema foi sugerido o TreeSet pelo professor, porém tentamos apesar da resposta correta o UVa sempre dava o veredito *TLE - Time Limited Exceeded*. Acharmos então que poderia ser a impressão e trabalhamos com StringBuilder para realizar apenas um print ao final do programa, porém ainda sem sucesso. Logo após algumas tentativas de otimização encontramos um artigo que explicava os diferentes "Set"s do Java e vimos que a complexidade dos métodos "add" e "contains" é maior no TreeSet. Então resolvemos utilizar o HashSet e o TLE escapou por pouco (2.920s/3.000s). A solução em si se baseou em criar um HashSet para colocar os CDs do Jack e logo após fazer a interseção com os CDs do Jill, verificando CD por CD se continha no conjunto do Jack. O número de interseções foi contado e o resultado é a saída do caso de teste (a partir da linha 28).

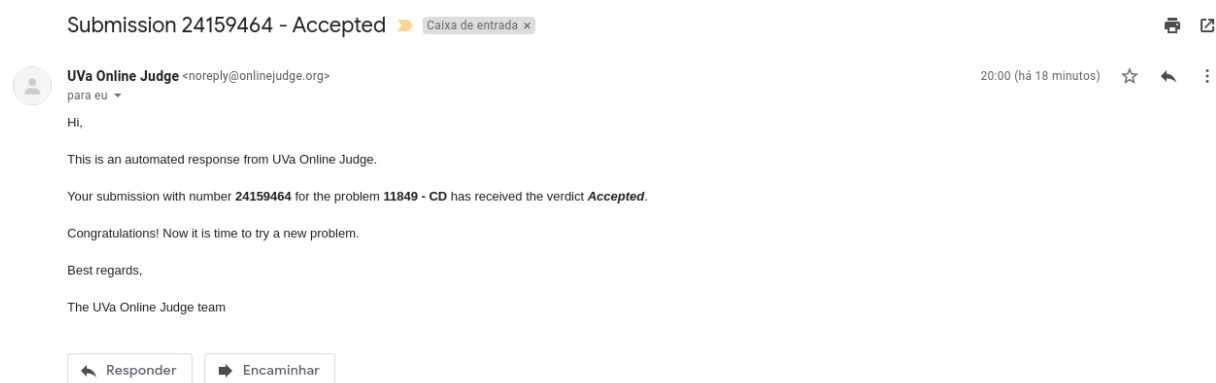


Figura 19: Email de aceitação da solução *UVa 11849 – CD*.

```

1  import java.util.Scanner;
2  import java.util.HashSet;
3
4  public class Main {
5
6      private static Scanner scanner;
7
8      /**
9       * Main function
10      *
11      * @param args
12      */
13     public static void main(String[] args) {
14         scanner = new Scanner(System.in);
15         StringBuilder output = new StringBuilder();
16         while (true) {
17             int n = scanner.nextInt();
18             int m = scanner.nextInt();
19
20             if (n == 0 && m == 0) {
21                 break;
22             }
23
24             HashSet<Integer> jack = new HashSet<>();
25             int numberOfCds = 0;
26
27             /**
28              * Save all Jack CDs in HashSet
29              */
30             for (int i = 0; i < n; i++) {
31                 jack.add(scanner.nextInt());
32             }
33
34             /**
35              * For each Jill CD count the intersection with Jack CDs
36              */
37             for (int i = 0; i < m; i++) {
38                 if (jack.contains(scanner.nextInt())) { // Jill cd intersec
39                     ↪ Jack
40                     numberOfCds++;
41                 }
42             }
43             output.append(numberOfCds).append("\n");
44             System.out.print(output);
45         }
46     }

```

Código Fonte 49: Solução para o problema UVa 11849 – CD.