

Advent of code 2022

Sergio Jiménez Roncero



diciembre, 2022.

<https://github.com/SergioJimenezR/AdventOfCode2022>

<https://www.linkedin.com/in/sergiojimenezr/>

[--- Day 1: Calorie Counting --- 2](#_Toc120907388)

[Parte 1 2](#_Toc120907389)

[Enunciado 2](#_Toc120907390)

[Código 3](#_Toc120907391)

[Utils.java 3](#_Toc120907392)

[MainParte1.java 3](#_Toc120907393)

[Argumentación 4](#_Toc120907394)

[Resultado 4](#_Toc120907395)

[Parte 2 4](#_Toc120907396)

[Enunciado 4](#_Toc120907397)

[Código 4](#_Toc120907398)

[MainParte2.java 4](#_Toc120907399)

[Argumentación 5](#_Toc120907400)

[Resultado 5](#_Toc120907401)

[--- Day 2: Rock Paper Scissors --- 5](#_Toc120907402)

[Parte 1 5](#_Toc120907403)

[Enunciado 5](#_Toc120907404)

[Código 6](#_Toc120907405)

[MainParte1.java 6](#_Toc120907406)

[Argumentación 7](#_Toc120907407)

[Resultado 7](#_Toc120907408)

[Parte 2 7](#_Toc120907409)

[Enunciado 7](#_Toc120907410)

[Código 7](#_Toc120907411)

[MainParte2.java 7](#_Toc120907412)

[Argumentación 8](#_Toc120907413)

[Resultado 8](#_Toc120907414)

[Día 3 8](#_Toc120907415)

[Parte 1 8](#_Toc120907416)

[Enunciado 8](#_Toc120907417)

[Código 8](#_Toc120907418)

[MainParte1.java 8](#_Toc120907419)

[Argumentación 8](#_Toc120907420)

[Resultado 8](#_Toc120907421)

[Parte 2 8](#_Toc120907422)

[Enunciado 8](#_Toc120907423)

[Código 9](#_Toc120907424)

[MainParte2.java 9](#_Toc120907425)

[Argumentación 9](#_Toc120907426)

[Resultado 9](#_Toc120907427)

[Día N 9](#_Toc120907428)

[Parte 1 9](#_Toc120907429)

[Enunciado 9](#_Toc120907430)

[Código 9](#_Toc120907431)

[MainParte1.java 9](#_Toc120907432)

[Argumentación 9](#_Toc120907433)

[Resultado 9](#_Toc120907434)

[Parte 2 9](#_Toc120907435)

[Enunciado 9](#_Toc120907436)

[Código 9](#_Toc120907437)

[MainParte2.java 9](#_Toc120907438)

[Argumentación 9](#_Toc120907439)

[Resultado 9](#_Toc120907440)

# --- Day 1: Calorie Counting ---

## Parte 1

### Enunciado

Santa's reindeer typically eat regular reindeer food, but they need a lot of magical energy to deliver presents on Christmas. For that, their favorite snack is a special type of star fruit that only grows deep in the jungle. The Elves have brought you on their annual expedition to the grove where the fruit grows.

To supply enough magical energy, the expedition needs to retrieve a minimum of fifty stars by December 25th. Although the Elves assure you that the grove has plenty of fruit, you decide to grab any fruit you see along the way, just in case.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the Advent calendar; the second puzzle is unlocked when you complete the first. Each puzzle grants one star. Good luck!

The jungle must be too overgrown and difficult to navigate in vehicles or access from the air; the Elves' expedition traditionally goes on foot. As your boats approach land, the Elves begin taking inventory of their supplies. One important consideration is food - in particular, the number of Calories each Elf is carrying (your puzzle input).

The Elves take turns writing down the number of Calories contained by the various meals, snacks, rations, etc. that they've brought with them, one item per line. Each Elf separates their own inventory from the previous Elf's inventory (if any) by a blank line.

For example, suppose the Elves finish writing their items' Calories and end up with the following list:

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

This list represents the Calories of the food carried by five Elves:

* The first Elf is carrying food with 1000, 2000, and 3000 Calories, a total of 6000 Calories.
* The second Elf is carrying one food item with 4000 Calories.
* The third Elf is carrying food with 5000 and 6000 Calories, a total of 11000 Calories.
* The fourth Elf is carrying food with 7000, 8000, and 9000 Calories, a total of 24000 Calories.
* The fifth Elf is carrying one food item with 10000 Calories.

In case the Elves get hungry and need extra snacks, they need to know which Elf to ask: they'd like to know how many Calories are being carried by the Elf carrying the most Calories. In the example above, this is 24000 (carried by the fourth Elf).

Find the Elf carrying the most Calories. How many total Calories is that Elf carrying?

### Código

#### Utils.java

**package** utils;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Scanner;

**public** **class** Utils {

**private** Utils() {

}

**public** **static** List<String> leerArchivo(String ruta) **throws** FileNotFoundException {

List<String> stringList = **new** ArrayList<>();

Scanner sc = **new** Scanner(**new** File(ruta));

**while** (sc.hasNextLine())

stringList.add(sc.nextLine());

sc.close();

**return** stringList;

}

}

#### MainParte1.java

**package** day1;

**import** java.io.FileNotFoundException;

**import** java.util.List;

**import** utils.Utils;

**public** **class** MainParte1 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

List<String> lista = Utils.*leerArchivo*("src/day1/input.txt");

**int** maximo = 0;

**int** sumatorio = 0;

**for** (String strNum : lista)

**if** (!strNum.equals(""))

sumatorio += Integer.*parseInt*(strNum);

**else** {

**if** (sumatorio > maximo)

maximo = sumatorio;

sumatorio = 0;

}

System.***out***.println(maximo);

}

}

### Resultado

70764

## Parte 2

### Enunciado

By the time you calculate the answer to the Elves' question, they've already realized that the Elf carrying the most Calories of food might eventually run out of snacks.

To avoid this unacceptable situation, the Elves would instead like to know the total Calories carried by the top three Elves carrying the most Calories. That way, even if one of those Elves runs out of snacks, they still have two backups.

In the example above, the top three Elves are the fourth Elf (with 24000 Calories), then the third Elf (with 11000 Calories), then the fifth Elf (with 10000 Calories). The sum of the Calories carried by these three elves is 45000.

Find the top three Elves carrying the most Calories. How many Calories are those Elves carrying in total?

### Código

#### MainParte2.java

**package** day1;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** utils.Utils;

**public** **class** MainParte2 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

List<String> lista = Utils.*leerArchivo*("src/day1/input.txt");

List<Integer> sumatorios = **new** ArrayList<>();

**int** sumatorio = 0;

**for** (String strNum : lista)

**if** (!strNum.equals(""))

sumatorio += Integer.*parseInt*(strNum);

**else** {

sumatorios.add(sumatorio);

sumatorio = 0;

}

Collections.*sort*(sumatorios);

System.***out***.println(sumatorios.get(sumatorios.size() - 1) + sumatorios.get(sumatorios.size() - 2)

+ sumatorios.get(sumatorios.size() - 3));

}

}

### Resultado

203905

# --- Day 2: Rock Paper Scissors ---

## Parte 1

### Enunciado

The Elves begin to set up camp on the beach. To decide whose tent gets to be closest to the snack storage, a giant Rock Paper Scissors tournament is already in progress.

Rock Paper Scissors is a game between two players. Each game contains many rounds; in each round, the players each simultaneously choose one of Rock, Paper, or Scissors using a hand shape. Then, a winner for that round is selected: Rock defeats Scissors, Scissors defeats Paper, and Paper defeats Rock. If both players choose the same shape, the round instead ends in a draw.

Appreciative of your help yesterday, one Elf gives you an encrypted strategy guide (your puzzle input) that they say will be sure to help you win. "The first column is what your opponent is going to play: A for Rock, B for Paper, and C for Scissors. The second column--" Suddenly, the Elf is called away to help with someone's tent.

The second column, you reason, must be what you should play in response: X for Rock, Y for Paper, and Z for Scissors. Winning every time would be suspicious, so the responses must have been carefully chosen.

The winner of the whole tournament is the player with the highest score. Your total score is the sum of your scores for each round. The score for a single round is the score for the shape you selected (1 for Rock, 2 for Paper, and 3 for Scissors) plus the score for the outcome of the round (0 if you lost, 3 if the round was a draw, and 6 if you won).

Since you can't be sure if the Elf is trying to help you or trick you, you should calculate the score you would get if you were to follow the strategy guide.

For example, suppose you were given the following strategy guide:

A Y

B X

C Z

This strategy guide predicts and recommends the following:

* In the first round, your opponent will choose Rock (A), and you should choose Paper (Y). This ends in a win for you with a score of 8 (2 because you chose Paper + 6 because you won).
* In the second round, your opponent will choose Paper (B), and you should choose Rock (X). This ends in a loss for you with a score of 1 (1 + 0).
* The third round is a draw with both players choosing Scissors, giving you a score of 3 + 3 = 6.

In this example, if you were to follow the strategy guide, you would get a total score of 15 (8 + 1 + 6).

What would your total score be if everything goes exactly according to your strategy guide?

### Código

#### MainParte1.java

**package** day2;

**import** java.io.FileNotFoundException;

**import** java.util.Arrays;

**import** java.util.List;

**import** utils.Utils;

**public** **class** MainParte1 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

List<String> lista = Utils.*leerArchivo*("src/day2/input.txt");

List<String> listJugadas = Arrays.*asList*("ZXY", "XYZ", "YZX");

**int** sumatorio = 0;

**for** (String strJugada : lista) {

**char** eleccionOponente = strJugada.charAt(0);

**char** miEleccion = strJugada.charAt(2);

String planteamiento = listJugadas.get(eleccionOponente - 'A');

sumatorio += planteamiento.indexOf(miEleccion) \* 3;

sumatorio += miEleccion - 'X' + 1;

}

System.***out***.println(sumatorio);

}

}

### Resultado

10941

## Parte 2

### Enunciado

The Elf finishes helping with the tent and sneaks back over to you. "Anyway, the second column says how the round needs to end: X means you need to lose, Y means you need to end the round in a draw, and Z means you need to win. Good luck!"

The total score is still calculated in the same way, but now you need to figure out what shape to choose so the round ends as indicated. The example above now goes like this:

* In the first round, your opponent will choose Rock (A), and you need the round to end in a draw (Y), so you also choose Rock. This gives you a score of 1 + 3 = 4.
* In the second round, your opponent will choose Paper (B), and you choose Rock so you lose (X) with a score of 1 + 0 = 1.
* In the third round, you will defeat your opponent's Scissors with Rock for a score of 1 + 6 = 7.

Now that you're correctly decrypting the ultra top secret strategy guide, you would get a total score of 12.

Following the Elf's instructions for the second column, what would your total score be if everything goes exactly according to your strategy guide?

### Código

#### MainParte2.java

**package** day2;

**import** java.io.FileNotFoundException;

**import** java.util.Arrays;

**import** java.util.List;

**import** utils.Utils;

**public** **class** MainParte2 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

List<String> lista = Utils.*leerArchivo*("src/day2/input.txt");

List<String> listJugadas = Arrays.*asList*("ZXY", "XYZ", "YZX");

**int** sumatorio = 0;

**for** (String strJugada : lista) {

**char** eleccionOponente = strJugada.charAt(0);

String planteamiento = listJugadas.get(eleccionOponente - 'A');

**char** debidoResultado = strJugada.charAt(2);

**char** miDebidaJugada = planteamiento.charAt(debidoResultado - 'X');

sumatorio += listJugadas.get(strJugada.charAt(0) - 'A').indexOf(miDebidaJugada) \* 3;

sumatorio += miDebidaJugada - 'X' + 1;

}

System.***out***.println(sumatorio);

}

}

### Resultado

13071

# --- Day 3: Rucksack Reorganization ---

## Parte 1

### Enunciado

One Elf has the important job of loading all of the rucksacks with supplies for the jungle journey. Unfortunately, that Elf didn't quite follow the packing instructions, and so a few items now need to be rearranged.

Each rucksack has two large compartments. All items of a given type are meant to go into exactly one of the two compartments. The Elf that did the packing failed to follow this rule for exactly one item type per rucksack.

The Elves have made a list of all of the items currently in each rucksack (your puzzle input), but they need your help finding the errors. Every item type is identified by a single lowercase or uppercase letter (that is, a and A refer to different types of items).

The list of items for each rucksack is given as characters all on a single line. A given rucksack always has the same number of items in each of its two compartments, so the first half of the characters represent items in the first compartment, while the second half of the characters represent items in the second compartment.

For example, suppose you have the following list of contents from six rucksacks:

vJrwpWtwJgWrhcsFMMfFFhFp

jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL

PmmdzqPrVvPwwTWBwg

wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn

ttgJtRGJQctTZtZT

CrZsJsPPZsGzwwsLwLmpwMDw

* The first rucksack contains the items vJrwpWtwJgWrhcsFMMfFFhFp, which means its first compartment contains the items vJrwpWtwJgWr, while the second compartment contains the items hcsFMMfFFhFp. The only item type that appears in both compartments is lowercase p.
* The second rucksack's compartments contain jqHRNqRjqzjGDLGL and rsFMfFZSrLrFZsSL. The only item type that appears in both compartments is uppercase L.
* The third rucksack's compartments contain PmmdzqPrV and vPwwTWBwg; the only common item type is uppercase P.
* The fourth rucksack's compartments only share item type v.
* The fifth rucksack's compartments only share item type t.
* The sixth rucksack's compartments only share item type s.

To help prioritize item rearrangement, every item type can be converted to a priority:

* Lowercase item types a through z have priorities 1 through 26.
* Uppercase item types A through Z have priorities 27 through 52.

In the above example, the priority of the item type that appears in both compartments of each rucksack is 16 (p), 38 (L), 42 (P), 22 (v), 20 (t), and 19 (s); the sum of these is 157.

Find the item type that appears in both compartments of each rucksack. What is the sum of the priorities of those item types?

### Código

#### MainParte1.java

**package** day3;

**import** java.io.FileNotFoundException;

**import** utils.Utils;

**public** **class** MainParte1 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

**int** sumatorio = 0;

**for** (String s : Utils.*leerArchivo*("src/day3/input.txt"))

sumatorio += *prioridadUnicoCaracterComun*(s.substring(0, s.length() / 2), s.substring(s.length() / 2));

System.***out***.println(sumatorio);

}

**public** **static** **int** prioridadUnicoCaracterComun(String... cadenas) {

**for** (**char** c : cadenas[0].toCharArray()) {

**boolean** enTodas = **true**;

**for** (**int** i = 1; i < cadenas.length && enTodas; i++)

enTodas = cadenas[i].contains(c + "");

**if** (enTodas)

**return** c >= 'a' ? c - 'a' + 1 : c - 'A' + 27;

}

**return** 0;

}

}

### Resultado

7850

## Parte 2

### Enunciado

As you finish identifying the misplaced items, the Elves come to you with another issue.

For safety, the Elves are divided into groups of three. Every Elf carries a badge that identifies their group. For efficiency, within each group of three Elves, the badge is the only item type carried by all three Elves. That is, if a group's badge is item type B, then all three Elves will have item type B somewhere in their rucksack, and at most two of the Elves will be carrying any other item type.

The problem is that someone forgot to put this year's updated authenticity sticker on the badges. All of the badges need to be pulled out of the rucksacks so the new authenticity stickers can be attached.

Additionally, nobody wrote down which item type corresponds to each group's badges. The only way to tell which item type is the right one is by finding the one item type that is common between all three Elves in each group.

Every set of three lines in your list corresponds to a single group, but each group can have a different badge item type. So, in the above example, the first group's rucksacks are the first three lines:

vJrwpWtwJgWrhcsFMMfFFhFp

jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL

PmmdzqPrVvPwwTWBwg

And the second group's rucksacks are the next three lines:

wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn

ttgJtRGJQctTZtZT

CrZsJsPPZsGzwwsLwLmpwMDw

In the first group, the only item type that appears in all three rucksacks is lowercase r; this must be their badges. In the second group, their badge item type must be Z.

Priorities for these items must still be found to organize the sticker attachment efforts: here, they are 18 (r) for the first group and 52 (Z) for the second group. The sum of these is 70.

Find the item type that corresponds to the badges of each three-Elf group. What is the sum of the priorities of those item types?

### Código

#### MainParte2.java

**package** day3;

**import** java.io.FileNotFoundException;

**import** java.util.List;

**import** utils.Utils;

**public** **class** MainParte2 {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

List<String> lista = Utils.*leerArchivo*("src/day3/input.txt");

**int** sumatorio = 0;

**for** (**int** i = 0; i < lista.size(); i += 3)

sumatorio += MainParte1.*prioridadUnicoCaracterComun*(lista.get(i), lista.get(i + 1), lista.get(i + 2));

System.***out***.println(sumatorio);

}

}

### Resultado

2581

# Día 4

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 5

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 6

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 7

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 8

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 9

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

El código se encarga de simular el movimiento de una cuerda. La función main() recibe como entrada un archivo con una serie de movimientos en forma de cadenas de caracteres. El primer carácter representa la dirección determinada (arriba, abajo, derecha o izquierda) y el número que sigue indica cuántos pasos se deben dar en esa dirección.

La función comienza asumiendo que el nudo de la cabeza (h - head) y la cola (t - tail) de la cuerda están en la misma posición, cubriéndose mutuamente.

Luego, la función itera sobre esta lista de movimientos. Para cada movimiento, se actualiza la posición de la cabeza de la cuerda según la dirección y el número de pasos especificados en la tupla. Si después de actualizar la posición de la cabeza, la cola ya no sigue en contacto, se actualiza también la cola de acuerdo al criterio de cercanía especificado en el enunciado. La cola se debe de encontrar dentro del recuadro 3x3 cuyo centro es la cabeza.

Para ello, después de actualizar la posición de la cabeza de la cuerda, la función comprueba si la cabeza y la cola siguen estando en contacto. Si sigue habiendo contacto, no se realiza ninguna acción adicional. Si ya no hay contacto, se actualiza la posición de la cola de acuerdo a las siguientes reglas:

* Si la cabeza y la cola están en la misma fila o columna, la cola se mueve un paso en la misma dirección que la cabeza.
* En cualquier otro caso, la cola se mueve un paso en la dirección diagonal que la acerque a la cabeza para situarla detrás.

Esta serie de reglas asegura que la cola siempre siga de cerca a la cabeza, sin importar el número de movimientos consecutivos que se realicen.

Para cada movimiento de la cola, éste se registra en una estructura de datos de tipo conjunto de no repetición (set) para después numerar cuántas casillas distintas ha visitado la cola.

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

A diferencia de la primera parte del reto, ahora no solamente se tiene que mover la cola (t), sino una serie de 9 nudos detrás de la cabeza (h).

Aunque el enunciado pueda resultar algo complejo de entender a primera vista, solamente viene a querer decir que los nudos se muevan en orden solamente si necesitan moverse para acercarse con respecto al anterior y seguir respetando el criterio de cercanía, tras haberse movido la cabeza.

La función moverNudos() se encarga de actualizar en orden dichos nudos para, de uno en uno, asegurarse que está cerca del anterior y moverlo si no, con el objetivo de asegurar que todos los nudos sigan de cerca a la cabeza de la cuerda.

### Resultado

# Día 10

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

En resumen, después de haber calculado la lista de valores que va tomando X en cada ciclo, estos valores se corresponden a la columna donde está situado el sprite en cada ciclo. Si los tres píxeles del sprite cubren el píxel que se está dibujando en el ciclo actual, este píxel quedará iluminado (‘#’); y no en caso contrario (‘.’). Entrando más en detalle:

La pantalla viene representada por una matriz de píxeles de 6 filas y 40 columnas, reuniendo un total de 240 celdas. Además, a lo largo de las 146 instrucciones ‘addx V’ y ‘noop’ se recorre un total de 240 ciclos; es decir, cada celda de la matriz viene correspondida con un ciclo tal y como explica el enunciado en esta parte:

Ciclo 1 🡪 ######################################## 🡨 Ciclo 40

…

Ciclo 201 🡪 ######################################## 🡨 Ciclo 240

Es decir, la posición del valor X en la lista correspondiente a una celda cuya fila sea F (0..5) y columna sea C (0..39), empezando la indexación en 0, vendrá dada por la fórmula 40\*F+C. Por ejemplo, al momento de dibujar la celda de la segunda fila (F=1) y séptima columna (C=6), la posición correspondiente en la lista será 40\*1+6=46. (La primera fila va de 0 a 39, la segunda de 40 a 79, y así sucesivamente hasta la última fila que va desde 200 hasta 239).

La parte del enunciado en la que se explica cómo determinar si un píxel debe estar iluminado o no, es la siguiente:

It seems like the X register controls the horizontal position of a sprite. Specifically, the sprite is 3 pixels wide, and the X register sets the horizontal position of the middle of that sprite. (In this system, there is no such thing as "vertical position": if the sprite's horizontal position puts its pixels where the CRT is currently drawing, then those pixels will be drawn.)

Esto viene a querer decir que el registro X controla la componente horizontal de la posición del sprite, es decir, su columna, tomando valores en torno a entre 1 y 40. Además, se sabe que el sprite ocupa tres píxeles de ancho, cuya columna del píxel central viene dada por el valor de X.

En cada ciclo de los 240 ciclos, el sprite estará situado en una columna en concreto de la misma fila del píxel que se está dibujando. Es decir, por ejemplo, en el i-ésimo ciclo, el centro del sprite estará situado en la columna correspondiente al valor de X de la i-ésima posición de la lista.

A modo de ejemplo, supongamos que, durante el dibujado del píxel del quinto ciclo, es decir, el quinto pixel, el sprite se encuentra en la posición 10 (dada por el valor de X), cubriendo los píxeles 9, 10 y 11. En este caso, el sprite no está cubriendo el pixel que se está dibujando en este ciclo, es decir, el quinto.

Si el sprite a través de su anchura de 3 cubriese el píxel que se está dibujando en ese mismo ciclo, el píxel quedaría iluminado (‘#’); y no en caso contrario (‘.’). Por tanto, para saber si el sprite está cubriendo el píxel a dibujar, será tan sencillo como calcular la distancia en valor absoluto entre el punto central del sprite y el píxel a dibujar; como la anchura es de 3, si la distancia es superior a 1, no se estaría cubriendo.

### Resultado

# Día 11

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 12

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 13

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 14

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 15

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 16

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 17

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 18

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 19

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 20

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 21

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 22

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 23

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 24

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado

# Día 25

## Parte 1

### Enunciado

### Código

#### MainParte1.java

### Argumentación

### Resultado

## Parte 2

### Enunciado

### Código

#### MainParte2.java

### Argumentación

### Resultado