Relatório Projeto Países/Produção de Frutos

Realizado por:

1211131 – Pedro Pereira

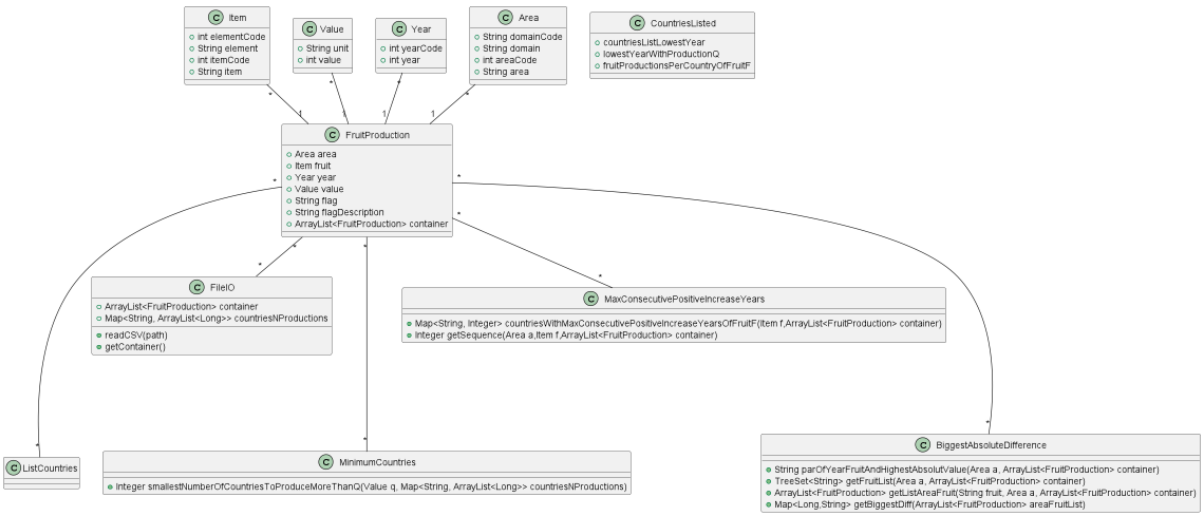
1211151 – Alexandre Geração

1211128 – Tiago Oliveira

1211089 – José Gouveia

1210828 – Ricardo Venâncio

Diagrama de classes



Algoritmos usados

Exercício 1 – Carregar estruturas de dados com informação de ficheiros .csv

Nesta primeira parte código do método "*readCSV*", o algoritmo vai fazer uma validação do "*path*" recebido por parâmetro.

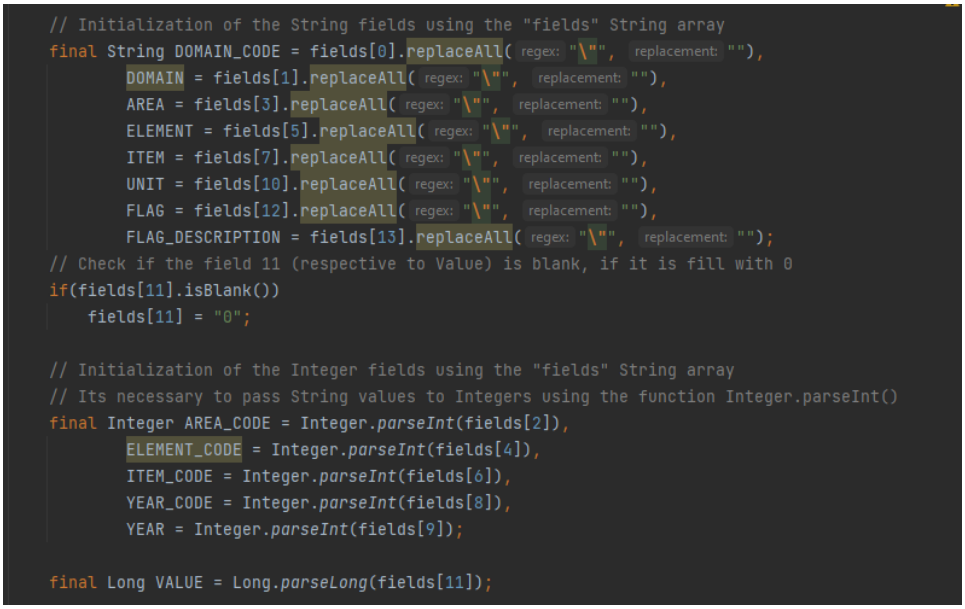
```
public boolean readCSV(String path) throws FileNotFoundException {

    if(path == null)
        return false;

    File f = new File(path);

    if(!f.exists())
        return false;
```

Após verificar que o caminho é destinado a um ficheiro, a leitura inicia. O cabeçalho é lido e não guardado e enquanto houver linhas no ficheiro, o algoritmo vai trabalhar. É utilizada uma expressão regular para fazer a correta leitura dos vários campos do ficheiro recebido. Se a linha lida do ficheiro contiver uma aspa, a separação vai ser feita sempre que forem detetadas as seguintes sequências de caracteres: \,\ OU \

Caso a linha não contenha aspas, a separação é feita sempre que for encontrada uma vírgula.

```
Scanner sc = new Scanner(f);

// Reads the file headr (no data!).
sc.nextLine();

while(sc.hasNext()){

    // Variable line will contain data from the .csv each iteration. Fields will be the splitter.
    String line = sc.nextLine();
    String[] fields;

    // There isn't only one type of .csv documents, in other words, data is separated by commas, sometimes
    // have quotation marks, null fields and a field with a comma to maintain.
    // So, checking if the line contains a quotation mark, will make the regular expression
    // (to split) a bit different.
    if(line.contains("\""))
        fields = line.split( regex: "\",\"");
    else
        fields = line.split( regex: ",");
```

Nesta parte é feita a atribuição e "correção" dos campos.



```
// Initialization of the String fields using the "fields" String array
final String DOMAIN_CODE = fields[0].replaceAll( regex: "\"",  replacement: ""),
        DOMAIN = fields[1].replaceAll( regex: "\"",  replacement: ""),
        AREA = fields[3].replaceAll( regex: "\"",  replacement: ""),
        ELEMENT = fields[5].replaceAll( regex: "\"",  replacement: ""),
        ITEM = fields[7].replaceAll( regex: "\"",  replacement: ""),
        UNIT = fields[10].replaceAll( regex: "\"",  replacement: ""),
        FLAG = fields[12].replaceAll( regex: "\"",  replacement: ""),
        FLAG_DESCRIPTION = fields[13].replaceAll( regex: "\"",  replacement: "");
// Check if the field 11 (respective to Value) is blank, if it is fill with 0
if(fields[11].isBlank())
    fields[11] = "0";

// Initialization of the Integer fields using the "fields" String array
// Its necessary to pass String values to Integers using the function Integer.parseInt()
final Integer AREA_CODE = Integer.parseInt(fields[2]),
        ELEMENT_CODE = Integer.parseInt(fields[4]),
        ITEM_CODE = Integer.parseInt(fields[6]),
        YEAR_CODE = Integer.parseInt(fields[8]),
        YEAR = Integer.parseInt(fields[9]);

final Long VALUE = Long.parseLong(fields[11]);
```

De seguida tem as inicializações dos objetos.

```
// Initialization of the objects Area,Item, Year and Value using the elements initialized before
Area a = new Area(DOMAIN_CODE, DOMAIN, AREA_CODE, AREA);
Item i = new Item(ELEMENT_CODE, ELEMENT, ITEM_CODE, ITEM);
Year y = new Year(YEAR_CODE, YEAR);
Value v = new Value(UNIT, VALUE);

// Initialization of the object Fruit Production compound by the Objects initialized before
FruitProduction fp = new FruitProduction(a, i, y, v, FLAG, FLAG_DESCRIPTION);
```

E por final tem a criação de uma estrutura de dados que armazena os países e uma lista de valores de produção e um contentor que armazena a informação de uma forma integral.

```java
// Countries and productions map.
if(countriesNProductions.get(AREA) == null) {
    ArrayList<Long> quantities = new ArrayList<>();
    countriesNProductions.put(AREA, quantities);
}
countriesNProductions.get(AREA).add(VALUE);


// Adding the Fruit Production object to a container (ArrayList of FruitProductions)
container.add(fp);
```

Exercício 2 – Lista de países com pelo menos um ano de produção do fruto F com quantidade maior ou igual a Q.

Este método tem como parâmetros uma fruta do tipo *Item*, uma quantidade do tipo *long* (que seria o atributo *value* da classe *Value*), um contentor com as *FruitProductions e uma lista com os nomes dos países registados*. Inicialmente vai verificar se ora o contentor, a lista dos países ou a fruta \são nulos e caso se confirme, o algoritmo não executa.

```java
public static LinkedList<FruitProduction> countriesListLowestYear(ArrayList<FruitProduction> container,Set<String> countryNames, Item fruit, long productionQuantity) {
    if (container == null || fruit == null || countryNames == null) return null;
    LinkedList<FruitProduction> countriesList = new LinkedList<>();

    TreeSet<FruitProduction> fruitProductionsOfAItemPerCountry;
    FruitProduction aux;
```

Após a verificação dos valores é criado um TreeSet de FruitProductions, que vai armazenar as produções de um fruto F de um país, e uma FruitProduction, que vai auxiliar ao preenchimento da lsita de retorno (countries). Apos a criação destes dois objetos, o programa ira iterar através dos nomes dos países guardados em countryNames, e por cada nome ira chamar os seguintes métodos: fruitProductionsOfAItemPerCountry e lowestYearWithProductionQ (o funcionamento destes métodos será explicado mais a frente deste relatório). Estes métodos trabalham em conjunto para determinar o menor ano em que o país produziu um fruto F com uma quantidade maior ou igual do que Q (sabendo que o país tem de ter pelo menos um ano de produção), caso o objeto contido na variável aux seja null o algoritmo apago todos os valores armazenados em productionsOfAItem e passa para a próxima iteração do ciclo, caso contrário irá acrescentar a lista countriesList o valor armazenado em aux e apaga os valores armazenados em fruitProductionsPerCountry, e procede a próxima iteração do ciclo.

```java
for (String countryName : countryNames) {
    fruitProductionsOfAItemPerCountry = fruitProductionsPerCountryOfFruitF(container, fruit, countryName);
    aux = lowestYearWithProductionQ(fruitProductionsOfAItemPerCountry, productionQuantity);

    if (aux == null) {
        fruitProductionsOfAItemPerCountry.clear();
        continue;
    }

    countriesList.add(aux);
    fruitProductionsOfAItemPerCountry.clear();
}
```

Uma vez o programa percorra todos os nomes dos países armazenados em countryNames o programa vai recorrer ao método sort, passando como parâmetro um objeto que implemente a interface Comparator (Comparator de FruitProduction), para assim ordenar os valores armazenados na lista countries, e posterior mente ao retorno da mesma.

```
            productionsOfAItem.clear();
        }

        countries.sort(new FruitProductionComparator());
        return countries;
    }
```

```
public class FruitProductionComparator implements Comparator<FruitProduction> {
    jose
    @Override
    public int compare(FruitProduction o1, FruitProduction o2) {
        if (o1.getYear().getYear() == o2.getYear().getYear()) {
            if (o1.getValue().getValue() == o2.getValue().getValue()) {
                return 0;
            } else if (o1.getValue().getValue() < o2.getValue().getValue()) {
                return 1;
            } else {
                return -1;
            }
        } else if (o1.getYear().getYear() > o2.getYear().getYear()) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

O método fruitProductionPerCountryOfFruitF recebe como parâmetros, uma lista com todos os objetos de FruitProduction, o fruto F e o nome do país que queremos obter os resultados. Caso algum dos parâmetros seja nulo o algoritmo retorna null. Caso contrário o algoritmo criara um TreeSet onde ira guardar os dados de um determinado país (o TreeSet recebe como parâmetro um objeto que implemente a interface Comparator de FruitProduction, assim o TreeSet ficará ordenado, isto será necessário para o próximo método), depois percorrerá todos os objetos do container verificando se o fruto e nome do país guardados no objeto correspondem aos que foram passados como parâmetro, caso assim seja este objeto será acrescentado ao TreeSet inicial. Uma vez o algoritmo percorra todos os objetos do container o programa retorna o TreeSet countryList.

```
public static TreeSet<FruitProduction> fruitProductionsPerCountryOfFruitF(ArrayList<FruitProduction> container, Item fruit, String countryName) {
    if (container == null || fruit == null || countryName == null) return null;
    TreeSet<FruitProduction> countryList = new TreeSet<>(new FruitProductionComparator());
    for (FruitProduction production : container) {
        if (production.getFruit().getItemCode().equals(fruit.getItemCode()) && production.getArea().getArea().equals(countryName)) {
            countryList.add(production);
        }
    }
    return countryList;
}
```

O método llowestYearWithProductionQ recebe dois parâmetros, o TreeSet criado no método anterior e a quantidade mínima de produção requerida, caso o primeiro parâmetro seja nulo o programa retorna null. Apos esta verificação o programa inicializa um contador int que representa os anos que o pais já produziu o fruto z (isto é possível uma vez que o TreeSet já foi ordenado no método anterior), o programa ira percorrer todos os valores armazenados em fruitProductionsOfAItemPerCountry verificando se o país pelo menos leva um ano de produção e a quantidade produzida é maior ou igual a especificada pela variável quantity, o algoritmo irá retornar o objeto que cumpra com aquelas condições, caso contrario retorna null.

```
public static FruitProduction lowestYearWithProductionQ(TreeSet<FruitProduction> fruitProductionsOfAItemPerCountry, long productionQuantity) {
    if (fruitProductionsOfAItemPerCountry == null) return null;
    int yearsProducing = 0;
    for (FruitProduction production : fruitProductionsOfAItemPerCountry) {
        if ((yearsProducing >= 1) && (production.getValue().getValue() >= productionQuantity)) {
            return production;
        }
        yearsProducing++;
    }
    return null;
}
```

Exercício 3 – Número mínimo de países que, em conjunto, produz mais que Q.

O seguinte método vai receber uma quantidade Q por parâmetro, criar uma variável para armazenar o valor introduzido (Value é um objeto que contém o atributo *unit* e *value*, para este método apenas o valor é necessário) e por final criar uma lista ligada que vai armazenar a produção total de todos os países em questão (despreza o nome dos países, pois é pedido o número mínimo). Vai também verificar se os valores recebidos por argumento são válidos para iniciar o algoritmo.

```java
public Integer smallestNumberOfCountriesToProduceMoreThanQ(Value quantity, Map<String, ArrayList<Long>> countriesNProductions){

    final int NO_COUNTRIES = 0;

    Long value = quantity.getValue();

    if(value <= 0 || countriesNProductions.isEmpty())
        return -1;


    List<Long> countries = new LinkedList<>();
```

De seguida, entra num *loop* que vai percorrer um mapa, que armazena os países (em *keys*) e uma lista de produções (em *values*), soma as quantidades produzidas de cada país e armazena no *array countries* as quantidades totais existentes. Após todas as quantidades serem registadas, *countries* vai ser ordenado de forma descendente.

```java
for(ArrayList<Long> al : countriesNProductions.values()){
    Long sum=0L;
    for(Long i : al){
        sum += i;
    }
    countries.add(sum);
}

countries.sort(Collections.reverseOrder());
```

Para perceber o motivo de *countries* ser ordenado de forma descendente, é preciso chegar à última parte do código. Para perceber o número mínimo de países a produzir, deve-se comparar o valor Q passado por argumento com o país que mais produz, caso não seja suficiente, vai sucessivamente calcular até ser suficiente ou até ao final.

```java
long sum = 0;
int count = 0;

int numberOfCountries = countries.size()-1;

while(sum < value && count <= numberOfCountries) sum += countries.get(count++);

return (sum < value) ? NO_COUNTRIES : count;
```

Exercício 4 – Dado um fruto F, devolve os países agrupados pelo número máximo de anos consecutivos em que houve crescimento de quantidade de produção do fruto F.

O método seguinte é o principal método do exercício 4. Este recebe um item, ou seja, uma fruta escolhida pelo utilizador e também recebe a variável "container" onde vai buscar todos os dados. Começa por verificar se a fruta escolhida já está presente no HashMap. Se esta já estiver presente com os respetivos dados para todos os países não efetua mais nenhuma operação. Caso contrário, esta verifica quais dos países é que ainda não têm os dados sobre o máximo número de anos consecutivos com crescimento de quantidade de produção do fruto

escolhido, e manda os dados do fruto e país para o método "getSequence". Após receber os dados de volta, atualiza o HashMap.

```
public Map<String, Integer> countriesWithMaxConsecutivePositiveIncreaseYearsOfFruitF(Item f,ArrayList<FruitProduction> container){
    Map<String,Integer> map = new HashMap<>();
    for (FruitProduction fp : container) {
        if(fp.fruit.item.equals(f.item)){
            if(!map.containsKey(fp.area.area)){
                map.put(fp.area.area, getSequence(fp.area,f,container));
            }
        }
    }
    return map;
}
```

O método "getSequence" recebe uma fruta na forma de um item, um país na forma de uma área e o ficheiro com todos os dados. É criada uma variável "count" para ir mantendo conta do máximo número de anos e a variável "save" é usada como forma de guardar o valor final para dar de volta ao método principal. Esta começa como -1 para ser logo um valor que possa aumentar se for encontrado uma sequência de anos.

Após a criação das variáveis, cria um ciclo "for" em que procura linha a linha pelo ficheiro pelo país e fruta. Quando encontra esses dados verifica linha a linha se o valor de número de frutos produzido de um ano é inferior ao do ano seguinte. Se este for o caso, aumenta a contagem na variável "count". Mal encontre uma linha onde a produção diminui de um ano para o outro, verifica se a contagem obtida é superior à contagem máxima guardada na variável "save" ou não, sobrepondo o valor se esta condição for verdadeira. Finalmente reinicia o "count" e volta a fazer a comparação na linha seguinte do ficheiro. Faz este ciclo até terminar de ebcontrar casos em que a fruta e o país escolhidos estejam presentes.

```
public Integer getSequence(Area a,Item f,ArrayList<FruitProduction> container){
    Integer count=0;
    Integer save =-1;
    for (int i = 0; i < container.size(); i++) {
        if(container.get(i).area.area.equals(a.area)){
            if(container.get(i).fruit.item.equals(f.item)){
                while ((container.get(i).area.area.equals(a.area))&&(container.get(i).fruit.item.equals(f.item))&&(container.get(i).value.value<(container.get(i+1).value.value))){
                    count++;
                    i++;
                    if(i==container.size()-1){
                        break;
                    }
                }
            }
        }
        if(count>save){
            save=count;
        }
        count=0;
    }
    return save;
}
```

Exercício 5 – Dado um país P, devolve o par de anos, o fruto e o maior valor absoluto de diferença de produção.

Este primeiro método é o método principal da classe e do exercício. É a partir dele que alcançamos o resultado pretendido através da ajuda de outros métodos "secundários" criados para realizar certas tarefas necessárias.

Inicialmente, o método recebe como parâmetros uma certa "Area" (um país) e o contentor ("container") que possui todos os dados lidos a partir do ficheiro ".csv". De seguida cria um *Map* que irá receber as informações relativas à maior diferença absoluta da produção entre dois anos e é criado ainda um *TreeSet* que guarda o resultado devolvido pela chamada do método "getFruitList". Este *TreeSet* é útil pois guarda apenas as frutas que foram produzidas pelo país em questão e vai ser utilizado no ciclo *for* que se segue. Neste ciclo *for* vamos percorrer o *TreeSet* que foi previamente criado e para cada fruta nele presente vai criar um *ArrayList* com as informações acerca da "Fruit Production". Este *ArrayList* vai ser passado como parâmetro para o método "getBiggestDiff", onde vai ser calculada a maior diferença absoluta de produção para a fruta atual. Por fim o nome da fruta e o valor da diferença, juntamente com os anos em que se verificou vão ser guardados no *Map*.

```java
/**Given a certain country (Area), it searches the cointainer with all the information about the fruit production and finds
 * the pair of years and the fruit of which the difference of production was the biggest. First, it stores the all the fruits
 * that were produced by the given country in a TreeSet, by calling the method getFruitList. Then, it goes through this TreeSet
 * and, for each fruit, gets the information of the production of the fruit, by calling the method getListAreaFruit, and stores
 * it in a ArrayList. Then it takes this ArrayList and will use it to calculate the biggest absolute difference, calling the method
 * getBiggestDiff and stores the results in the Map. Finally, it will compare all the results of at the Map to discover which one is
 * the biggest absolute difference out of all fruits.
 *
 * @param area a certain country information
 * @param  container the list containing all FruitProduction objects
 * @return a string with the par of years, the fruit and the biggest absolute difference of production*/
4 usages  1211128
public static String pairOfYearFruitAndHighestAbsolutValue(Area area, ArrayList<FruitProduction> container){
    Map<String,Map<Long,String>> map = new HashMap<>();
    TreeSet<String> fruitsOfCountry = getFruitList(area, container);

    for (String fruitName : fruitsOfCountry){
        ArrayList<FruitProduction> areaFruitList = getListAreaFruit(fruitName,area,container);
        Map<Long,String> diffAndYearsMap= getBiggestDiff(areaFruitList);
        map.put(fruitName,diffAndYearsMap);
    }

    Long maxValue = 0L;
    int index = 0;
    String output = "";
    Object[] listFruits = map.keySet().toArray();
```

No que podemos considerar a segunda parte deste método, vai ser feita a comparação dos valores da maior diferença absoluta que estão guardados no *Map*. Como no *Map* foram guardados os dados da maior diferença absoluta, mas apenas falando da mesma fruta, agora comparemos todos os valores até encontrarmos o verdadeiro maior valor e para isso é utilizado um ciclo for, onde se o valor da diferença do *Map* for maior do que aquele armazenado na variável "maxValue", o valor desta variável vai ser trocado pelo novo maior valor. Ao mesmo tempo vai sendo construída a *String* output que irá conter o resultado final que será retornado pelo método.

```java
    Long maxValue = 0L;
    int index = 0;
    String output = "";
    Object[] listFruits = map.keySet().toArray();


    for (Map map1 : map.values()){
        if (Long.parseLong(map1.keySet().toString().replaceAll( regex: "[()\\[\\]]",  replacement: "")) >= maxValue){
            maxValue = Long.parseLong(map1.keySet().toString().replaceAll( regex: "[()\\[\\]]",  replacement: ""));
            output = String.format("[%s,%s,%d]",map1.get(maxValue),listFruits[index],maxValue);
        }
        index++;
    }

    return output;
}
```

Este método "getFruitList" retorna um *TreeSet* (escolhido por não permitir valores repetidos), este *TreeSet* é composto por todas as frutas que existem numa "area", que é recebida por parâmetro. Para isso este método percorre todos os valores de "Fruit Production" contidos num *ArrayList* de "FruitProductions" chamado "container", verifica se a Area coincide com a pretendida e se coincidir adiciona ao *TreeSet*.

```java
/**This method goes through the container and searches for all the fruits that were produced by the given country (area)
 *
 * @param area a certain country information
 * @param container the list containing all FruitProduction objects
 * @return a TreeSet with all the fruits of the given country*/
5 usages  1211128
public static TreeSet<String> getFruitList(Area area, ArrayList<FruitProduction> container){
    TreeSet<String> fruits = new TreeSet<>();

    for (FruitProduction fp : container){
        if(area.area.equals(fp.area.area)){
            fruits.add(fp.fruit.item);
        }
    }
    return fruits;
}
```

Este método "getListAreaFruit" retorna uma lista com os objetos do tipo "FruitProduction" com o fruto e área passada por parâmetro. O método, primeiro cria um *ArrayList* do tipo "FruitList", apos a sua criação o algoritmo procede a percorrer todos os objetos "FruitProduction" no container e sempre que encontrar algum objeto que cumpra os requisitos (ter o mesmo fruto e área especificados no parâmetro do método), acrescenta o objeto à lista "areaFruitList", uma vez percorrido todos os objetos do container o método retorna a lista já preenchida.

```java
/**This method goes through the container and gets all the information about the production of the given fruit in the given country (area).
 *
 * @param fruit the name of the wanted fruit
 * @param area a certain country information
 * @param container the list containing all FruitProduction objects
 * @return an ArrayList with the information of a*/
7 usages  1211128
public static ArrayList<FruitProduction> getListAreaFruit(String fruit, Area area, ArrayList<FruitProduction> container){
    ArrayList<FruitProduction> areaFruitList = new ArrayList<>();

    for (FruitProduction fp : container){
        if((fp.area.area.equals(area.area) && (fp.fruit.item.equals(fruit)))){
            areaFruitList.add(fp);
        }
    }
    return areaFruitList;
}
```

Este método "getBiggestDiff" retorna uma *Map* composto por um *Long* e uma *String*, o *Long* é relativo à maior diferença de produções apresentada e a *String* é relativa aos anos onde ocorreram essas diferenças. Para achar este valor o método recebe um *ArrayList* de "FruitProductions" composto por todas as instâncias de um certo produto do país que nós queremos, com estes dados iremos comparar os anos de produção começando no primeiro elemento da lista e comparando com todos até ao final da lista à medida que a diferença for ficando maior o valor máximo é guardado numa variável chamada "diff" bem como os anos respetivos na variável "year". De seguida pegamos no segundo elemento da lista e comparamos com todos os valores até ao fim da lista repetindo este procedimento até ao penúltimo elemento. No fim é retornado o *Map* composto pelo maior valor e pelos respetivos anos.

```java
/**This method goes through the list, and calculates the biggest difference in the production between all years.
 * It takes two years at a time, calculates their difference and stores the value so that it can be compared in the next
 * calculation of the difference. Every time that appeares a difference bigger than the one stored, it updates the value of the
 * variable diff.
 *
 * @param areaFruitList list with all the information about the production of a certain fruit in a certain country
 * @return a map with the biggest absolute difference of production of a certain fruit and the pair of years*/
4 usages   1211128
public static Map<Long,String> getBiggestDiff(ArrayList<FruitProduction> areaFruitList){
    Map<Long,String> map = new HashMap<>();
    Long diff = 0L;
    String year = "";

    for (int i = 0; i < areaFruitList.size()-1; i++) {
        for (int j = i+1; j < areaFruitList.size(); j++) {
            if(Math.abs(areaFruitList.get(i).value.value - areaFruitList.get(j).value.value) > diff){
                diff = Math.abs(areaFruitList.get(i).value.value - areaFruitList.get(j).value.value);
                year = areaFruitList.get(i).year.year + "/" + areaFruitList.get(j).year.year;
            }
        }
    }

    map.put(diff,year);
    return map;
}
```

Melhoramentos possíveis

- Melhorar a leitura dos dados, neste caso poderíamos alterar o modo de leitura de dados para guardar os dados de outra maneira de forma a otimizar a busca dos dados nos exercícios.

- Melhorar esta parte do código:

```java
Map<String,Map<Long,String>> map = new HashMap<>();
```

```java
Object[] listFruits = map.keySet().toArray();


for (Map map1 : map.values()){
    if (Long.parseLong(map1.keySet().toString().replaceAll( regex: "[()\\[\\]]",  replacement: "")) >= maxValue){
        maxValue = Long.parseLong(map1.keySet().toString().replaceAll( regex: "[()\\[\\]]",  replacement: ""));
        fruit = String.format("[%s,%s,%d]",map1.get(maxValue),listFruits[index],maxValue);
    }
    index++;
}
```

Nesta parte compreendemos que haveria forma mais eficiente de construir esta parte do código, a criação de um Map onde o Value vai ser outro Map com apenas um Long e uma String

não parece ser muito eficiente, para além disso dificulta a análise dos valores contidos neles o que nos levou a criar um Array de Object que é genérico demais.