



# RELATÓRIO PROJETO PAÍSES/PRODUÇÃO DE FRUTOS

ESTRUTURAS DE INFORMAÇÃO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

André Barros \_ 1211299

Carlos Lopes \_ 1211277

Ricardo Moreira \_ 1211285

Tomás Russo \_ 1211288

Tomás Lopes \_ 1211289



# Índice

O Problema.....	2
A Solução .....	3
Diagrama de classes.....	3
Algoritmos usados .....	6
CountriesWithGreaterProduction .....	9
CountriesWithProductionGrowth .....	11
CountrySetWithHigherProduction.....	14
GreatestDifferenceInProduction.....	17
Possíveis melhorias.....	18

# O Problema

O objetivo deste projeto passa por gerir, de forma eficiente, a informação relativa à produção de vários tipos de fruto ao longo dos anos, em vários países.

Esta informação encontra-se disponível a partir de um ficheiro com extensão CSV que deve ser carregada pelo programa.

A grande questão é tornar a introdução e consulta destes dados o mais eficientes possível a partir da escolha das estruturas de informação que melhor se adaptam ao problema.

# A Solução

## Diagrama de classes

Toda a estrutura de dados utilizada foi abstraída por outras classes que permitem manipular os dados através de uma interface específica que não depende da estrutura utilizada internamente

Desta forma, ganhamos flexibilidade e permite que cada módulo seja testado independentemente dos outros com todas as suas especificidades e condições.

Depois de analisado o problema apresentado, a solução encontrada foi a seguinte:

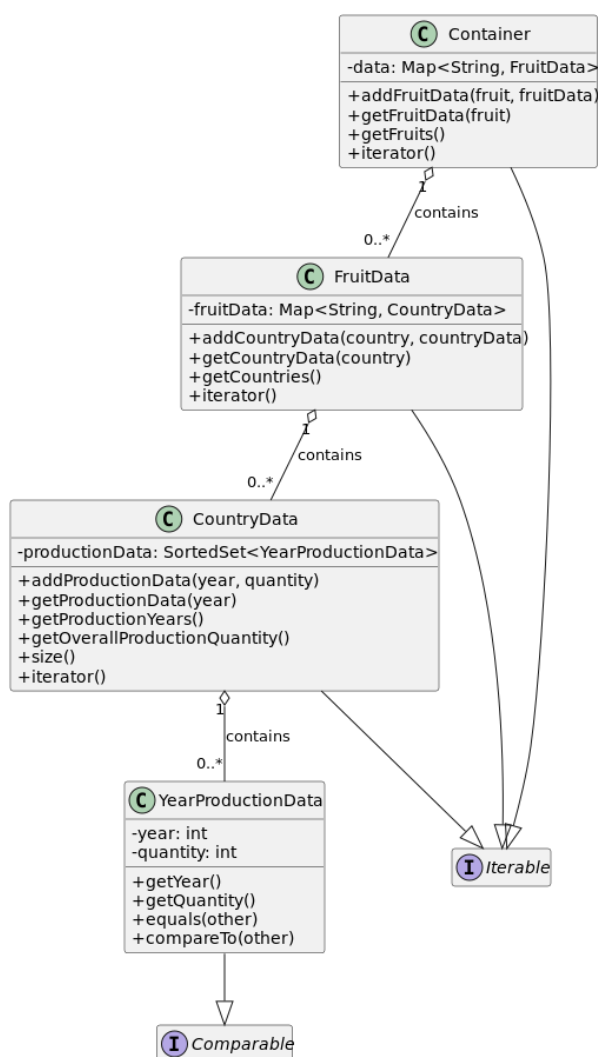


Figura 1 \_ Class Diagram simplificado

A solução não é perfeita para todos as funcionalidades, mas acabou por ser a que consideramos mais adequada a reduzir a complexidade da maioria das operações pedidas.

Ao decompor e simplificar estas classes numa só estrutura ficaríamos com:

```
Map<String, Map<String, SortedSet<YearProductionData>>>
```

onde as *keys* representam, respetivamente, Fruto, País e Ano de produção. A razão para a escolha dos mapas é a complexidade constante para as operações de consulta a uma *key* específica como é exigido na maioria das funcionalidades.

Pela mesma razão, como não é necessária a consulta dos dados de um ano em específico em nenhuma das funcionalidades não foi necessário o uso de um *Map* no último nível da estrutura.

Estas classes são usadas da seguinte forma em cada *usecase*:

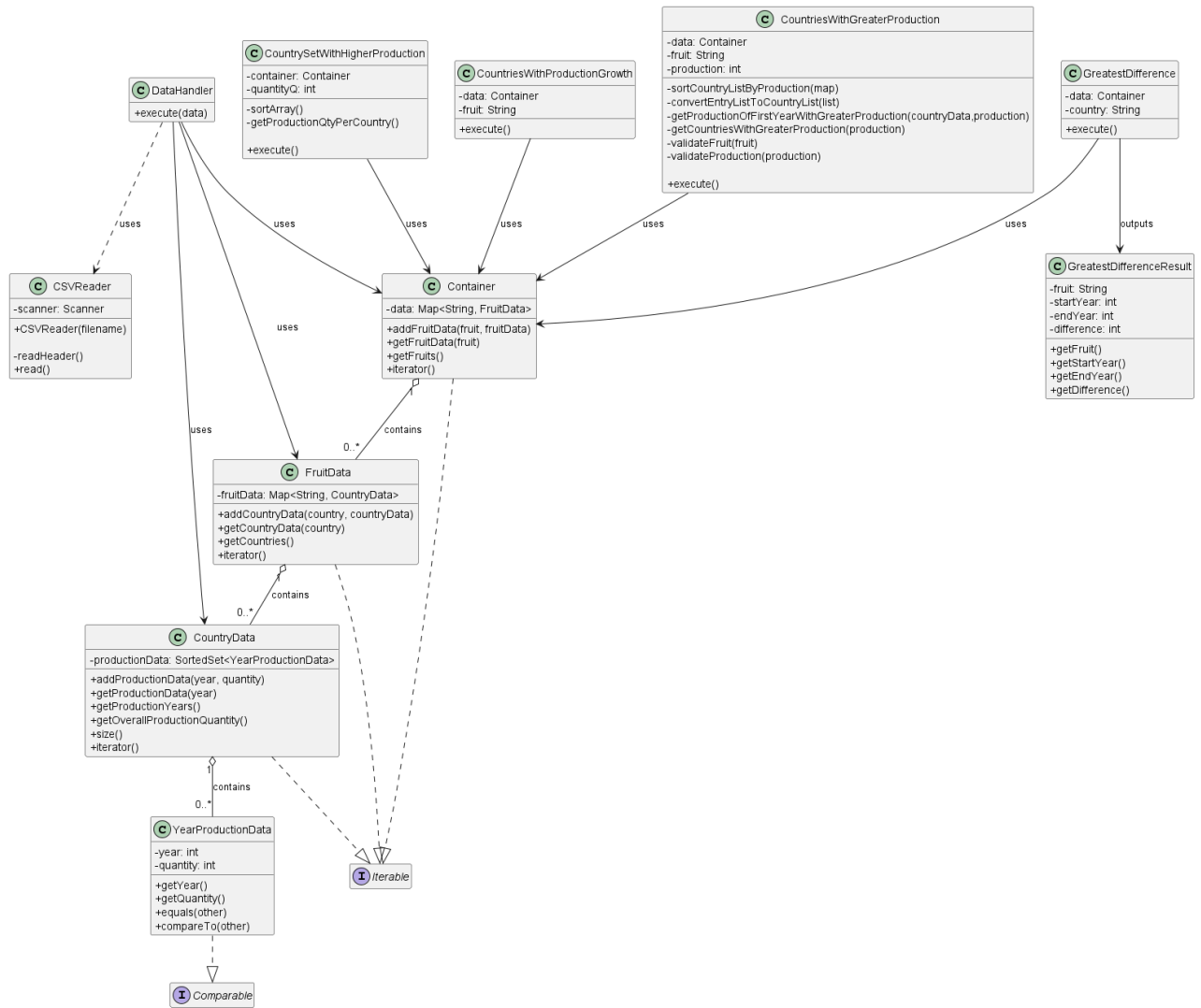


Figura 2 \_ Class Diagram completo

Todas as classes são simples contentores para as restantes exceto a classe *YearProductionData* que tem como objetivo agrupar os dados relativos à produção de cada ano num dado país ao estilo de uma *struct*. Desta forma, é mais simples obter os dados relativos às produções anuais de forma a facilitar a resolução das funcionalidades pedidas.

# Algoritmos usados

## *DataHandler*

A classe *DataHandler* é responsável por organizar os dados obtidos de ficheiros ".csv". Apesar de esta classe ser responsável pela estruturação da informação, os dados são lidos por uma classe *CSVReader* que lê um ficheiro ".csv" e retorna uma Lista de Mapas, onde cada item da lista representa um registo do ficheiro, as chaves do mapa representam o nome do campo especificado no cabeçalho do ficheiro e o valor representa o valor respetivo ao campo do cabeçalho indicado na chave.

```
public class CSVReader {  
    private Scanner scanner;  
  
    public CSVReader(String fileName) throws FileNotFoundException {  
        File file = new File(fileName);  
        scanner = new Scanner(file);  
    }  
  
    public List<? extends Map<String, String>> read() { return list; }  
}
```

Figura 3\_ CSVReader class

A classe *DataHandler* tem o método *Execute* que recebe a estrutura devolvida pelo *CSVReader* e retorna um Container com a informação estruturada.

```
public Container execute(List<? extends Map<String, String>> data)  
throws MissingFieldException { return container; }
```

Figura 4\_ Método execute de DataHandler

O método percorre todos os elementos da lista recebida como parâmetro e começa por fazer as verificações necessárias para os dados lidos previamente pelo *CSVReader*. Se o campo “*Flag*” de um registo lido do ficheiro for “M” significa que a data não há dados suficientes para fazer um registo. Se tal acontecer, o método avança para o próximo elemento da lista. Verifica também se todos os campos estão preenchidos, e, no caso de omissão de um campo, é lançada uma exceção com uma mensagem descritiva do erro. No caso do campo “*Value*” se encontrar vazio, consideramos que a quantidade produzida seja igual a 0 toneladas.

```
for (int i = 0; i < data.size(); i++) {
    CountryData cd = new CountryData();
    FruitData fd = new FruitData();

    if (data.get(i).get("Flag") != null && data.get(i).get("Flag").equals("M")) continue;

    if (data.get(i).get("Year") == null) throw new MissingFieldException("Year field is required.");
    if (data.get(i).get("Area") == null) throw new MissingFieldException("Area field is required.");
    if (data.get(i).get("Item") == null) throw new MissingFieldException("Item field is required.");

    int year = Integer.parseInt(data.get(i).get("Year"));
    String country = data.get(i).get("Area");
    String fruit = data.get(i).get("Item");

    int quantity = 0;
    if (data.get(i).get("Value") != null && !data.get(i).get("Value").equals("")) quantity =
        Integer.parseInt(data.get(i).get("Value"));
}
```

*Figura 5\_ Verificações dos dados a armazenar*

Passadas com sucesso todas as verificações, passamos para a estruturação dos dados. É necessário verificar se já existem dados referentes ao fruto ou fruto-país do elemento que estamos a tratar, pois se tal se verificar, é necessário ir buscar esses dados para adicionar os do novo registo. Pelo contrário, deve-se então criar a estrutura necessária para armazenar esses dados.



```
if (container.contains(fruit)) {
    fd = container.getFruitData(fruit);
    if (fd.contains(country)) {
        cd = fd.getCountryData(country);
        cd.addProductionData(year, quantity);
    } else {
        cd.addProductionData(year, quantity);
        fd.addCountryData(country, cd);
    }
} else {
    cd.addProductionData(year, quantity);
    fd.addCountryData(country, cd);
    container.addFruitData(fruit, fd);
}
```

*Figura 6 \_ Armazenamento dos dados de forma estruturada*

Após fazer percorrer todos itens da lista é retornado o *Container* com os dados lidos.

## *CountriesWithGreaterProduction*

A funcionalidade *CountryWithGreaterProduction* tem como objetivo obter uma lista de países que dado um certo fruto, *F*, produzam quantidade igual ou superior a uma dada quantidade *Q*.

```
private Map<String, YearProductionData> getCountriesWithGreaterProduction(int production) throws
FruitNotFoundException {
    Map<String, YearProductionData> res = new LinkedHashMap<>();

    if (data.getFruitData(fruit) == null) {
        throw new FruitNotFoundException("Invalid fruit.");
    }

    FruitData fruitData = data.getFruitData(fruit);
    Set<String> countries = fruitData.getCountries();

    for (String country : countries) {
        CountryData countryData = fruitData.getCountryData(country);
        YearProductionData firstYearProductionData = getProductionOfFirstYearWithGreaterProduction(countryData,
production);

        if (firstYearProductionData != null) res.put(country, firstYearProductionData);
    }

    return res;
}
```

Figura 7 \_ Obtenção dos dados de cada país relativos a produção

Primeiro, é verificado se o país produz o fruto *F*. Caso seja verdade, o método prossegue, caso seja falso, é lançada uma exceção.

De seguida, percorrem-se todos os países de forma a obter os dados de cada um e verificar se estão dentro dos critérios referidos.

Para verificar se um país produz uma dada quantidade equivalente ou superior à indicada, é usado o método *getProductionOfFirstYearWithGreaterProduction* que vai retornar uma instância da classe *YearProductionData*, a classe que agrupa informação relativa a ano e respetiva produção.

```
private YearProductionData getProductionOfFirstYearWithGreaterProduction(CountryData countryData, int production) {
    for (YearProductionData productionData : countryData)
        if (productionData.getQuantity() >= production) return productionData;

    return null;
}
```

Figura 8 \_ Verificação se o país produz quantidade igual ou superior a um certo *Q*

Caso a produção esteja dentro o pretendido é inserido num *LinkedHashMap*. Caso contrário, passa para o seguinte. Esta interface foi escolhida pois é necessário o país e o ano da produção para conseguirmos ordenar.

Após percorrer a lista toda de países é devolvido o *Map*.

Como foi pedido unicamente obter uma lista dos países, iremos então obter do *Map* os países e adicionar para uma Lista ordenadamente. Com isto convertemos as *keys* numa lista de *Entry*, passando depois para uma *List*.


```
private List<String> sortCountryListByProduction(Map<String, YearProductionData> map) {  
    List<Entry<String, YearProductionData>> list = new ArrayList<>(map.entrySet());  
    list.sort(Entry.comparingByValue());  
  
    return convertEntryListToCountryList(list);  
}  
  
private List<String> convertEntryListToCountryList(List<Entry<String, YearProductionData>> list) {  
    List<String> countries = new ArrayList<>();  
  
    for (Entry<String, YearProductionData> entry : list)  
        countries.add(entry.getKey());  
  
    return countries;  
}
```

Figura 9 \_ Conversão dos dados para List

## *CountriesWithProductionGrowth*

Esta funcionalidade tem como principal objetivo obter, dado um fruto  $F$ , os países agrupados pelo número máximo de anos consecutivos em que houve crescimento da quantidade de produção do fruto  $F$ .

A melhor estrutura de retorno para esta informação será um mapa que associa um inteiro a uma lista de *strings*, onde o inteiro corresponde ao número máximo de anos consecutivos onde houve crescimento da produção de  $F$ , e a lista a um *ArrayList<String>* que contém todos os países que apresentam esse número máximo de anos.

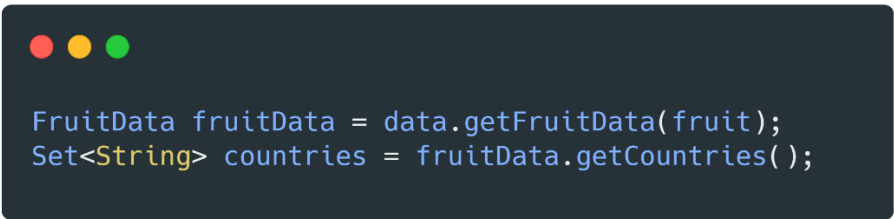


```
Map<Integer, ArrayList<String>> result = new HashMap<>();
```

Figura 10 \_ Declaração do mapa que armazena o retorno da função

Para implementar esta funcionalidade devemos, em primeiro lugar, obter os dados relativos à fruta  $F$ , representados pelo objeto *fruitData*, da classe *FruitData*. Este, por sua vez, contém um mapa onde cada país está associado aos seus dados (anos de produção e respectivas quantidades).

Obtido o objeto *fruitData*, vamos agora armazenar todos os países lá contidos num *set*. Para tal, devemos obter o *keyset* do mapa contido no objeto *fruitData*. Como as estruturas de dados estão abstraídas, esta ação está “escondida” através do método *getCountries()*, relativo à classe *FruitData*.



```
FruitData fruitData = data.getFruitData(fruit);  
Set<String> countries = fruitData.getCountries();
```

Figura 11 \_ Obtenção dos dados da fruta, bem como dos países que possuem produção da mesma

Depois, devemos iterar por todos os países contidos no *set countries*, obtendo, para cada um, os dados dos mesmos. Estes dados correspondem à classe *CountryData*, que contém a informação da quantidade de produção de cada ano.

```
for (String country : countries) {  
    CountryData countryData = fruitData.getCountryData(country);  
    ...  
}
```

Figura 12 \_ Iteração pelos países obtidos anteriormente a fim de analisar a sua produção

A classe *CountryData* possui um *sorted set* que armazena objetos da classe *YearProductionQuantity*, que guarda o ano e respetiva quantidade de produção. Como estamos a utilizar um *set* ordenado, estes objetos estarão sempre ordenados de forma ascendente relativamente ao ano.

Posto isto, podemos iterar pelos objetos contidos no objeto *countryData*, onde, garantidamente, iremos iterar por ordem ascendente e sequencial relativamente ao ano.


```
...  
for (YearProductionData productionYear : countryData) {  
    if (productionYear.getQuantity() > previousProductionQuantity) count++;  
    else count = 1;  
  
    if (count > max) max = count;  
  
    previousProductionQuantity = productionYear.getQuantity();  
}  
...
```

Figura 13 \_ Iteração de todos os objetos de *CountryData*

Tendo isto em mente, basta agora implementar um simples algoritmo para encontrar o número máximo de anos consecutivos onde houve crescimento da produção: começamos por comparar a quantidade de produção do ano atual com a quantidade do ano anterior. Se for superior, incrementamos uma variável de contagem em uma unidade, que armazena o número de anos consecutivos encontrados até então; caso seja inferior (ou igual), o contador é colocado a zero.

No final desta comparação, verificamos se o número de anos consecutivos atual (armazenado na variável contadora), é superior ao armazenado numa outra variável que guarda o número máximo de anos consecutivos. Se for maior, o número atual é colocado na variável que armazena o número máximo de anos. Repetimos este processo até iterar por todos os anos contidos no *countryData*.

Tendo encontrado o número máximo de anos, devemos agora colocar esta informação na estrutura de retorno, adicionado o país atual à lista de países que estão associados ao número máximo de anos. Se essa lista não existir, uma lista nova é criada.



```
ArrayList<String> list = result.get(max);  
if (list == null) list = new ArrayList<String>();  
  
list.add(country);
```

*Figura 14 \_ Inserção dos dados calculados no mapa de retorno*

No final, basta retornar o mapa *result*.

## *CountrySetWithHigherProduction*

A funcionalidade *CountrySetWithHigherProduction* tem como finalidade obter o número mínimo de países que, em conjunto, a sua soma de produção é maior que  $Q$ , dada uma quantidade de produção  $Q$ .

```
Map<String, Integer> productionQtyPerCountry = new HashMap<>();

for (FruitData fruitData : this.container) {
    for (String country : fruitData.getCountries()) {
        CountryData countryData = fruitData.getCountryData(country);

        if (productionQtyPerCountry.get(country) != null) {
            int newProdQty = productionQtyPerCountry.get(country) + countryData.getOverallProductionQuantity();
            productionQtyPerCountry.put(country, newProdQty);
        } else {
            productionQtyPerCountry.put(country, countryData.getOverallProductionQuantity());
        }
    }
}
```

*Figura 15 \_ Obtenção das quantidades de produção de cada fruta por país*

Como foi dito, a solução não é perfeita para todas as situações, como é o caso desta funcionalidade. A solução encontrada para as estruturas de dados não favoreceu a consulta de dados pois foi necessário percorrer todas as frutas e, para cada país, somar as quantidades de produção de cada um em todos os anos registados. Desta forma, obtém-se um *Map* em que a cada país (chave) corresponde uma quantidade total de produção (valor).

Como não é necessário obter uma lista de países para esta funcionalidade e para facilitar o processamento, converteu-se os valores do *Map* para um *array* de inteiros.

```
return productionQtyPerCountry.values().stream().mapToInt(i -> i).toArray();
```

*Figura 16 \_ Conversão dos valores do Map para array de inteiros*

Com o *array* de inteiros podemos agora efetuar o cálculo. Como só necessitamos de encontrar o mínimo número de países, podemos ordenar o *array* e iterá-lo iniciando pela maior quantidade de produção e somando estas quantidades – desta forma é garantido que teremos o mínimo número de países que excedem  $Q$ . Quando for encontrado um total maior que  $Q$ , é retornado o número de iterações que foram efetuadas.

```
if (array.length <= 1)
    return original;

int mid = array.length / 2;

int[] left = sort(Arrays.copyOfRange(array, 0, mid));
int[] right = sort(Arrays.copyOfRange(array, mid, array.length));

return merge(left, right);
```

Figura 17 \_ Algoritmo de ordenação "Merge Sort"

```
private static int[] merge(int[] left, int[] right) {
    int length = left.length + right.length;
    int[] merged = new int[length];
    int i = 0, j = 0;

    while (i < left.length && j < right.length) {
        if (left[i] < right[j]) {
            merged[i + j] = left[i++];
        } else {
            merged[i + j] = right[j++];
        }
    }

    while (i < left.length) {
        merged[i + j] = left[i++];
    }
    while (j < right.length) {
        merged[i + j] = right[j++];
    }

    return merged;
}
```


Figura 18 \_ Algoritmo de ordenação "Merge Sort" (cont.)



Vale notar a importância de ter o *array* ordenado: desta forma, garante-se que ao somar de forma decrescente, o número de iterações efetuadas para verificar se a soma de  $n$  quantidades de produção de diferentes países é maior que  $Q$  será o mínimo número de países necessários para que, em conjunto, seja possível exceder  $Q$ , evitando desta forma o método de procura de forma “bruta”.

O algoritmo escolhido para ordenar o *array* foi o “*Merge Sort*”. Sabendo que o programa deve suportar o processamento de grandes “datasets”, foi importante implementar um algoritmo eficiente, com uma complexidade no pior caso de  $O(n \log n)$ .

No caso de não ser possível encontrar um conjunto de países cuja soma é superior a  $Q$ , a função retorna -1.



```
int total = 0;
for (int i = productionQtyPerCountry.length - 1; i >= 0; i--) {
    total += productionQtyPerCountry[i];

    if (total > quantityQ) return productionQtyPerCountry.length - i;
}
```

Figura 19\_ Cálculo do número mínimo de países que em conjunto excedem  $Q$

## *GreatestDifferenceInProduction*

Nesta funcionalidade, dado um país *P* devemos descobrir a maior diferença de produção entre dois anos consecutivos.

Deste modo, temos de percorrer todos os frutos, mas graças ao uso de mapas podemos obter as informações dos países em tempo constante. Desta forma, este algoritmo é executado em tempo quadrático devido a ser necessário percorrer também todos os registos de cada país.

No caso de não ser possível computar um valor válido para a maior diferença entre produções do país, então é retornado o valor *null*.

```
for (String fruit : data.getFruits()) {
    FruitData fruitData = data.getFruitData(fruit);
    CountryData countryProductionData = fruitData.getCountryData(country);

    if (countryProductionData == null) continue;

    for (Integer productionYear : countryProductionData.getProductionYears()) {
        int production = countryProductionData.getProductionData(productionYear);
        Integer nextProduction = countryProductionData.getProductionData(productionYear + 1);

        if (nextProduction != null) {
            int difference = Math.abs(production - nextProduction);

            if (difference > maxDifference) {
                maxDifference = difference;
                maxDifferenceFruit = fruit;
                maxDifferenceStartYear = productionYear;
                maxDifferenceEndYear = productionYear + 1;
            }
        }
    }
}
```

Figura 20 \_ Algoritmo para descobrir a maior diferença de produção entre dois anos consecutivos

No final é retornada uma classe que agrupa todos estes atributos. Como todas as produções são positivas, a variável *maxDifference* é inicializada a -1, se este valor não se alterar então não foi possível computar uma diferença válida.

```
if (maxDifference == -1) return null;
return new GreatestDifferenceResult(maxDifferenceFruit, maxDifferenceStartYear, maxDifferenceEndYear, maxDifference);
```

Figura 21 \_ Condições de retorno do algoritmo da figura anterior

## Possíveis melhorias

Como referido ao longo deste documento, a estrutura adotada não é a ideal para todas as funcionalidades. Com o uso de mapas foi possível melhorar o tempo de execução de muitas dos casos de uso, mas dependendo da quantidade de dados inseridos os tempos de execução podem parar de ser ideais. Por exemplo, no caso de existirem muitos países e frutas e poucos dados sobre as quantidades de produção, a estrutura acabaria por apresentar uma performance muito inferior à que apresentaria se a estrutura escolhida fosse invertida (o primeiro mapa referir-se a anos de produção em vez de frutas).

O uso de mapas com recursos a estruturas em árvore também poderia ter sido mais explorado de forma a reduzir, em alguns casos, tempos de execução lineares para logarítmicos.

Para além disso, neste momento só temos 4 dados relevantes para cumprir os requisitos (Fruta, País, Ano, Quantidade Produção). Uma possível melhoria seria criar classes mais complexas que guardariam dados adicionais para obtermos funcionalidades mais completas, como por exemplo, as unidades da produção ou o código da área FAO (Food Agriculture Organization), podendo assim, converter unidades e organizar por áreas.

Uma outra possível melhoria seria desenvolver uma interface simples e intuitiva de forma a tornar a aplicação operacional, visto que o trabalho se focou nas estruturas de informação adequadas para o suporte da mesma.