UNIVERSIDADE DE ÉVORA

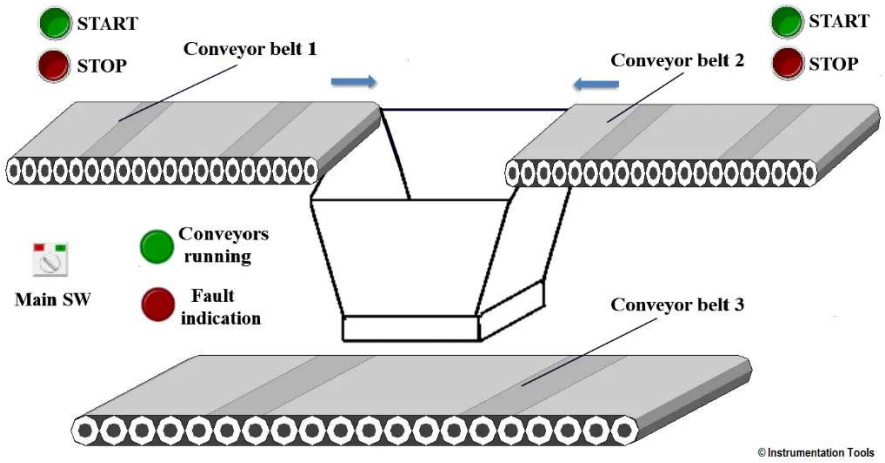
Relatório - 1º Trabalho Prático

Estrutura de Dados e Algoritmos II

Engenharia Informática

2022 - 2023

Problema: Conveyor Belts



Trabalho Realizado por:

- Daniel Barreiros nº48452

- Tomás Antunes nº48511

- Grupo do Mooshak: g121

Índice

I - Introdução

O presente relatório apresenta todos os detalhes referentes à resolução do problema "Conveyor Belts". Para a resolução deste problema procurámos obter uma solução que fosse suficientemente eficiente por forma a cumprir os parâmetros impostos pelo professor através da plataforma Mooshak bem como através da qual nos fosse possível aplicar alguma matéria lecionada nesta disciplina.

O problema apresentado consiste em dois tapetes rolantes nos quais se encontram produtos de diversos tipos e valores. É importante destacar que os produtos do mesmo tipo podem formar pares enquanto os de diferentes tipos não se podem agrupar. Através do rodar de cada tapete, todos os produtos que caírem e não se agruparem não irão ter qualquer tipo de valor. Assim, o objetivo da solução desenvolvida consiste na obtenção da melhor sequencia para rodar cada tapete por forma a que se criem os pares mais valiosos.

O critério final de decisão consiste em adquirir o valor máximo possível e o menor número de pares para obter esse valor.

II - Resolução do Problema

2.1 – Leitura do Input

Após a leitura e interpretação do enunciado começámos por realizar a leitura de um dos inputs que nos foi dado por forma a que o nosso programa conseguisse obter os dados necessários para posteriormente seguir para a sua resolução.

Através da interpretação do seguinte exemplo de input (ligeiramente alterado comparativamente com o do enunciado) realizamos o seguinte esquema:

Input:

1

4

nail B 5000

spoon A 1200

orange C 5

nail B 50

3

fork A 50000

hammer B 10

apple C 600

Output:

51805 2

Esquema:

VALOR NºPARES	0 0	50000 A	10 B	600 C
0 0	0 0	0 0	0 0	0 0
5000 B	0 0	0 0	5010 1	5010 1
1200 A	0 0	51200 1	51200 1	51200 1
5 C	0 0	51200 1	51200 1	51805 2
50 B	0 0	51200 1	51260 2	**51805** **2**

Para obtermos este esquema, utilizamos como base o algoritmo "maior subsequência comum" lecionado pelo professor na aula teórica por forma a maximizar o valor dos pares escolhidos e minimizar o número de pares para obter o valor máximo.

2.2 – Função Recursiva

Através da análise deste esquema conseguimos elaborar as seguintes funções recursivas referentes:

- À maximização dos valores retirados dos produtos dos tapetes:

$tapete_1 \rightarrow$ vetor com o valor dos elementos do tapete1;

$tapete_2 \rightarrow$ vetor com o valor dos elementos do tapete2;

$v_{xy}(i,j) \rightarrow$ valor máximo obtido com a soma dos valores retirados dos dois tapetes;

z $\rightarrow v_{xy}(i-1,j-1) + tapete_1(i) + tapete_2(j)$;

$$\begin{cases} 0 \; se \; i = 0 \; \vee j = 0 \\ z \; se \; x_i = y_j \; \wedge z > v_{xy}(i-1,j) \; \wedge z > v_{xy}(i,j-1) \; \wedge \; i,j > 0 \\ v_{xy}(i-1,j) \; se \; x_i = y_j \; \wedge z < v_{xy}(i-1,j) \; \wedge v_{xy}(i-1,j) \geq v_{xy}(i,j-1) \; \wedge \; i,j > 0 \\ v_{xy}(i,j-1) \; se \; x_i = y_j \; \wedge z < v_{xy}(i,j-1) \; \wedge v_{xy}(i-1,j) < v_{xy}(i,j-1) \; \wedge i,j > 0 \\ \max\{v_{xy}(i-1,j), v_{xy}(i,j-1)\} \quad se \; x_i \neq y_j \; \wedge i,j > 0 \end{cases}$$

- À minimização dos pares formados para conseguirmos obter o maior valor possível:

$P_{xy}(i,j) \rightarrow$ número de pares necessário para obter o valor máximo

$$\begin{cases} 0 \; se \; i = 0 \; \vee j = 0 \\ 1 + P_{xy}(i-1,j-1) \; se \; x_i = y_j \; \wedge z > v_{xy}(i-1,j) \wedge z > v_{xy}(i,j-1) \\ P_{xy}(i,j-1) \; se \; \Big(x_i = y_j \; \wedge \Big(z < \; v_{xy}(i-1,j) \; \vee \; z < v_{xy}(i,j-1)\Big) \wedge v_{xy}(i-1,j) = v_{xy}(i,j-1) \; \wedge OP < P_{xy}(i-1,j)\Big) \vee \\ \Big(x_i = y_j \; \wedge \Big(z < \; v_{xy}(i-1,j) \; \vee \; z < v_{xy}(i,j-1)\Big) \; \wedge v_{xy}(i,j-1) > v_{xy}(i-1,j)\Big) \vee \\ \big(x_i \neq y_j \; \wedge v_{xy}(i-1,j) = v_{xy}(i,j-1) \; \wedge P_{xy}(i,j-1) < P_{xy}(i-1,j)\big) \vee \\ \big(x_i \neq y_j \; \wedge v_{xy}(i,j-1) > v_{xy}(i-1,j)\big) \\ P_{xy}(i-1,j) \; se \; \Big(x_i = y_j \; \wedge \Big(z < \; v_{xy}(i-1,j) \; \vee \; z < v_{xy}(i,j-1)\Big) \wedge v_{xy}(i-1,j) \geq v_{xy}(i,j-1) \; \wedge P_{xy}(i-1,j) < OP\Big) \vee \\ \big(x_i \neq y_j \; \wedge v_{xy}(i-1,j) = v_{xy}(i,j-1) \; \wedge P_{xy}(i-1,j) < P_{xy}(i,j-1)\big) \vee \\ \big(x_i \neq y_j \; \wedge v_{xy}(i-1,j) > v_{xy}(i,j-1)\big) \end{cases}$$

III – Explicação do Código

3.1 – Classe Main

```java
class Main {
    public static void main(String[] args) throws NumberFormatException, IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        int cases = Integer.parseInt(input.readLine());

        for(; cases > 0; cases--) {
            // first belt
            int productsFirst = Integer.parseInt(input.readLine());
            String[] firstBelt = new String[productsFirst];

            for(int i = 0; i < productsFirst; i++)
                firstBelt[i] = input.readLine();

            // second belt
            int productsSecond = Integer.parseInt(input.readLine());
            String[] secondBelt = new String[productsSecond];

            for(int j = 0; j < productsSecond; j++)
                secondBelt[j] = input.readLine();

            Belt belt = new Belt(productsFirst, productsSecond);
            belt.setProducts(1, firstBelt);
            belt.setProducts(2, secondBelt);

            belt.beltPairs();
        }
    }
}
```

Inicialmente declaramos a classe main. Este método realiza o throw de duas exceções, sendo estas:

NumberFormatException – Esta exceção irá ocorrer se tentarmos converter uma String para um tipo numérico, mas a string não está no formato apropriado.

IOException – Esta exceção irá ocorrer no caso de o input introduzido não ser o esperado pelo programa. Um possível exemplo seria o programa estar à espera de um input do tipo inteiro e acaba por receber um input do tipo string. Desta forma, a exceção será levantada.

Para realizarmos a leitura do input declaramos um novo objeto da classe BufferedReader que irá utilizar uma nova instância da classe InputStreamReader que vai ler o input em System.in.

O primeiro valor a ser lido corresponde ao número de casos que irão ser avaliados. Guardamos este valor no inteiro “cases”.

Utilizando este número de casos, corremos um *for* por forma a realizar a leitura dos dados de todos os casos que irão ser estudados.

Dentro deste ciclo realizamos a leitura do número de produtos que estão no primeiro tapete e guardamos este valor no inteiro “productsFirst”. Quando realizamos este tipo de leitura, utilizamos o método parseInt() da classe Integer por forma a guardar o valor do tipo inteiro.

Criamos depois o array de Strings “firstBelt”, de tamanho correspondente ao número de produtos obtido acima (productsFirst) por forma a podermos utilizá-lo para guardar a informação de cada produto deste primeiro tapete.

Para o segundo tapete, voltamos a repetir o mesmo processo por forma a realizarmos a leitura e o armazenamento dos dados.

Assim, após realizarmos a leitura de todos os dados, criamos um objeto da classe Belt no qual fornecemos como parâmetros o número de produtos do primeiro tapete e do segundo tapete. Através do método setProduct() da classe Belt completamos os arrays correspondentes a cada tapete com os tipos e valores correspondentes a cada produto.

No final, invocamos o método beltPairs() da classe Belt que vai calcular o maior valor possível que se pode retirar dos tapetes e o menor numero de pares necessários para alcançar esse valor (Explicaremos detalhadamente este método mais à frente no relatório).

3.2 – Classe Belt

```
class Belt {
    // first belt
    int productsFirst;
    String[] typeFirst;
    int[] valueFirst;

    // second belt
    int productsSecond;
    String[] typeSecond;
    int[] valueSecond;

    long[][] maxValues;
    int[][] minPairs;

    public Belt(int productsFirst, int productsSecond) {
        this.productsFirst = productsFirst;
        typeFirst = new String[productsFirst];
        valueFirst = new int[productsFirst];

        this.productsSecond = productsSecond;
        typeSecond = new String[productsSecond];
        valueSecond = new int[productsSecond];

        maxValues = new long[productsFirst + 1][productsSecond + 1];
        minPairs = new int[productsFirst + 1][productsSecond + 1];
    }
```

3.2.1 – Variáveis de Classe

Na classe Belt inicializamos as seguintes variáveis de classe:

int productsFirst – irá ficar com o número de produtos existentes no primeiro tapete;

String[] typeFirst - irá guardar em cada posição o tipo do produto, sendo a primeira posição correspondente ao tipo do primeiro produto, a segunda posição correspondente ao tipo do segundo produto…(Importante realçar que este array corresponde aos tipos de produtos do primeiro tapete).

int[] valueFirst – irá guardar em cada posição o valor de cada produto, sendo a primeira posição correspondente ao valor do primeiro produto, a segunda posição correspondente ao valor do segundo produto…..(Importante realçar que este array corresponde aos valores dos produtos do primeiro tapete).

Através da observação do código podemos verificar que as três variáveis que se encontram declaradas posteriormente são do mesmo género que estas primeiras três, contudo, são referentes aos produtos do segundo tapete (int productSecond, String[] typeSecond, int[] valueSecond).

long[][]maxValues – o tamanho desta matriz irá ser determinado pelo número de produtos de cada tapete +1, sendo o seu número de linhas correspondente ao número de produtos do primeiro tapete e o número de colunas correspondente ao número de produtos do segundo tapete. Este array irá guardar em cada posição o valor correspondente à soma dos valores correspondentes aos pares que até a essa posição foi possível criar. (Iremos explicar mais detalhadamente como este array será preenchido mais à frente no relatório).

int[][]maxPairs – o tamanho deste array bidimensional irá ser determinado da mesma forma que no array maxValues. Este array irá guardar em cada posição o número de pares que até a essa posição foi possível criar. (Iremos explicar de uma forma mais detalhada como este array é preenchido mais à frente no relatório).

3.2.2 – Construtor da Classe Belt

Após criarmos estas variáveis de classe seguimos para a criação do construtor da classe Belt.

Este construtor recebe como argumentos os inteiros productsFirst e productsSecond que correspondem ao número de produtos do primeiro tapete e ao número de produtos do segundo tapete respetivamente.

São criados os seguintes arrays:

typeFirst – o tamanho deste array será determinado pelo número de produtos existentes no primeiro tapete (productsFirst), e irá guardar o tipo de cada produto (Já explicado acima).

valueFirst – o tamanho deste array será determinado pelo número de produtos existentes no primeiro tapete (productsFirst), e irá guardar o valor de cada produto (Já explicado acima).

O mesmo processo é realizado para o typeSecond e valueSecond, a única diferença corresponde a que o seu tamanho, os valores, e os tipos são correspondentes aos produtos do segundo tapete.

É criada uma matriz maxValues que irá ter o seu tamanho determinado pelo número de produtos do primeiro tapete (linhas) +1 e do segundo tapete (colunas) +1. Este +1 é necessário para poder existir o espaço correspondente ao caso base. Este caso base poderá corresponder ao primeiro tapete ter produtos e o segundo não ter ou ao caso de o segundo tapete ter produtos e o primeiro não ou ao caso de nenhum dos tapetes ter produtos.

Assim, a primeira linha e a primeira coluna irão ser preenchidas pelo número 0, visto que não é possível formar nenhum par só com a existência de um tapete com elementos. Com a não formação de par, o valor da sua soma (que acaba por não existir) irá ser 0.

É criada uma matriz minPairs que, da mesma forma que no array maxValues, vai ter o seu tamanho determinado pelo número de produtos existentes em cada tapete +1. O motivo pelo qual o tamanho tem de ser +1 é exatamente o mesmo pelo explicado acima. É necessário essa linha e essa coluna para poder guardar o caso base.

3.2.3 – Método setProducts

```
public void setProducts(int belt, String[] products) {
    if(belt == 1) { // first belt
        for(int i = 0; i < productsFirst; i++) {
            String[] product = products[i].split(" ");
            typeFirst[i] = product[1];
            valueFirst[i] = Integer.parseInt(product[2]);
        }
    } else { // second belt
        for(int i = 0; i < productsSecond; i++) {
            String[] product = products[i].split(" ");
            typeSecond[i] = product[1];
            valueSecond[i] = Integer.parseInt(product[2]);
        }
    }
}
```

O método setProducts recebe como argumentos:

int belt – irá corresponder ao número do tapete, se o primeiro tapete (1) se o segundo tapete (2).

String[] products – corresponde ao array que contém os produtos de um dos tapetes.

Primeiramente é utilizado um *if* para verificar se os produtos serão correspondentes ao primeiro tapete ou ao segundo tapete.

Após esta verificação será iniciado um *for* que irá correr o número de vezes correspondente ao número de produtos desse mesmo tapete. Ou seja, no caso de ser o primeiro tapete o *for* irá ter o número de ciclo igual ao número de produtos do primeiro tapete.

Dentro deste ciclo, através do método split(), guardamos num array temporário (String[] product) o nome, o tipo e o valor do produto (cada ciclo irá corresponder a um produto do tapete), visto que, no array products, cada posição irá ter uma string (corresponde ao nome do produto, que para o exercício não tem interesse) seguida de outra string (que corresponde ao tipo do produto) e um inteiro (que corresponde ao valor desse produto).

Com a informação armazenada temporariamente neste array, no qual a informação irá ficar com o nome em product[0], o tipo em product[1] e o valor em product[2] podemos guardar a informação em arrays separados, sendo que o tipo irá para o array typeFirst e o valor irá para o array valueFirst (No caso de o produto pertencer ao primeiro tapete). No caso de os produtos pertencerem ao segundo tapete irão ser guardados no array typeSecond e valueSecond respetivamente.

3.2.4 – Método beltPairs

```
public Result beltPairs() {
    for(int i = 0; i <= productsFirst; i++) {
        minPairs[i][0] = 0;
        maxValues[i][0] = 0;
    }

    for(int j = 0; j <= productsSecond; j++) {
        minPairs[0][j] = 0;
        maxValues[0][j] = 0;
    }
```

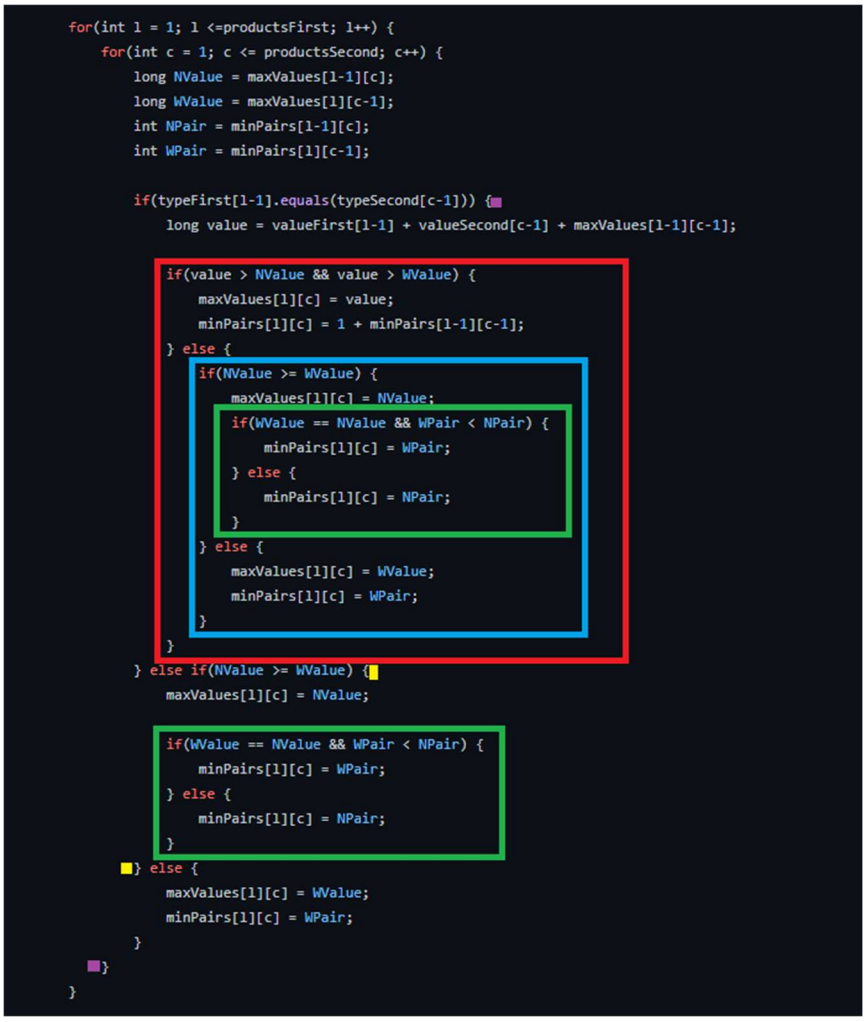
Inicialmente é executado um *for* que irá ter o seu número de ciclos igual ao número de produtos correspondentes ao primeiro tapete +1. Neste caso, executamos o ciclo até o valor ser igual ao número de produtos por forma a que exista mais um ciclo que o número de produtos.

Este ciclo tem como objetivo preencher o caso base (0) na matriz que irá guardar o número de pares bem como preencher o caso base na matriz que irá guardar o valor da soma dos produtos quando ocorre a formação de pares.

Assim, com este ciclo, preenchemos a primeira linha com o valor 0 em todas as posições. Importante esclarecer que este *for* preenche a primeira linha do array dos pares, bem como a primeira linha do array dos valores (minPairs e maxValues respetivamente).

Da mesma forma, executamos outro *for* que irá preencher o caso base em ambos os arrays, mas ao invés de preencher a primeira linha irá preencher a primeira coluna com o valor 0 em todas as posições.

Assim, através destes dois ciclos, preenchemos a primeira linha e a primeira coluna de ambos os arrays (minPairs e maxValues) com o valor 0.



```
for(int l = 1; l <=productsFirst; l++) {
    for(int c = 1; c <= productsSecond; c++) {
        long NValue = maxValues[l-1][c];
        long WValue = maxValues[l][c-1];
        int NPair = minPairs[l-1][c];
        int WPair = minPairs[l][c-1];

        if(typeFirst[l-1].equals(typeSecond[c-1])) {
            long value = valueFirst[l-1] + valueSecond[c-1] + maxValues[l-1][c-1];

            if(value > NValue && value > WValue) {
                maxValues[l][c] = value;
                minPairs[l][c] = 1 + minPairs[l-1][c-1];
            } else {
                if(NValue >= WValue) {
                    maxValues[l][c] = NValue;
                    if(WValue == NValue && WPair < NPair) {
                        minPairs[l][c] = WPair;
                    } else {
                        minPairs[l][c] = NPair;
                    }
                } else {
                    maxValues[l][c] = WValue;
                    minPairs[l][c] = WPair;
                }
            }
        } else if(NValue >= WValue) {
            maxValues[l][c] = NValue;

            if(WValue == NValue && WPair < NPair) {
                minPairs[l][c] = WPair;
            } else {
                minPairs[l][c] = NPair;
            }
        } else {
            maxValues[l][c] = WValue;
            minPairs[l][c] = WPair;
        }
    }
}
```

Neste troço de código é onde se encontram as decisões fundamentais de preenchimento dos arrays por forma a conseguirmos obter os resultados corretos para o problema até agora estudado.

Iremos explicar passo por passo de cada decisão por forma a tornar o mais claro possível a compreensão da solução implementada.

Por forma a correrrmos as matrizes de modo a preenchermos o seu conteúdo, iniciamos um ciclo que irá ter início no número 1 e fim no número total de produtos do primeiro tapete (Irá percorrer as linhas) seguido de outro ciclo que irá começar do número 1 e terminará no número final de produtos do segundo tapete (Irá percorrer as colunas). Decidimos começar os ciclos no número 1 sendo que a primeira linha e a primeira coluna já estão preenchidas, e como precisamos dos valores anteriores à posição atual, não faria sentido começar no 0, pois a posição -1 não existe.

Criámos 4 variáveis temporárias que irão obter, em cada ciclo, valores que nos são essenciais para realizar as decisões e alguns cálculos. Decidimos utilizar estas variáveis temporárias pois o código fica mais limpo, levando assim a uma melhor compreensão do mesmo.

Long NValue – corresponde ao valor do produto que se encontra a norte da posição que estamos a estudar.

Long Wvalue – corresponde ao valor do produto que se encontra a oeste da posição que estamos a estudar.

int NPair – corresponde ao número de pares que foi possível criar até à posição que se encontra a norte da posição que estamos a estudar.

int WPair – corresponde ao número de pares que foi possível criar até à posição que se encontra a oeste da posição que estamos a estudar.

Realizamos a primeira verificação, na qual comparamos se o tipo do produto do primeiro tapete é igual ao tipo do produto do segundo tapete. Ou seja, estamos a verificar se os produtos são do mesmo tipo por forma a poderem formar pares de produtos ou não.

No caso desta condição **ser positiva**, ou seja, os produtos serem do mesmo tipo podendo formar um par de produtos vamos primeiramente inicializar uma variável temporária (long value) que irá armazenar a soma dos valores dos produtos do par com o valor máximo até aí calculado com a formação dos pares anteriores (Este valor encontra-se na posição noroeste (l-1, c-1)).

Após obtermos este valor verificamos se a escolha do par é a melhor opção tendo em conta os valores já calculados a norte e a oeste. No caso de este valor ser o maior guardamo-lo na posição atual da matriz e incrementamos um ao valor dos pares até aí formados. O número de pares até então formados encontra-se na posição noroeste. No caso de a soma do par não ser a melhor opção verificamos se o valor norte é maior ou igual ao valor oeste, e, caso se verifique, guardamos o valor norte na matriz maxValues. Para escolher o menor número de pares verificamos se o valor norte é igual ao valor oeste e se o número de pares a oeste é menor que o número de pares a norte, escolhendo e guardando assim o número a oeste na posição atual da matriz minPairs. No caso de os valores norte e oeste não serem iguais ou o número de pares norte ser menor que o número de pares a oeste escolhemos e guardamos o número de pares norte nesta mesma matriz. No caso de o valor oeste ser maior que o valor norte guardamos o valor oeste e o número de pares oeste nas suas matrizes correspondentes (maxValues, minPairs).

Se os produtos não forem do mesmo tipo (**Não se verifica a primeira condição**) iremos verificar se o valor que se encontra a norte da posição atual é maior ou igual ao do valor que se encontra a oeste da posição atual (Realizamos esta verificação pois queremos maximizar o valor). No caso afirmativo, guardamos o valor que se encontra a norte na posição atual (Na matriz maxValues) e vamos verificar se, no caso dos valores a norte e a oeste da posição atual serem iguais, qual é o que detêm menor número de pares (Pois a nossa solução quer encontrar o maior valor possível com o menor número de pares).

No caso de o valor ser igual e o menor número de pares ser o que esta a oeste, guardamos este número na posição atual da matriz (minPairs).

No caso negativo, significa que o número de pares mais pequeno se encontra na posição a norte ou o valor norte é estritamente maior que o valor oeste, logo, guardamos o número de pares que se encontra na posição a norte da posição atual.

Já se o valor que se encontra a norte da posição atual for menor que o da posição a oeste, passamos para o último else, que irá simplesmente guardar na posição atual de ambas as matrizes, os valores que se encontram a oeste da posição atual.

Assim, no final da execução destes ciclos, ficamos com ambos os arrays preenchidos.

```
        return new Result(maxValues[productsFirst][productsSecond], minPairs[productsFirst][productsSecond]);
    }
}
```

No final deste método realizamos o return do valor máximo obtido e do menor número de pares necessários para obter este valor. Estes encontram-se na posição final de ambas as matrizes.

3.3 - Classe Result

```
class Result {
    public Result(long maxValue, int minPair) {
        System.out.print(maxValue + " " + minPair + "\n");
    }
}
```

A classe Result é apenas composta pelo seu construtor. Recebe como argumentos o valor máximo obtido e o menor número de pares utilizado para obter esse valor.

Deste modo, este método realiza o print de acordo com o formato pedido no enunciado.

IV – Cálculo de Complexidade

4.1 - Complexidade Temporal

A complexidade temporal consiste no tempo que um determinado programa necessita para completar uma tarefa da forma correta.

$|F|$ - número de produtos do primeiro tapete (productsFirst).

$|S|$ - número de produtos do segundo tapete (productsSecond).

A linha 9 tem uma complexidade temporal de $O(1)$.

Das linhas 11 – 31 podemos considerar que temos uma complexidade de $O(cases*|F|*|S|)$.

A linha 26 tem uma complexidade de $O(|F|*|S|)$, pois executa o método das linhas 49 – 60 (Construtor da classe Belt).

A linha 27 tem uma complexidade de $O(|F|)$ pois esta chama o método setProducts que vai fazer uso de um ciclo com complexidade de $O(|F|)$.

A linha 28 tem uma complexidade de $O(|S|)$ pois esta chama o método setProducts que vai fazer uso de um ciclo com complexidade de $O(|S|)$.

A linha 30 tem uma complexidade de $O(|F|*|S|)$ pois executa o método beltPairs() que vai fazer uso de 4 ciclos. O primeiro tem uma complexidade de $O(|F|)$, o segundo tem uma complexidade de $O(|S|)$, e os restantes dois que estão contidos um no outro têm uma complexidade de $O(|F|*|S|)$.

Das linhas 49 – 60 podemos considerar que temos uma complexidade de $O(|F|*|S|)$ pois estamos a inicializar arrays de tamanho $|F|$ e $|S|$ e inicializamos duas matrizes em que a complexidade é $O((|F|+1)*(|S|+1))$ o que pode ser simplificado para $O(|F|*|S|)$.

Das linhas 64 – 68 temos uma complexidade de $O(|F|)$, e das linhas 70 – 74 temos uma complexidade de $O(|S|)$.

Das linhas 79 – 82 temos uma complexidade de $O(|F|)$, e das linhas 84 – 87 temos uma complexidade de $O(|S|)$.

Das linhas 89 – 128 temos uma complexidade de $O(|F|*|S|)$, pois trata-se de um ciclo contido noutro em que um vai ser executado $|F|$ vezes e o segundo $|S|$ vezes. Dentro destes ciclos são realizadas varias operações e todas tem uma complexidade de $O(1)$.

A linha 130 apresenta uma complexidade de $O(1)$ pois executa um método (Construtor da classe Result) que tem uma complexidade de $O(1)$.

Da linha 134 – 137 têm uma complexidade $O(1)$ pois a operação executada é um print (que pode ser considerado ter uma complexidade constante).

Tendo em conta as complexidades temporais obtidas a complexidade final do programa será $O(cases*|F|*|S|)$.

4.2 – Complexidade Espacial

A complexidade espacial consiste na quantidade de memória necessária para realizar a execução de um programa ou algoritmo.

$|F|$ - número de produtos do primeiro tapete (productsFirst).

$|S|$ - número de produtos do segundo tapete (productsSecond).

A linha 9,13, 20 têm uma complexidade de *O(1),* pois criam apenas uma variável do tipo int.

Na linha 14 é alocada memória para um array de tamanho $|F|$, logo a complexidade espacial desta operação é $O(|F|)$. Na linha 21 é alocada memória para um array de tamanho $|S|$, logo a complexidade espacial desta operação é $O(|S|)$.

A linha 26 tem uma complexidade de $O(|F|*|S|)$ pois executa um método (Construtor da classe Belt) que tem complexidade espacial de $O(|F|*|S|)$.

Da linha 49 – 60 tem uma complexidade de $O(|F|*|S|)$. É alocada memória para duas variáveis levando assim a que tenha uma complexidade constante de *O(1)*, são também criados quatro arrays, sendo que, dois deles têm uma complexidade de $O(|F|)$ e os restantes dois tem uma complexidade de $O(|S|)$. Por fim são criadas duas matrizes com tamanho |F|+1 por |S|+1 o que apresenta uma complexidade espacial de $O((|F|+1)*(|S|+1))$, o que pode ser simplificado para $O(|S|*|F|)$. Podendo assim calcular a complexidade espacial deste método da seguinte forma: $O(1) + O(1) + O(|F|) + O(|S|) + O(|F|*|S|) + O(|F|*|S|) = O(|F|*|S|)$.

As linhas 65 e 71 têm uma complexidade de O(n+k), sendo n o tamanho da i-ésima String presente no array products e k o numero de tokens resultantes do split feito a essa mesma string.

As linhas 66, 67, 72, 73 e 91 - 94 apresentam uma complexidade espacial de *O(1)* pois estas apenas alocam espaço na memória para uma variável.

Tendo em conta as complexidades espaciais obtidas a complexidade final do programa será $O(|F|*|S|)$.

V – Conclusão

Através da resolução deste problema foi-nos possível aplicar parte da matéria lecionada na disciplina, bem como desenvolver as nossas capacidades na resolução de problemas e aplicação de algoritmos.

Pelo nosso ponto de vista, consideramos que a dificuldade do problema residia na correta interpretação do enunciado, mais concretamente, no facto de que a escolha de realizar um par poderia não ser a melhor opção tendo em conta as sequências calculadas anteriormente.

Após algumas tentativas e o esclarecimento de dúvidas com o professor, conseguimos resolver o problema e aplicar o algoritmo necessário de forma eficiente e adequada ao pretendido.

Consideramos que, apesar das dificuldades encontradas, a resolução deste tipo de problemas é algo fundamental para o nosso desenvolvimento a níveis académicos e profissionais pois dá-nos uma melhor perceção e fornece-nos ferramentas importantes para desafios futuros.