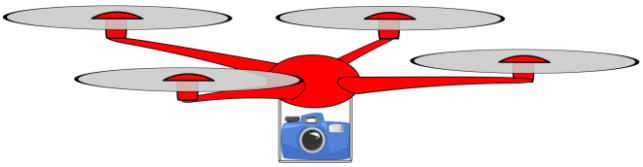
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA



DRINA'S DRONE

PROJETO 1 ADA

Francisco Mateus (53270) Vasco Martins (53299)

Francisco Mateus (53270) Vasco Martins (53299)

EQUAÇÃO

Seja N o número de ofertas, existem O_1, O_2,…,O_i ofertas tal que 1≤i≤N e existe uma lista, *offers*, cujo conteúdo é composto pelas ofertas e cujo tamanho equivale a N.

C(i, j) verifica se Oi e Oj são compatíveis.

$$\mathbf{maxProfit}(\boldsymbol{i}) = \begin{cases} 0\,, & \mathrm{i} = 0 \\ \underset{1 \le \mathrm{k} < \mathrm{i} \wedge \mathrm{c(i,j)}}{\boldsymbol{max}} (\mathrm{m(i,k)},\ \mathrm{maxProfit}(i-1)), & \mathrm{i} \ge 1 \end{cases}$$

$$m(i,k) = \begin{cases} Pi, & maxProfit(k) \le maxProfit(k-1) \\ Pi + maxProfit(k), & maxProfit(k) > maxProfit(k-1) \end{cases}$$

$$c(i,j) = (Si + Di) \le Sj$$

COMPLEXIDADES

A complexidade temporal é **O(T*N^2)** porque se percorrem as ofertas todas no primeiro ciclo e no segundo apenas são percorridas até se encontrar uma oferta não compatível.

A complexidade espacial é **Θ(T*N)** em que N é o número de ofertas.

Nota: O T corresponde ao número de casos de teste.

Francisco Mateus (53270) Vasco Martins (53299)

CONCLUSÕES

A nossa solução mostrou-se ser o mais eficiente possível em termos de espaço. No entanto, o mesmo não se pode dizer em termos de tempo. O 2º ciclo, que serve para encontrar ofertas compatíveis com a i oferta, percorre desnecessariamente as i-1 ofertas anteriores, o que eleva a complexidade para $\Theta(T*N^2)$. Este erro podia ser corrigido procurando apenas a última oferta k compatível que contribuísse para o lucro anterior. Neste caso, a complexidade seria reduzida a $\Theta(T*N)$ no melhor e expectável caso. O ciclo seria otimizado de acordo com o seguinte excerto de código Java:

```
long r = getOffer(i).price;
boolean found = false;
for(int k = Math.max(i - 1, 1); k > 0 && !found;  k--)
{
     if((compatible(k, i) && (found = maxProfit[k] > maxProfit[k-1])))
          r = getOffer(i).price + maxProfit[k];
}
maxProfit[i] = Math.max( r, maxProfit[i-1]);
```

Verificamos que esta correção permitiu reduzir o tempo de resolução do problema de 2 segundos para 300 milissegundos, num dos testes que geramos, em particular, num teste de 28 casos com uma média de 2000 ofertas cada um. Além disso, houve umas comparações desnecessárias em certos trechos do código submetido que esquecemos de retirar:

- A verificação no *if* do 2º ciclo de que `k==0` não se mostra necessária.
- No método `compatible(i, j)`, a verificação de que o `i == 0` era igualmente dispensável.
- O `Math.max(r, max)` era também desnecessário, assim como uma das suas variáveis é redundante.

CÓDIGO FONTE

Classe Main

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.SortedSet;
import java.util.TreeSet;
import Drina.*;

public class Main
{

    /**
     * List of the program arguments.
     */
    public static List<String> argsL = null;
    /**
     * Flags
     */
    public static final String
    RECURSION_FLAG = "-r",
    TEST_INPUT_FLAG = "-t",
    DURATION_FLAG = "-d",
    GENERATE_TESTFILE_FLAG = "-g";

    public static void main(String[] args) throws Exception
    {
        long startTime = System.currentTimeMillis();
        argsL = Arrays.asList(args);
        BufferedReader in = new BufferedReader(new InputStreamReader(Syst
em.in));
        if (argsL.contains(GENERATE_TESTFILE_FLAG))
        {
            generateTestFile(in);
            in.close();
            return;
        }
```

```
        if (argsL.contains(TEST_INPUT_FLAG))
        {
            in = getTestInputReader(in);
            startTime = System.currentTimeMillis();
        }
        int test_cases = Integer.parseInt(in.readLine());
        List<Long> results = new LinkedList<Long>();
        for(int t = 0; t < test_cases; t++)
        {
            int numberOffers = Integer.parseInt(in.readLine());
            SortedSet<Offer> offers = new TreeSet<Offer>(new OfferCompEnd
Time());
            for(int i = 0; i < numberOffers; i++)
            {
                String[] s = in.readLine().split(" ");
                int start = Integer.parseInt(s[0]);
                int duration = Integer.parseInt(s[1]);
                int price = Integer.parseInt(s[2]);
                offers.add(new Offer(start, duration, price));
            }
            long result;
            if (argsL.contains(RECURSION_FLAG))
                result = new Drina(offers).solveR();
            else
                result = new Drina(offers).solveDP();
            results.add(result);
        }
        in.close();
        for(long r : results)
            System.out.println(r);
        long endTime = System.currentTimeMillis();
        if(argsL.contains(DURATION_FLAG))
            System.out.println("Duration: " + (endTime - startTime) + " m
s");
    }

    private static BufferedReader getTestInputReader(BufferedReader in) t
hrows Exception
    {
        System.out.print("Insert test filename: ");
        String file = in.readLine();
        in.close();
        return new BufferedReader(new InputStreamReader(new FileInputStre
am(file)));
    }

    private static void generateTestFile(BufferedReader in) throws IOExce
ption
    {
```

```
        System.out.print("Test cases: ");
        int test_cases = Integer.parseInt(in.readLine());
        System.out.print("Max offers: ");
        int max_offers = Integer.parseInt(in.readLine());
        System.out.print("Max start time: ");
        int max_start = Integer.parseInt(in.readLine());
        System.out.print("Max duration: ");
        int max_duration = Integer.parseInt(in.readLine());
        System.out.print("Max price: ");
        int max_price = Integer.parseInt(in.readLine());

        File f = new File("test" + Double.valueOf(1000*Math.random()).int
Value() + ".txt");
        f.createNewFile();
        BufferedWriter writer = new BufferedWriter(new FileWriter(f));
        writer.write(test_cases + "\n");
        for(int i = 0; i < test_cases; i++)
        {
            int offers = Double.valueOf(Math.random()*(max_offers - 1) +
1).intValue();
            writer.write(offers + "\n");
            for(int j = 0; j  < offers; j++)
            {
                int start = Double.valueOf(Math.random()*max_start).intVa
lue();
                int duration = Double.valueOf(Math.random()*(max_duration
 - 1) + 1).intValue();
                int price = Double.valueOf(Math.random()*(max_price - 1)
+ 1).intValue();
                writer.write(String.format("%d %d %d\n", start, duration,
 price));
            }
        }
        writer.close();
        System.out.println("File generated!");
    }
}
```

Classe Drina

```
package Drina;

import java.util.ArrayList;
import java.util.List;
import java.util.SortedSet;

public class Drina
{
    public static final int MAX_TEST_CASES = 100;
    public static final int MAX_OFFERS = 10000;
    public static final int MAX_STARTING_INSTANT = 1000000000;
    public static final int MAX_DURATION = 2*10000;
    public static final int MAX_PRICE = 10000;

    /**
     * List of offers ordered by their ending time.
     */
    private final List<Offer> offers;

    /**
     * Stores the maximum profit that is obtained
     * with i offers in its respective array position.
     */
    private long[] maxProfit;

    public Drina(SortedSet<Offer> offers)
    {
        this.offers = new ArrayList<Offer>(offers);
        this.maxProfit = null;
    }


    /**
     * Initializes and builds {@link Drina#maxProfit max profit array} in
     * order to solve the problem with {@link Drina#solveDP() dynamic pro
gramming}.
     */
    public void buildArray()
    {
        maxProfit = new long[offers.size() + 1];
        for(int i = 0; i < maxProfit.length; i++)
            maxProfit[i] = 0;
    }

    /**
     * Solves the problem with the dynamic programming technique.
```

```
 * @return Result of solving the problem
 */
public long solveDP()
{
    buildArray();
    for(int i = 1; i <= offers.size(); i++)
    {
        long max = getOffer(i).price;
        for(int k = 1; compatible(k, i);  k++)
        {
            long r = 0;
            if(k == 0 || maxProfit[k] > maxProfit[k-1])
                r = getOffer(i).price + maxProfit[k];
            max = Math.max(max, r);
        }
        maxProfit[i] = Math.max( max, maxProfit[i-1]);
    }
    return maxProfit[offers.size()];
}

/**
 * Auxiliary method of {@link Drina#solveR() }
 * to solve the problem with recursion.
 */
private long solveRS(int i)
{
    if (i == 0)
        return 0;
    long max = getOffer(i).price;
    for(int k = 1; compatible(k, i); k++)
    {
        long r = 0;
        long lastProfit = solveRS(k);
        if(k == 0 || lastProfit > solveRS(k-1))
            r = getOffer(i).price + lastProfit;
        max = Math.max(max, r);
    }
    return Math.max(max, solveRS(i - 1));
}

/**
 * Solves the problem with recursion.
 * The recursion flag must be passed to the
 * program in order to trigger this method.
 */
public long solveR()
{
    return solveRS(offers.size());
}
```

```
    private Offer getOffer(int i)
    {
        return offers.get(i-1);
    }

    /**
     * Checks if 2 offers are compatible, that is,
     * if the ending time of offer i is less or
     * equal to start time of offer j.
     */
    private boolean compatible(int i, int j)
    {
        return i == 0 || (getOffer(i).endTime <= getOffer(j).startingTime
);
    }
}
```

Classe Offer

```
package Drina;

public class Offer
{
    public final int endTime;
    public final int startingTime;
    public final int price;

    public Offer(int startingTime, int duration, int price)
    {
        this.startingTime = startingTime;
        this.endTime = startingTime + duration;
        this.price = price;
    }

    public String toString()
    {
        return String.format("(%d %d %d)", startingTime, endTime, price);
    }
}
```

Classe OfferCompEndTime (Comparador)

```
package Drina;
import java.util.Comparator;
public class OfferCompEndTime implements Comparator<Offer>
{
    public OfferCompEndTime(){}

    @Override
    public int compare(Offer o1, Offer o2) {
        if(o1.endTime < o2.endTime)
            return -1;
        if(o1.endTime > o2.endTime)
            return 1;
        if(o1.endTime == o2.endTime)
        {
            if(o1.startingTime < o2.startingTime)
                return -1;
            if(o1.startingTime > o2.startingTime)
                return 1;
            if(o1.startingTime == o2.startingTime)
                return o1.price < o2.price ? 0 : 1;
        }
        return 0;
    }
}
```