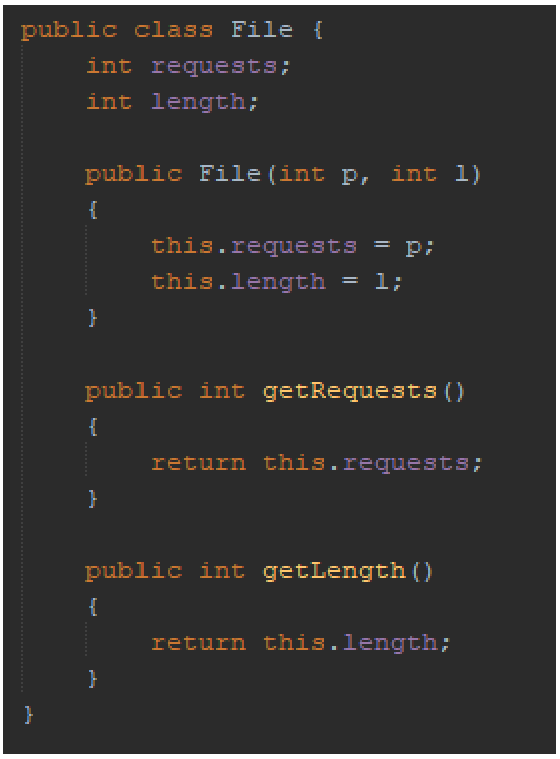
PROBLEMS: UNIT 2

VORACIOUS ALGORITHMS

JAVIER GARCÍA JIMÉNEZ AND ISABEL MARTÍNEZ GÓMEZ

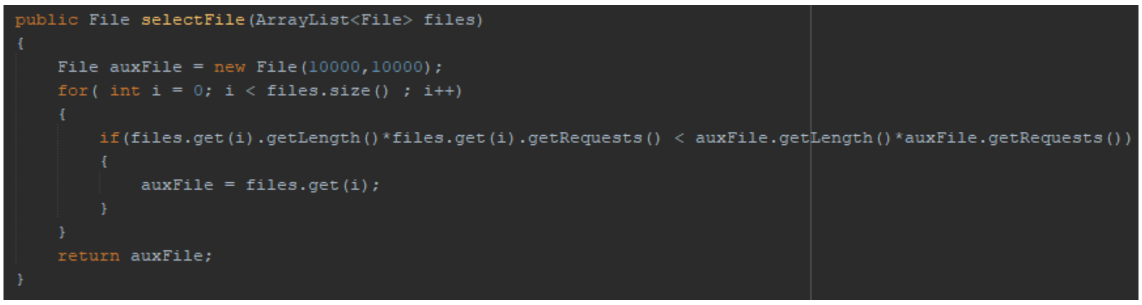
EXERCISE 2

We have created a class File that stores the number of the requests corresponding to the file and the length of the file.



```java
public class File {
    int requests;
    int length;

    public File(int p, int l)
    {
        this.requests = p;
        this.length = l;
    }

    public int getRequests()
    {
        return this.requests;
    }

    public int getLength()
    {
        return this.length;
    }
}
```

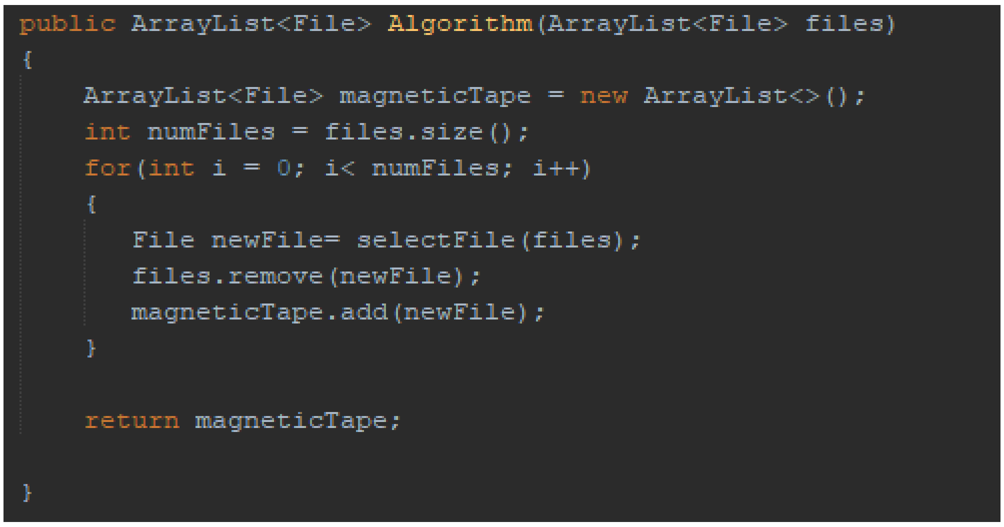
The function selectFile returns the file whose quantity is the lowest one from the set of files on the magnetic tape. This is the algorithm that selects the best candidate of voracious algorithm.



```java
public File selectFile(ArrayList<File> files)
{
    File auxFile = new File(10000,10000);
    for( int i = 0; i < files.size() ; i++)
    {
        if(files.get(i).getLength()*files.get(i).getRequests() < auxFile.getLength()*auxFile.getRequests())
        {
            auxFile = files.get(i);
        }
    }
    return auxFile;
}
```

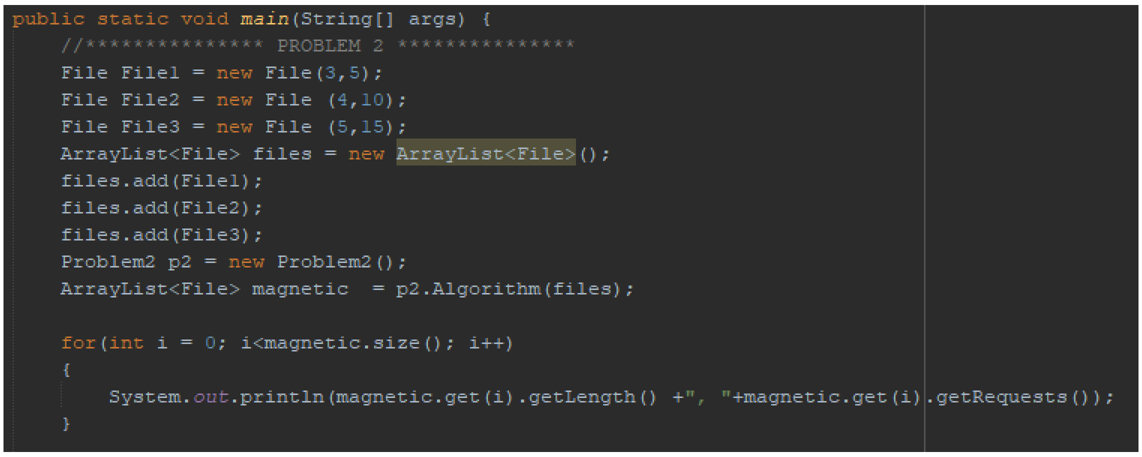
Finally, we have our voracious algorithm to solve this problem. In this function there is an entry of an array of files with size n that are going to be part of the solution.

To get the optimal solution, which is going to minimizes the quantity, in each step we try to add a file to the *ArrayList magneticTape* using the previous function explained. The algorithm ends when we have all the files in the array so that the optimal solution is reached.



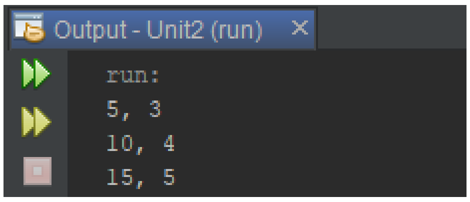
```java
public ArrayList<File> Algorithm(ArrayList<File> files)
{
    ArrayList<File> magneticTape = new ArrayList<>();
    int numFiles = files.size();
    for(int i = 0; i< numFiles; i++)
    {
        File newFile= selectFile(files);
        files.remove(newFile);
        magneticTape.add(newFile);
    }

    return magneticTape;

}
```

In the main class we have created some Files with its requests and length.



```java
public static void main(String[] args) {
    //*************** PROBLEM 2 ***************
    File File1 = new File(3,5);
    File File2 = new File (4,10);
    File File3 = new File (5,15);
    ArrayList<File> files = new ArrayList<File>();
    files.add(File1);
    files.add(File2);
    files.add(File3);
    Problem2 p2 = new Problem2();
    ArrayList<File> magnetic  = p2.Algorithm(files);

    for(int i = 0; i<magnetic.size(); i++)
    {
        System.out.println(magnetic.get(i).getLength() +", "+magnetic.get(i).getRequests());
    }
```

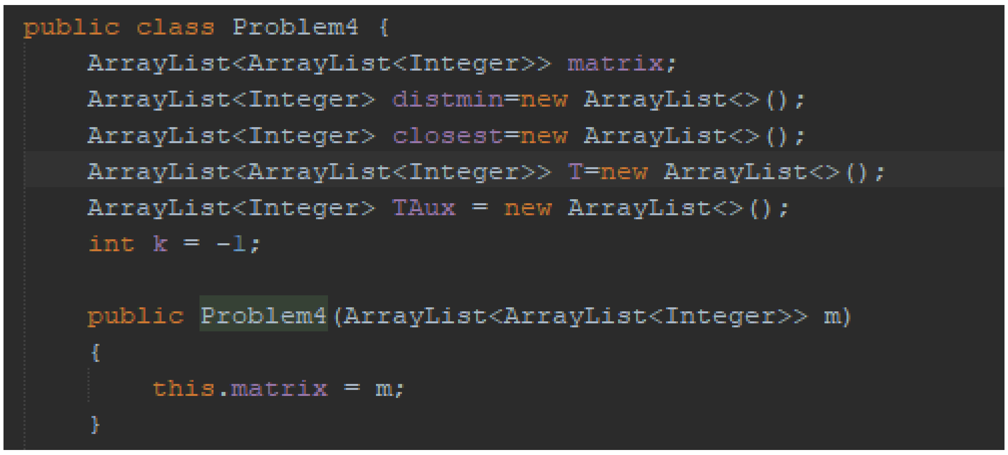
The result of this is the following:



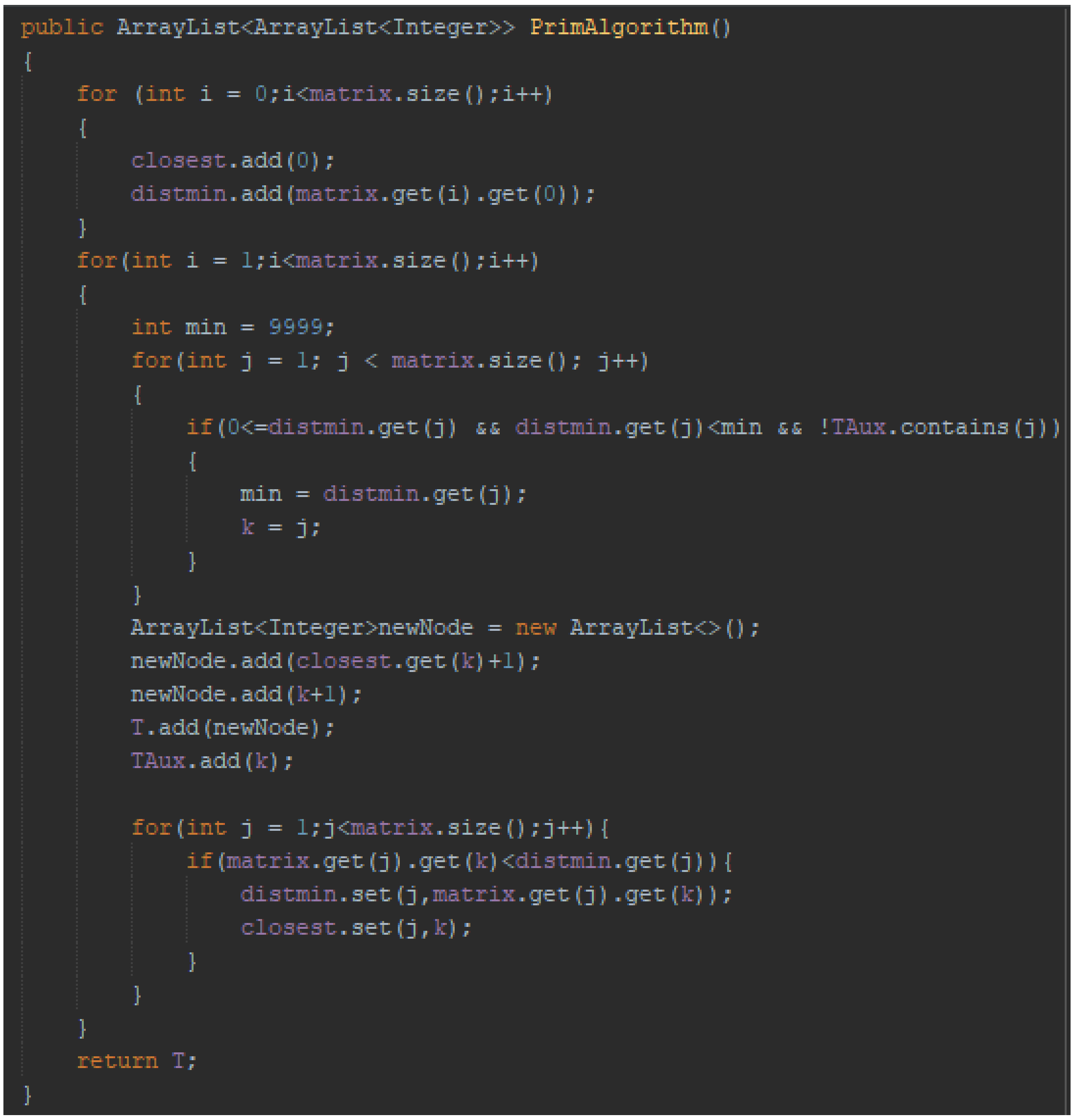
```
run:
5, 3
10, 4
15, 5
```

As we can see this solution is the optimal one because it minimizes the average loading time.

EXERCISE 4

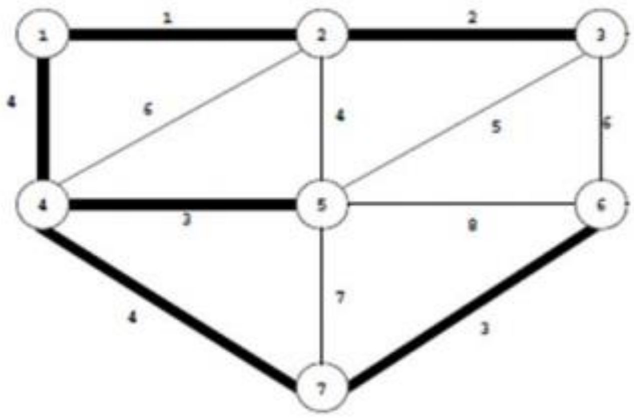


```java
public class Problem4 {
    ArrayList<ArrayList<Integer>> matrix;
    ArrayList<Integer> distmin=new ArrayList<>();
    ArrayList<Integer> closest=new ArrayList<>();
    ArrayList<ArrayList<Integer>> T=new ArrayList<>();
    ArrayList<Integer> TAux = new ArrayList<>();
    int k = -1;

    public Problem4(ArrayList<ArrayList<Integer>> m)
    {
        this.matrix = m;
    }
```

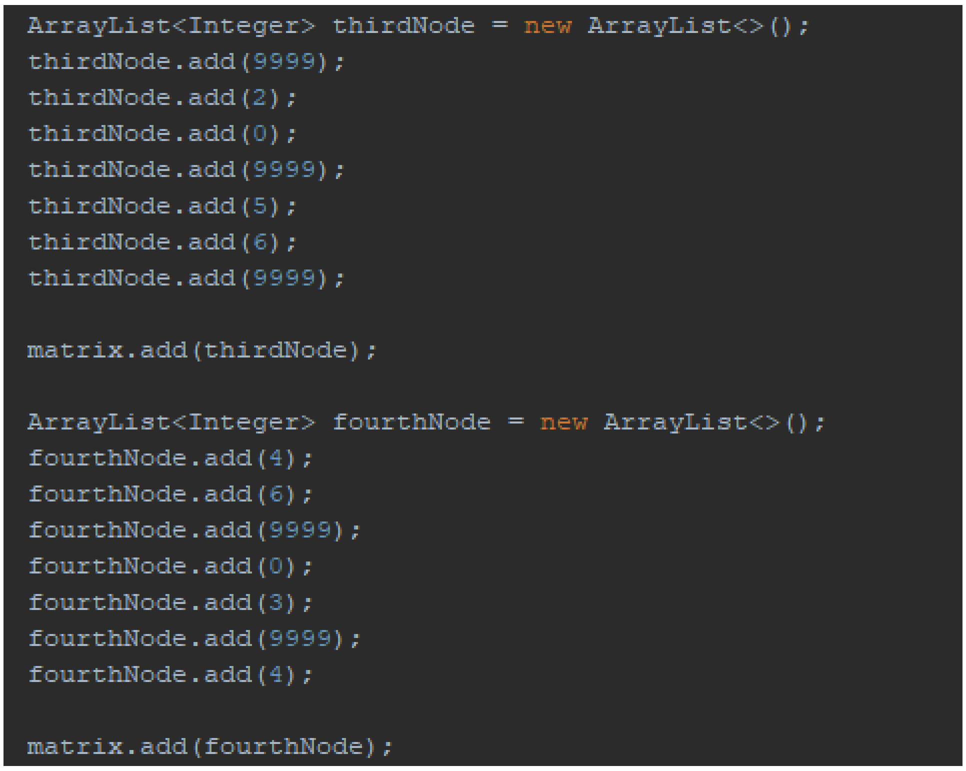


```java
public ArrayList<ArrayList<Integer>> PrimAlgorithm()
{
    for (int i = 0;i<matrix.size();i++)
    {
        closest.add(0);
        distmin.add(matrix.get(i).get(0));
    }
    for(int i = 1;i<matrix.size();i++)
    {
        int min = 9999;
        for(int j = 1; j < matrix.size(); j++)
        {
            if(0<=distmin.get(j) && distmin.get(j)<min && !TAux.contains(j))
            {
                min = distmin.get(j);
                k = j;
            }
        }
        ArrayList<Integer>newNode = new ArrayList<>();
        newNode.add(closest.get(k)+1);
        newNode.add(k+1);
        T.add(newNode);
        TAux.add(k);

        for(int j = 1;j<matrix.size();j++){
            if(matrix.get(j).get(k)<distmin.get(j)){
                distmin.set(j,matrix.get(j).get(k));
                closest.set(j,k);
            }
        }
    }
    return T;
}
```

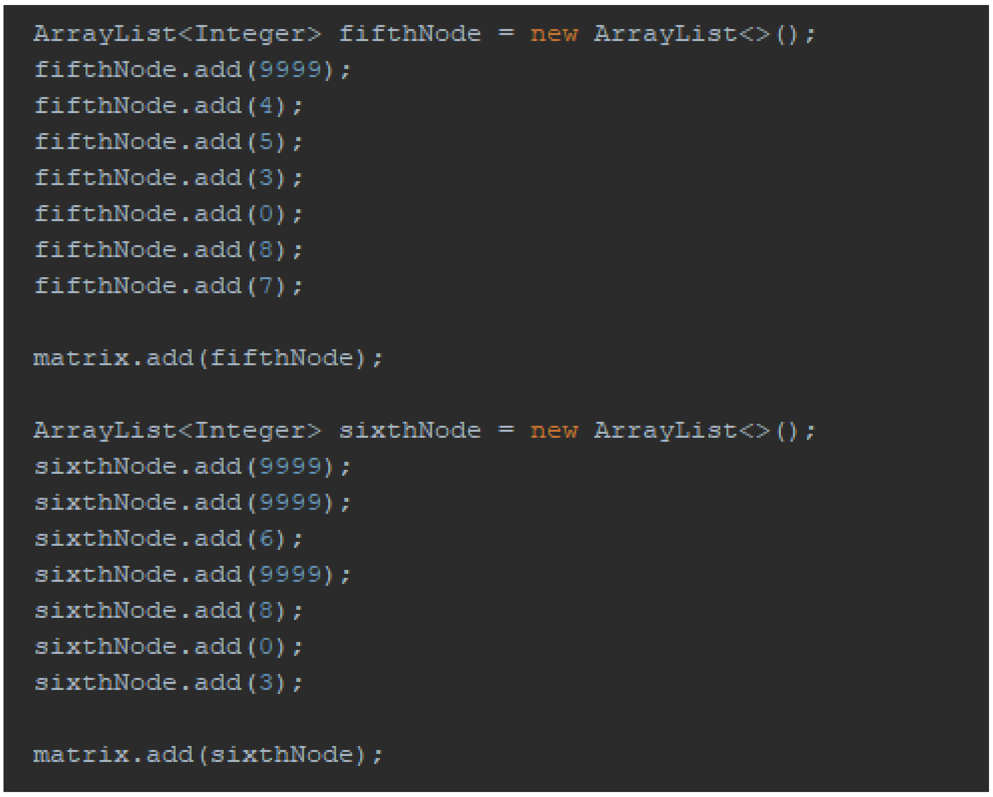
This is the main function where we are creating the following graph.



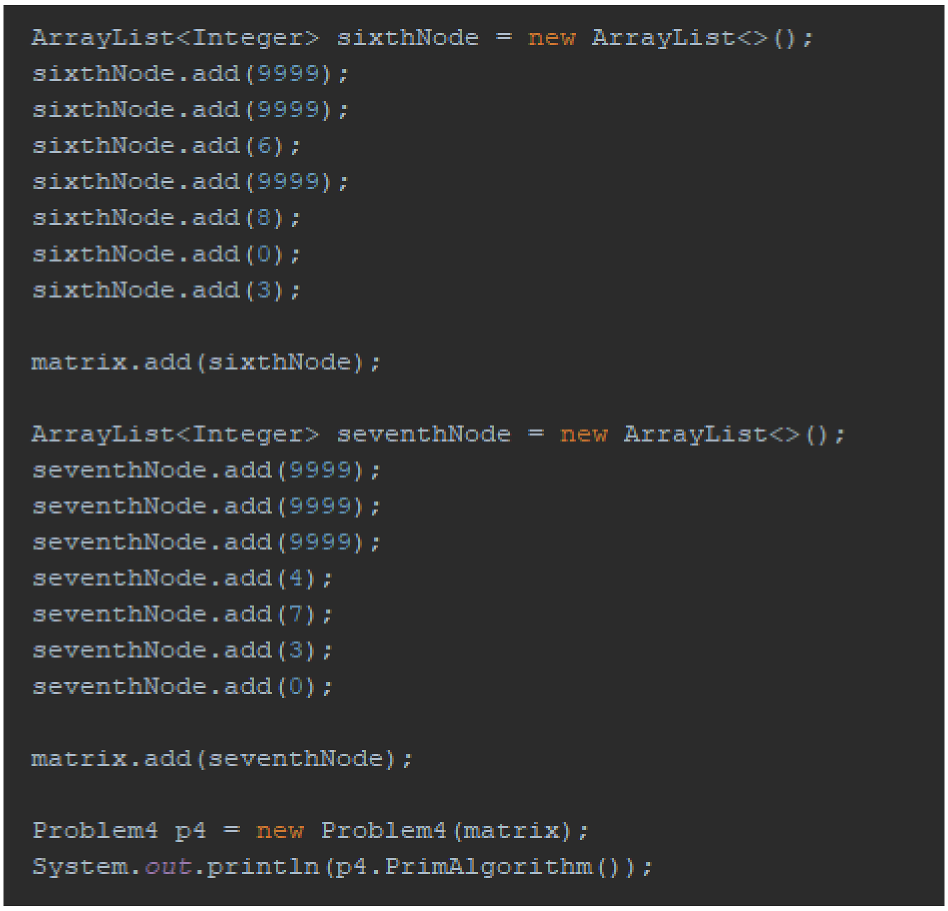
```
ArrayList<ArrayList<Integer>> matrix = new ArrayList<>();

ArrayList<Integer> firstNode = new ArrayList<>();
firstNode.add(0);
firstNode.add(1);
firstNode.add(9999);
firstNode.add(4);
firstNode.add(9999);
firstNode.add(9999);
firstNode.add(9999);

matrix.add(firstNode);

ArrayList<Integer> secondNode = new ArrayList<>();
secondNode.add(1);
secondNode.add(0);
secondNode.add(2);
secondNode.add(4);
secondNode.add(9999);
secondNode.add(9999);
secondNode.add(9999);

matrix.add(secondNode);
```



```
ArrayList<Integer> thirdNode = new ArrayList<>();
thirdNode.add(9999);
thirdNode.add(2);
thirdNode.add(0);
thirdNode.add(9999);
thirdNode.add(5);
thirdNode.add(6);
thirdNode.add(9999);

matrix.add(thirdNode);

ArrayList<Integer> fourthNode = new ArrayList<>();
fourthNode.add(4);
fourthNode.add(6);
fourthNode.add(9999);
fourthNode.add(0);
fourthNode.add(3);
fourthNode.add(9999);
fourthNode.add(4);

matrix.add(fourthNode);
```

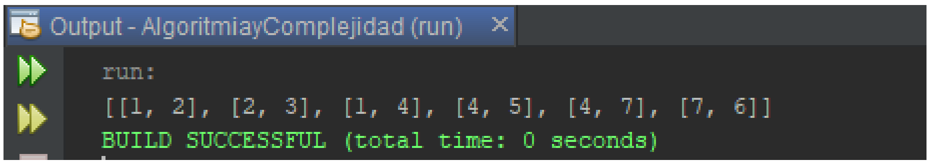


```
ArrayList<Integer> fifthNode = new ArrayList<>();
fifthNode.add(9999);
fifthNode.add(4);
fifthNode.add(5);
fifthNode.add(3);
fifthNode.add(0);
fifthNode.add(8);
fifthNode.add(7);

matrix.add(fifthNode);

ArrayList<Integer> sixthNode = new ArrayList<>();
sixthNode.add(9999);
sixthNode.add(9999);
sixthNode.add(6);
sixthNode.add(9999);
sixthNode.add(8);
sixthNode.add(0);
sixthNode.add(3);

matrix.add(sixthNode);
```



```java
ArrayList<Integer> sixthNode = new ArrayList<>();
sixthNode.add(9999);
sixthNode.add(9999);
sixthNode.add(6);
sixthNode.add(9999);
sixthNode.add(8);
sixthNode.add(0);
sixthNode.add(3);

matrix.add(sixthNode);

ArrayList<Integer> seventhNode = new ArrayList<>();
seventhNode.add(9999);
seventhNode.add(9999);
seventhNode.add(9999);
seventhNode.add(4);
seventhNode.add(7);
seventhNode.add(3);
seventhNode.add(0);

matrix.add(seventhNode);

Problem4 p4 = new Problem4(matrix);
System.out.println(p4.PrimAlgorithm());
```

Finally, this is the result of the example executed in the main function.



```
run:
[[1, 2], [2, 3], [1, 4], [4, 5], [4, 7], [7, 6]]
BUILD SUCCESSFUL (total time: 0 seconds)
```

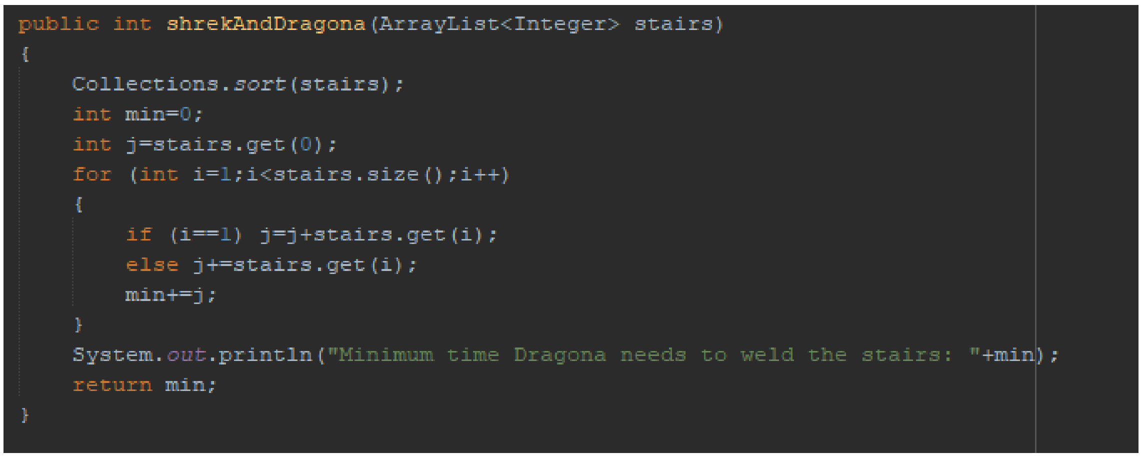
EXERCISE 6

In the attached image we have the voracious algorithm implemented where we calculate the minimum cost to weld all the stairs in the vector.

To do this algorithm, the first thing we will have to do is order the stairs arraylist so that we can obtain the minimum time possible.

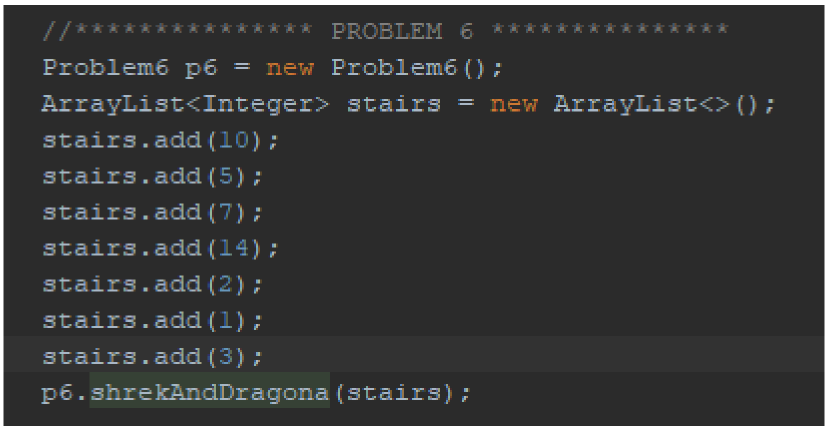
Then, we go through the arraylist stairs adding each first, the first stair with the second, then it is added the result of the first and the second with the third stair and so on until all the stairs are added.

Finally, we return min that is the minimum cost to weld all the stairs in the arraylist.



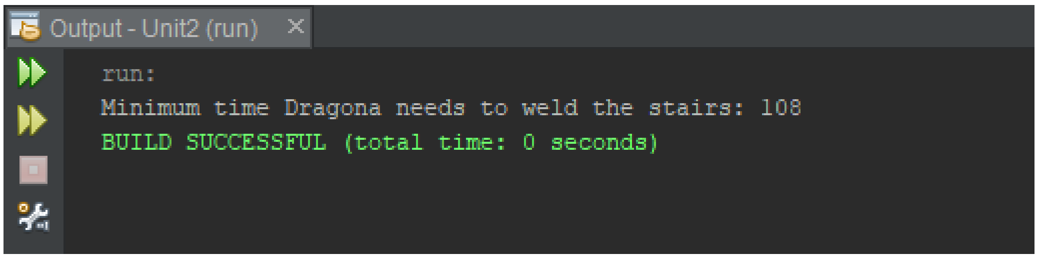
```
public int shrekAndDragona(ArrayList<Integer> stairs)
{
    Collections.sort(stairs);
    int min=0;
    int j=stairs.get(0);
    for (int i=1;i<stairs.size();i++)
    {
        if (i==1) j=j+stairs.get(i);
        else j+=stairs.get(i);
        min+=j;
    }
    System.out.println("Minimum time Dragona needs to weld the stairs: "+min);
    return min;
}
```

To prove it, we have considered that the arraylist **stairs** has the following length of stairs:



```
//*************** PROBLEM 6 ***************
Problem6 p6 = new Problem6();
ArrayList<Integer> stairs = new ArrayList<>();
stairs.add(10);
stairs.add(5);
stairs.add(7);
stairs.add(14);
stairs.add(2);
stairs.add(1);
stairs.add(3);
p6.shrekAndDragona(stairs);
```

We get the result:



```
run:
Minimum time Dragona needs to weld the stairs: 108
BUILD SUCCESSFUL (total time: 0 seconds)
```