# PROBLEMS: UNIT 3

DIVIDE AND CONQUER ALGORITHMS

JAVIER GARCÍA JIMÉNEZ AND ISABEL MARTÍNEZ GÓMEZ

## EXERCISE 4

To solve this problem, we have used a divide and conquer algorithm to close the bottles with their respective corks. To do this and to make it as efficient as possible we have used a pivot so the robot can place with his mechanical arms the corks in different positions of a table.

We do this because the robot cannot order the corks and bottles and compare them. But we consider in the program using the pivot if a cork is too small, large or equal for a bottle.

Next, we will explain this more carefully.

### DIVIDE AND CONQUER ALGORITHM

To perform the algorithm, we have created the function closeBottles.

**CLOSEBOTTLES**

We will first discuss the data structures we have used to solve the problem:

- ArrayList table → is an auxiliary arraylist that we will use to place the corks once compared with the bottle pivot.
- ArrayList bottles → is an arraylist which contains the size of all the bottles.
- ArrayList corks → is an arraylist which contains the size of all the corks.
- Int bottlePivot → it will be the pivot chosen in the first iteration of the algorithm in which we choose the first bottle by default as a pivot to put the corks in an "ordered" way on the table.
- Boolean first → it is true in the fist iteration and false in the rest. It helps us to know when the first iteration of the algorithm is, in which the robot will put the corks on the table using the chosen pivot.
- Int cont → It is used to divide the arraylist in two parts and recursively call the function or from position 0 to cont if the cork we are going to look for is smaller than the pivot or from 0 to table.size () if the cork we are going to look for is greater than the pivot.

Now we will explain the closeBottle function step by step.

First, we enter to the function with the arraylist bottles with all the sizes of the bottles, the arraylist corks with all the sizes of the corks, bottlePivot is a -1 and cont is a 0.

**FIRST CONDITION → IF (CONT == 0)**

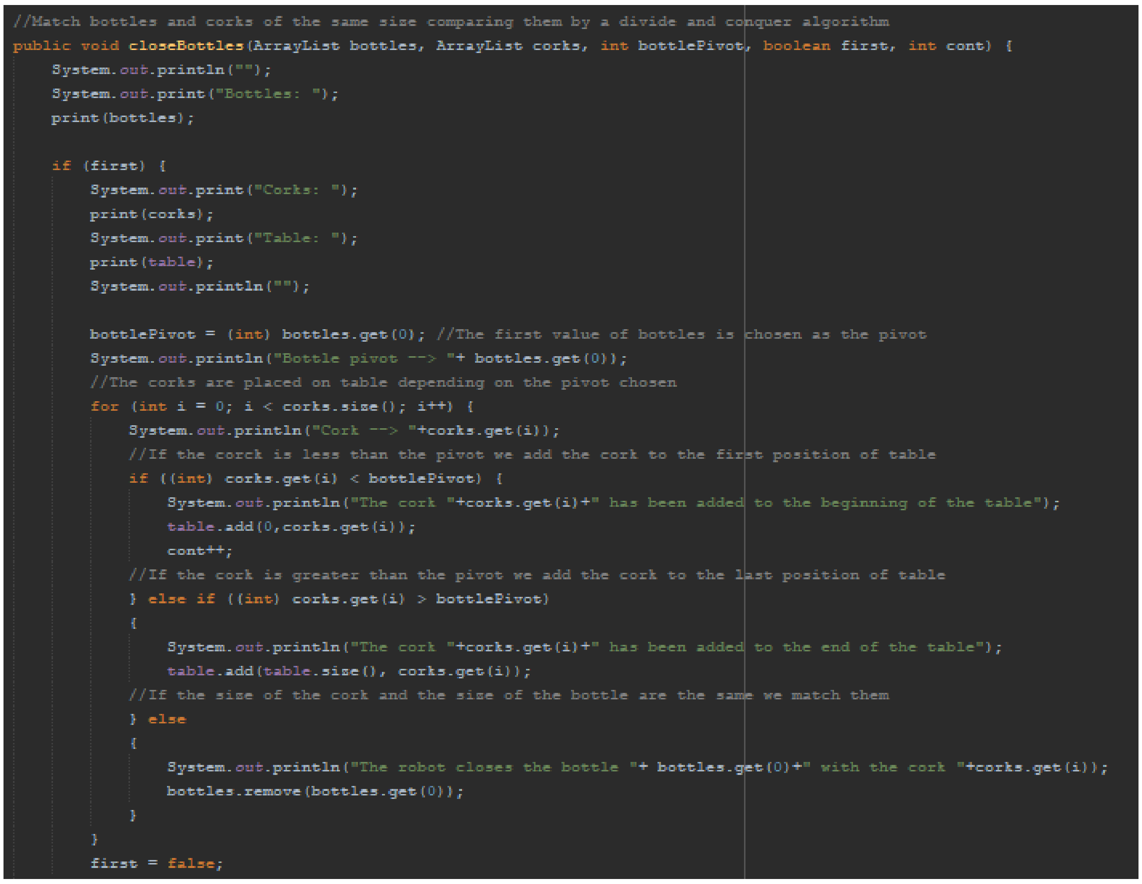
In the first iteration we will enter the first condition (cont=0) and inside the condition we will change the value of bottlePivot, to the value of the first bottle.

With the for loop we go through the corks arraylist and we compare if the obtained cork is less, greater than or equal to the pivot:

- If is less than the pivot we add it at the beginning of the auxiliar arraylist table and we increment the cont variable in 1.
- If is greater than the pivot we add it at the end of the auxiliar arraylist table.
- If is equal to the pivot we remove the bottle. This is done so that when the bottles arraylist is empty, the algorithm ends and then all the bottles will have been closed.

Finally, we put the variable first to false indicating that the first iteration is over.

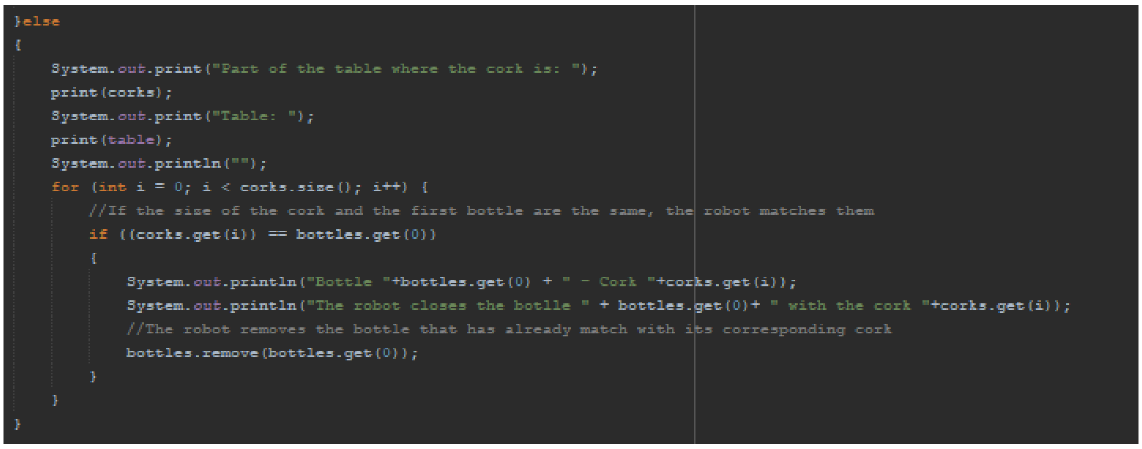
Once the first iteration ends, we have chosen a bottle pivot and also we have placed in the table the corks smaller than the pivot bottle on the left of the pivot and the corks larger than the pivot bottle on the right of the pivot.



```java
//Match bottles and corks of the same size comparing them by a divide and conquer algorithm
public void closeBottles(ArrayList bottles, ArrayList corks, int bottlePivot, boolean first, int cont) {
    System.out.println("");
    System.out.print("Bottles: ");
    print(bottles);

    if (first) {
        System.out.print("Corks: ");
        print(corks);
        System.out.print("Table: ");
        print(table);
        System.out.println("");

        bottlePivot = (int) bottles.get(0); //The first value of bottles is chosen as the pivot
        System.out.println("Bottle pivot --> "+ bottles.get(0));
        //The corks are placed on table depending on the pivot chosen
        for (int i = 0; i < corks.size(); i++) {
            System.out.println("Cork --> "+corks.get(i));
            //If the corck is less than the pivot we add the cork to the first position of table
            if ((int) corks.get(i) < bottlePivot) {
                System.out.println("The cork "+corks.get(i)+" has been added to the beginning of the table");
                table.add(0,corks.get(i));
                cont++;
            //If the cork is greater than the pivot we add the cork to the last position of table
            } else if ((int) corks.get(i) > bottlePivot)
            {
                System.out.println("The cork "+corks.get(i)+" has been added to the end of the table");
                table.add(table.size(), corks.get(i));
            //If the size of the cork and the size of the bottle are the same we match them
            } else
            {
                System.out.println("The robot closes the bottle "+ bottles.get(0)+" with the cork "+corks.get(i));
                bottles.remove(bottles.get(0));
            }
        }
        first = false;
```

### ELSE (if we are not in the first iteration)

If we are not in the first iteration, it means that we already have chosen a bottle pivot and that we already have the corks placed on the table based on a pivot and now we must iterate over the corks arraylist to find the cork that closes the first bottle. Once it finds the cork, it removes the bottle.



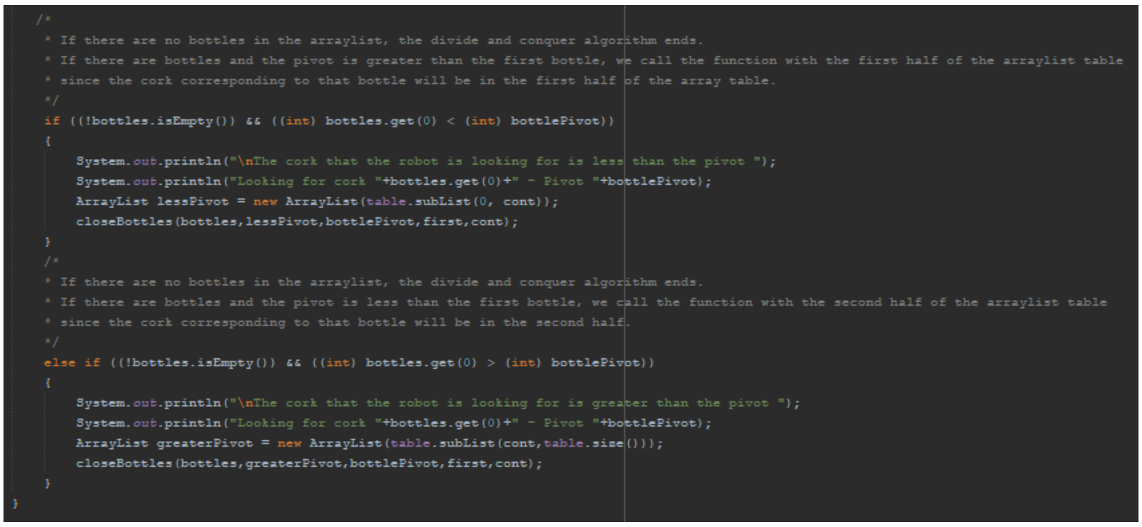
```java
}else
{
    System.out.print("Part of the table where the cork is: ");
    print(corks);
    System.out.print("Table: ");
    print(table);
    System.out.println("");
    for (int i = 0; i < corks.size(); i++) {
        //If the size of the cork and the first bottle are the same, the robot matches them
        if ((corks.get(i)) == bottles.get(0))
        {
            System.out.println("Bottle "+bottles.get(0) + " - Cork "+corks.get(i));
            System.out.println("The robot closes the bottle " + bottles.get(0)+ " with the cork "+corks.get(i));
            //The robot removes the bottle that has already match with its corresponding cork
            bottles.remove(bottles.get(0));
        }
    }
}
```

After the conditionals above, we will always do the following:

Here we check if the bottles arraylist is empty. If is empty, the function will end because all the bottles have been closed with their respective corks and there is no more bottle to close.

If on the contrary, the bottles arraylist is not empty, we have to keep closing bottles, so we check if the pivot is less than the bottle that we want to close. If is less than the pivot we have to call the function recursively with the corks that are on the left side of the pivot in the table.

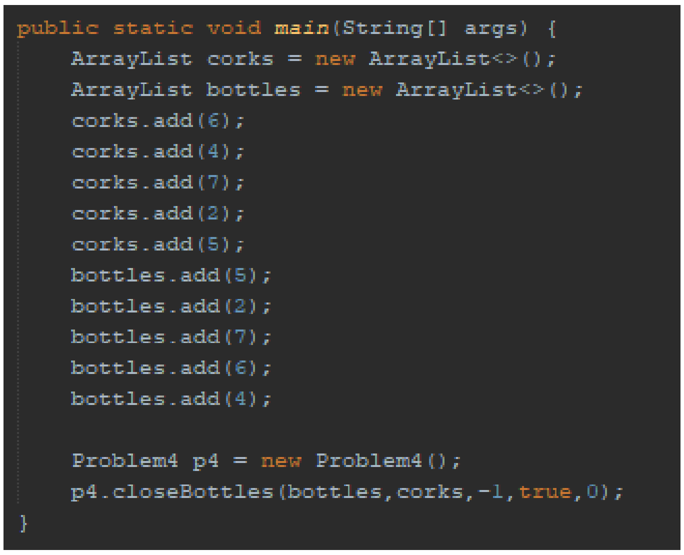
The other case would be that the cork of the bottle that we want to close is greater than the pivot, so this time we will have to recursively call the function with the right part of the table, which is where the cork we are looking for will be.



```java
    /*
     * If there are no bottles in the arraylist, the divide and conquer algorithm ends.
     * If there are bottles and the pivot is greater than the first bottle, we call the function with the first half of the arraylist table
     * since the cork corresponding to that bottle will be in the first half of the array table.
     */
    if ((!bottles.isEmpty()) && ((int) bottles.get(0) < (int) bottlePivot))
    {
        System.out.println("\nThe cork that the robot is looking for is less than the pivot ");
        System.out.println("Looking for cork "+bottles.get(0)+" - Pivot "+bottlePivot);
        ArrayList lessPivot = new ArrayList(table.subList(0, cont));
        closeBottles(bottles,lessPivot,bottlePivot,first,cont);
    }
    /*
     * If there are no bottles in the arraylist, the divide and conquer algorithm ends.
     * If there are bottles and the pivot is less than the first bottle, we call the function with the second half of the arraylist table
     * since the cork corresponding to that bottle will be in the second half.
     */
    else if ((!bottles.isEmpty()) && ((int) bottles.get(0) > (int) bottlePivot))
    {
        System.out.println("\nThe cork that the robot is looking for is greater than the pivot ");
        System.out.println("Looking for cork "+bottles.get(0)+" - Pivot "+bottlePivot);
        ArrayList greaterPivot = new ArrayList(table.subList(cont,table.size()));
        closeBottles(bottles,greaterPivot,bottlePivot,first,cont);
    }
}
```

**MAIN FUNCTION**

Here we have the main function where we have created an arraylist corks and other bottles with different sizes.

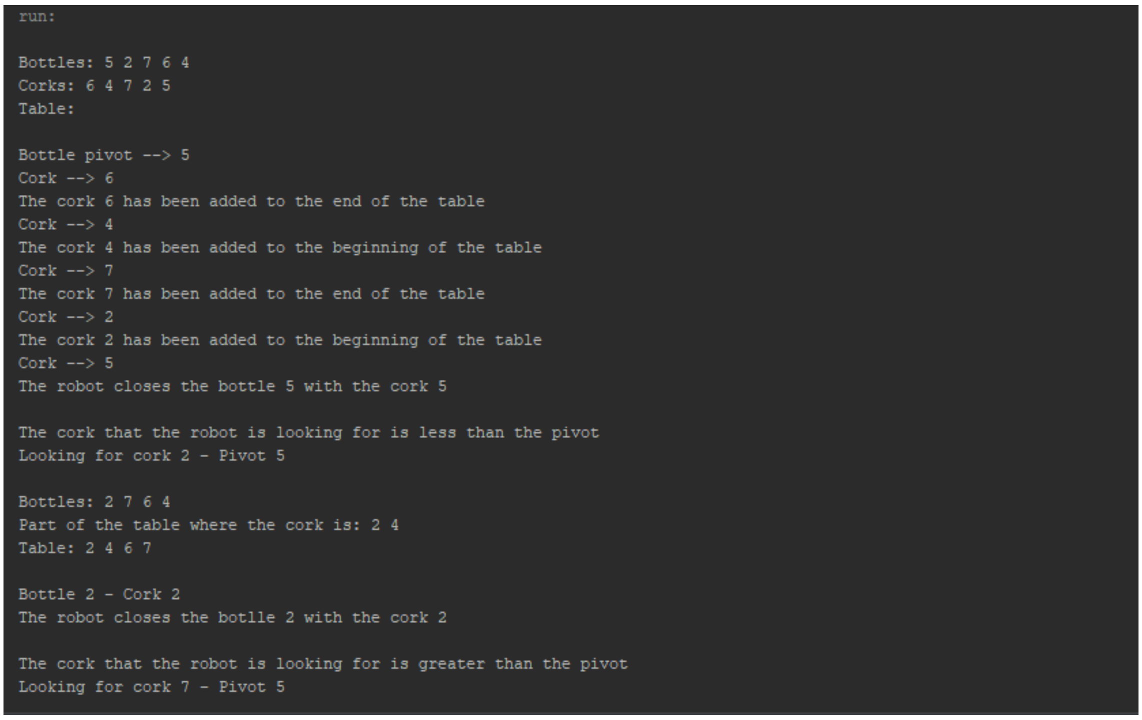


```java
public static void main(String[] args) {
    ArrayList corks = new ArrayList<>();
    ArrayList bottles = new ArrayList<>();
    corks.add(6);
    corks.add(4);
    corks.add(7);
    corks.add(2);
    corks.add(5);
    bottles.add(5);
    bottles.add(2);
    bottles.add(7);
    bottles.add(6);
    bottles.add(4);

    Problem4 p4 = new Problem4();
    p4.closeBottles(bottles,corks,-1,true,0);
}
```

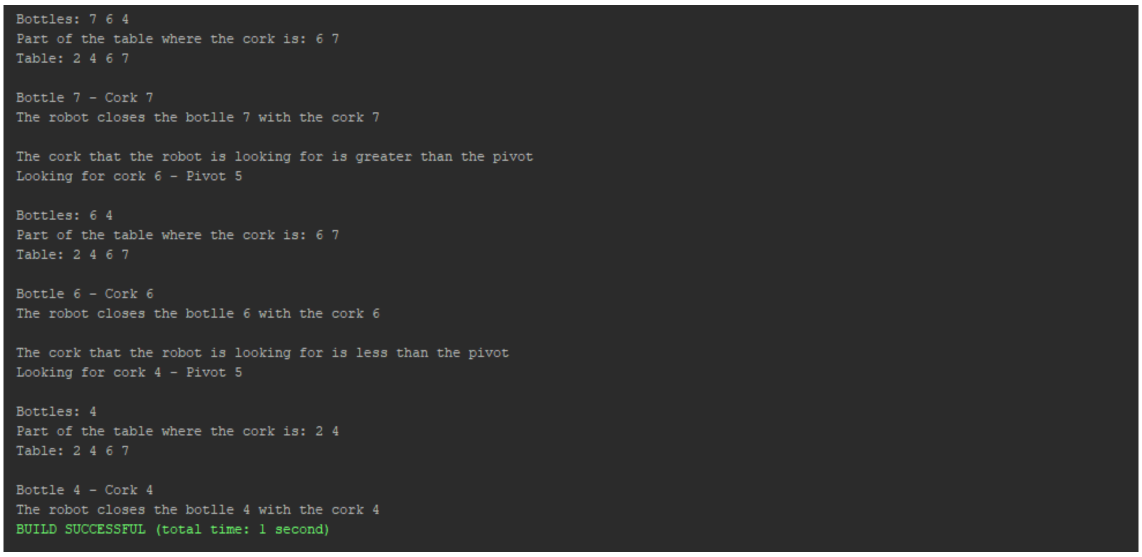
**RESULT OF THE PROGRAM**

As you can see the algorithm ends when there are no more bottles in the arraylist and all the bottles have already been closed with their corks.

We have printed all the steps so that the implementation can be understood more easily.



```
run:

Bottles: 5 2 7 6 4
Corks: 6 4 7 2 5
Table:

Bottle pivot --> 5
Cork --> 6
The cork 6 has been added to the end of the table
Cork --> 4
The cork 4 has been added to the beginning of the table
Cork --> 7
The cork 7 has been added to the end of the table
Cork --> 2
The cork 2 has been added to the beginning of the table
Cork --> 5
The robot closes the bottle 5 with the cork 5

The cork that the robot is looking for is less than the pivot
Looking for cork 2 - Pivot 5

Bottles: 2 7 6 4
Part of the table where the cork is: 2 4
Table: 2 4 6 7

Bottle 2 - Cork 2
The robot closes the botlle 2 with the cork 2

The cork that the robot is looking for is greater than the pivot
Looking for cork 7 - Pivot 5
```



```
Bottles: 7 6 4
Part of the table where the cork is: 6 7
Table: 2 4 6 7

Bottle 7 - Cork 7
The robot closes the botlle 7 with the cork 7

The cork that the robot is looking for is greater than the pivot
Looking for cork 6 - Pivot 5

Bottles: 6 4
Part of the table where the cork is: 6 7
Table: 2 4 6 7

Bottle 6 - Cork 6
The robot closes the botlle 6 with the cork 6

The cork that the robot is looking for is less than the pivot
Looking for cork 4 - Pivot 5

Bottles: 4
Part of the table where the cork is: 2 4
Table: 2 4 6 7

Bottle 4 - Cork 4
The robot closes the botlle 4 with the cork 4
BUILD SUCCESSFUL (total time: 1 second)
```

## EXERCISE 7: Abeceland

To solve the problem, we can transpose the given matrix. What the transpose does is to exchange position i for position j, therefore if M[i][j] = 0 in the given matrix, now it would be that M[j][i] = 1. Once the transpose of the given matrix is done, we could go from j to i, but not from i to j.

Therefore, the explanation of this exercise will be how we have implemented an algorithm that transposes a matrix using divide and conquer.

### DIVIDE AND CONQUER ALGORITHM

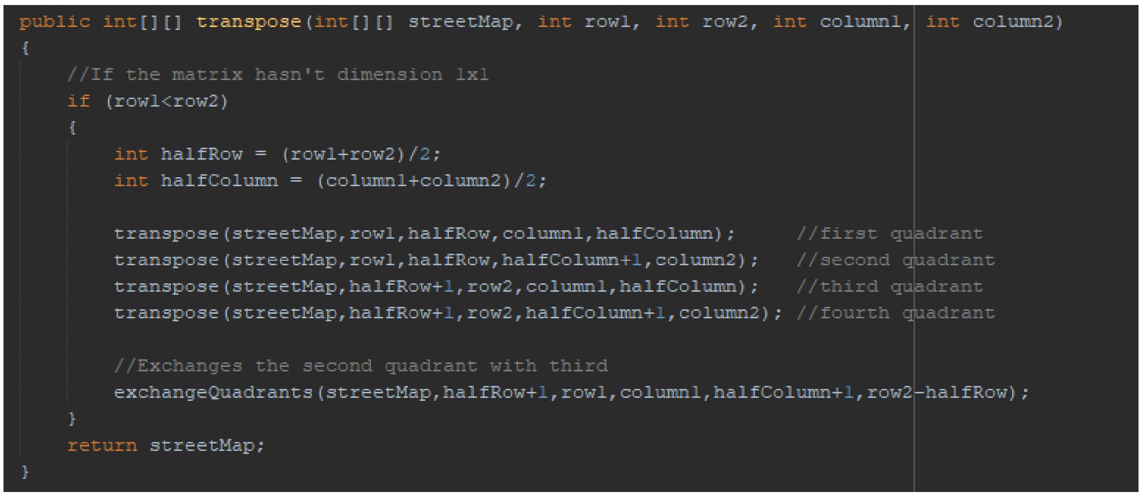
To perform the algorithm, we have created the function transpose and another exchangeQuadrants.

#### TRANSPOSE

This function recursively calls the matrix with the first quadrant, the second quadrant, the third quadrant, and the fourth quadrant until reaching the base case, which is when the matrix has dimension 1x1 and we can't continue dividing the quadrants by two.

The parameters of this function are the matrix which contains the street map, the position of the first row and the last and the position of the first column and the last of the matrix. So, if the first row is the same than the second row is because the matrix has dimension 1x1 and we return the street map.

Inside the function we call to the function exchangeQuadrants, this function will exchange the second quadrant with the third.

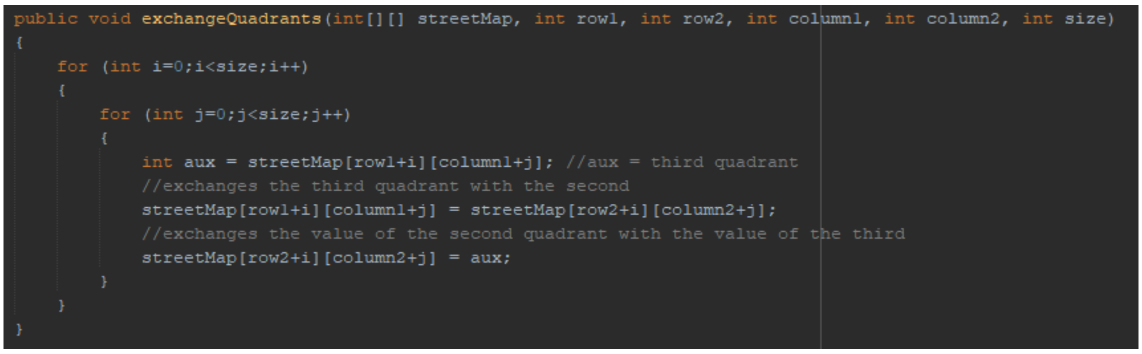


```
public int[][] transpose(int[][] streetMap, int row1, int row2, int column1, int column2)
{
    //If the matrix hasn't dimension 1x1
    if (row1<row2)
    {
        int halfRow = (row1+row2)/2;
        int halfColumn = (column1+column2)/2;

        transpose(streetMap,row1,halfRow,column1,halfColumn);      //first quadrant
        transpose(streetMap,row1,halfRow,halfColumn+1,column2);    //second quadrant
        transpose(streetMap,halfRow+1,row2,column1,halfColumn);    //third quadrant
        transpose(streetMap,halfRow+1,row2,halfColumn+1,column2); //fourth quadrant

        //Exchanges the second quadrant with third
        exchangeQuadrants(streetMap,halfRow+1,row1,column1,halfColumn+1,row2-halfRow);
    }
    return streetMap;
}
```

## EXCHANGE QUADRANTS

This function exchanges the second quadrant with the third. The function is entered with the parameters:

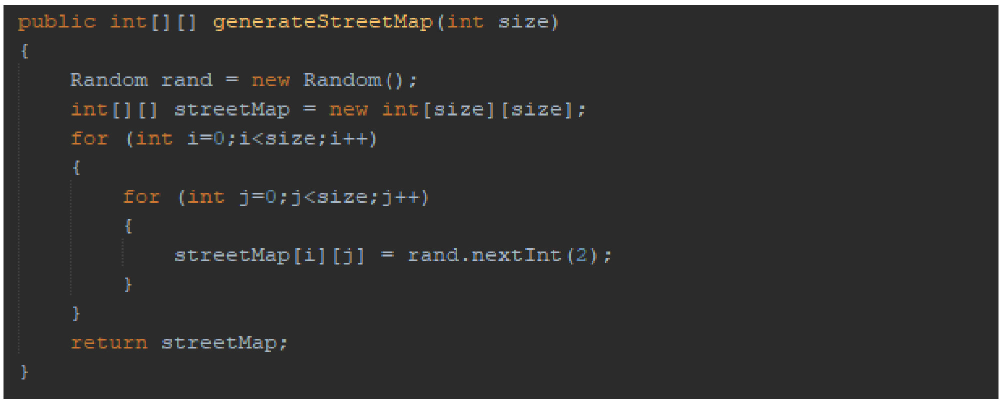
- matrix
- row1: from the third quadrant (halfRow+1)
- row2: from the second quadrant (row1)
- column1: from the third quadrant (column1)
- column2: from the second quadrant (halfColumn+1)
- size: we can calculate the dimension by substracting row2-halfRow or also we coud do column2-halfColumn.



```java
public void exchangeQuadrants(int[][] streetMap, int row1, int row2, int column1, int column2, int size)
{
    for (int i=0;i<size;i++)
    {
        for (int j=0;j<size;j++)
        {
            int aux = streetMap[row1+i][column1+j]; //aux = third quadrant
            //exchanges the third quadrant with the second
            streetMap[row1+i][column1+j] = streetMap[row2+i][column2+j];
            //exchanges the value of the second quadrant with the value of the third
            streetMap[row2+i][column2+j] = aux;
        }
    }
}
```

## GENERATE STREET MAP

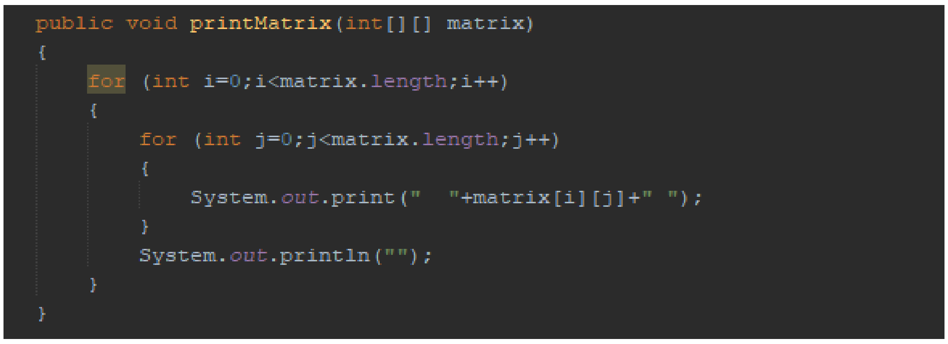
For generating the street map, we have used random numbers between 0 and 1 so that a 0 indicates that from this square you cannot go to another connected and a 1 indicates that from the square you can go to another connected.



```java
public int[][] generateStreetMap(int size)
{
    Random rand = new Random();
    int[][] streetMap = new int[size][size];
    for (int i=0;i<size;i++)
    {
        for (int j=0;j<size;j++)
        {
            streetMap[i][j] = rand.nextInt(2);
        }
    }
    return streetMap;
}
```

## PRINT MATRIX

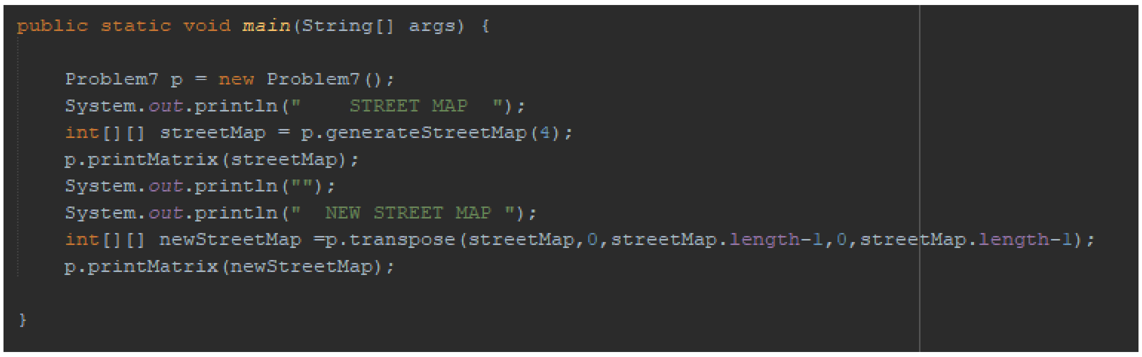
This function is used to print the street map.



```java
public void printMatrix(int[][] matrix)
{
    for (int i=0;i<matrix.length;i++)
    {
        for (int j=0;j<matrix.length;j++)
        {
            System.out.print("   "+matrix[i][j]+" ");
        }
        System.out.println("");
    }
}
```

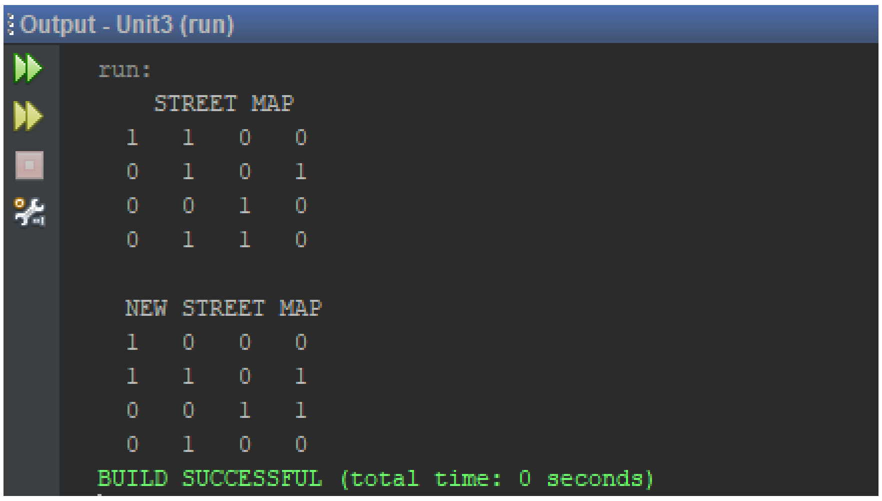
## MAIN FUNCTION

This is the main function where we generate the street map of dimension 4 and we print it. Then we call the divide and conquer algorithm, transpose, to obtain the new street map valid for the day of San Isidoro of Seville. Finally, we print the final matrix with the new street map.



```java
public static void main(String[] args) {

    Problem7 p = new Problem7();
    System.out.println("     STREET MAP   ");
    int[][] streetMap = p.generateStreetMap(4);
    p.printMatrix(streetMap);
    System.out.println("");
    System.out.println("  NEW STREET MAP ");
    int[][] newStreetMap =p.transpose(streetMap,0,streetMap.length-1,0,streetMap.length-1);
    p.printMatrix(newStreetMap);

}
```

This is the result of the program where we can see the original street map and the new street map:



Output - Unit3 (run)

```
run:
     STREET MAP
   1    1    0    0
   0    1    0    1
   0    0    1    0
   0    1    1    0

   NEW STREET MAP
   1    0    0    0
   1    1    0    1
   0    0    1    1
   0    1    0    0
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Why do we exchange the third and second quadrant?**

To make a transpose the second and third quadrants are exchanged. Let's see an example where we have a matrix A:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Then, we divide the matrix of dimension 4x4 in four quadrants:

$$A = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

After that we transpose the matrix, but we need to exchange the second quadrant with the third:

$$\begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ & \swarrow\!\!\!\!\nearrow & \\ \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} & & \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \end{bmatrix}$$

Finally, we exchange both quadrants and we get the transpose of the matrix A:

$$A^{\dagger} = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \end{bmatrix}$$