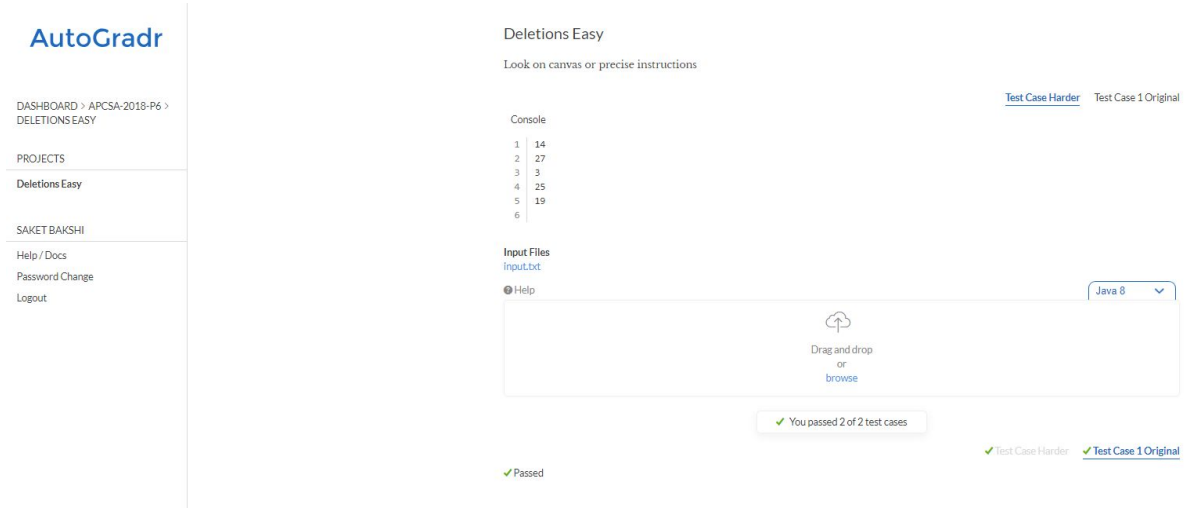
Deletions Easy

```
/* Saket Bakshi. AP Computer Science A. Deletions Easy. Due March 10, 2019.
This class solves the Deletions Easy problem.
*/
public class DeletionsEasyV2
{

	private boolean isCleared; //variable to judge if problem is solved
	private int[] input; //holds the problem sequence
	private int inputLength; //length of sequence
	private int steps; //steps to solve sequence

	/**
		This takes an integer array and solves it according to the rules of the deletions easy directions.
		@param initialSequence the initial array
	*/
	public DeletionsEasyV2(int[] initialSequence)
	{
		isCleared = false;
		input = initialSequence;
		inputLength = initialSequence.length;
		steps = 0;
	}

	/**
		This takes an integer array and solves it according to the rules of the deletions easy directions.
	*/
	public DeletionsEasyV2()
	{
		isCleared = false;
		input = new int[100];
		inputLength = 0;
		steps = 0;
	}

	/**
		This solves the sequence
	*/
	public void doTurn()
	{
		int zeroPlaceholder = 0; //sees if there are zeros, finds location of the rightmost 0.
```

```
		for(int i = 0; i < input.length; i++)
		{
			if(input[i] == 0)
			{
				zeroPlaceholder = i;
			}
		}

		if(zeroPlaceholder != 0) //if there is a 0 and it isn't the first digit
		{
			int[] holder = new int[input.length - zeroPlaceholder - 1];
			for(int j = 0; j < holder.length; j++) //cut the input from the beginning to the
rightmost 0
			{
				holder[j] = input[input.length - holder.length + j];
			}
			input = holder;
			inputLength = input.length;
			steps++;
		}
		else if(zeroPlaceholder == 0 && input.length == 1 && input[0] == 0) //if there is a
0 in the first part of a 1 digit array
		{
			int[] holder = new int[0];
			input = holder;
			steps++; //clear the array and add a step
		}

		checkIfClear();

		if(input.length !=0) //if the array isn't clear
		{
			int max = input[0];
			int placeholder = 0;

			for(int i = 1; i < inputLength; i++) //find the rightmost maximum and its
place
			{
				if(input[i] > max)
				{
					max = input[i];
				}
				if(input[i] == max)
```

```
                        {
                                placeholder = i;
                        }
                }

                if(input[placeholder] % 2 == 0) //if even, subtract max by 2
                {
                        input[placeholder] = input[placeholder] - 2;
                }
                else //if odd, subtract max by 1
                {
                        input[placeholder] = input[placeholder] - 1;
                }
                steps++;
        }
}

/**
        Checks if the solution has been solved.
*/
public void checkIfClear()
{
        if(input.length == 0)
                isCleared = true;
}

/**
        Returns if the solution has been solved.
        @return if the solution is solved
*/
public boolean isClear() {return isCleared;}

/**
        Returns the input length.
        @return the length of the input
*/
public int inpLength() {return inputLength;}

/**
        Returns if the number of steps taken to solve the problem.
        @return the number of steps
*/
public int howManySteps() {return steps;}
```

```
}
/* Saket Bakshi. AP Computer Science A. Deletions Easy. Due March 10, 2019.
This class tests the Deletions Easy problem.
*/

import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class Main
{
	public static void main(String[] args) throws FileNotFoundException
	{
		File inFile = new File("input.txt"); //input file and scanner
		Scanner scanned = new Scanner(inFile);


		for(int i = 0; i < 5; i++)
		{
			String string = scanned.nextLine().replaceAll("[^0-9]",""); //change line to
array of integers
			int[] array = new int[string.length()];
			for(int currentChar = 0; currentChar < string.length(); currentChar++) //fill
in array of integers from input
			{
				array[currentChar] = Integer.parseInt(string.substring(currentChar,
currentChar + 1));
			}

			DeletionsEasyV2 tester = new DeletionsEasyV2(array); //make Deletions
object

			while(!tester.isClear()) //do while the array isn't clear
			{
				tester.doTurn();
			}

			System.out.println(tester.howManySteps());
		}
		scanned.close();
	}
}
```


AutoGradr
DASHBOARD > APCSA-2018-P6 > DELETIONS EASY
PROJECTS
Deletions Easy
SAKET BAKSHI
Help / Docs
Password Change
Logout
Deletions Easy
Look on canvas or precise instructions
Test Case Harder
Test Case 1 Original
Console
1 14
2 27
3 3
4 25
5 19
6
Input Files
input.txt
Help
Java 8
Drag and drop
or
browse
You passed 2 of 2 test cases
Test Case Harder
Test Case 1 Original
Passed

Deletions Hard

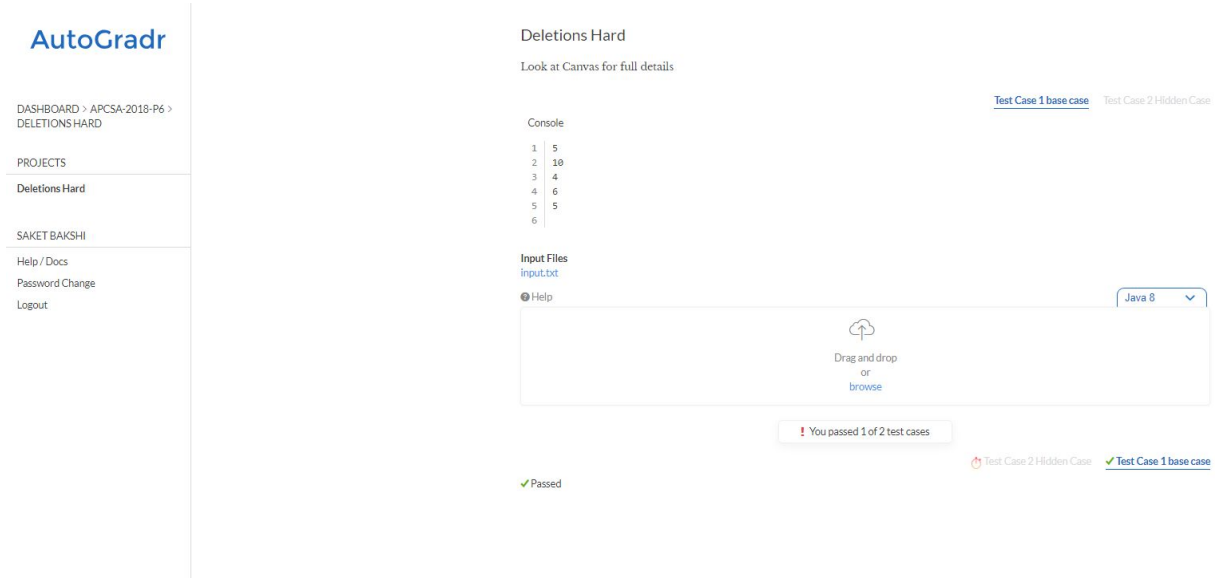
```
/* Saket Bakshi. AP Computer Science A. Deletions Hard. Due March 10, 2019.
This class solves the Deletions Hard problem.
*/
public class DeletionsHard
{
	private boolean isCleared; //variable to judge if problem is solved
	private int[] input; //holds the problem sequence
	private int inputLength; //length of sequence
	private int steps; //steps to solve sequence

	/**
		This takes an integer array and solves it according to the rules of the deletions hard directions.
		@param initialSequence the initial array
	*/
	public DeletionsHard(int[] initialSequence)
	{
		isCleared = false;
		input = initialSequence;
		inputLength = initialSequence.length;
		steps = 0;
	}

	/**
		This takes an integer array and solves it according to the rules of the deletions hard directions.
	*/
	public DeletionsHard()
	{
		isCleared = false;
		input = new int[100];
		inputLength = 0;
		steps = 0;
	}

	/**
		This solves the sequence
	*/
	public void doTurn()
	{
		int zeroPlaceholder = 0; //sees if there are zeros, finds location of the rightmost 0.
		for(int i = 0; i < input.length; i++)
```

```
		{
			if(input[i] == 0)
			{
				zeroPlaceholder = i;
			}
		}

		if(zeroPlaceholder != 0 || (input[0] == 0 && input.length > 1)) //if there is a 0 and it
isn't the first digit
		{
			int[] holder = new int[input.length - zeroPlaceholder - 1];
			for(int j = 0; j < holder.length; j++) //cut the input from the beginning to the
rightmost 0
			{
				holder[j] = input[input.length - holder.length + j];
			}
			input = holder;
			inputLength = input.length;
			steps++;
		}
		else if(zeroPlaceholder == 0 && input.length == 1 && input[0] == 0) //if there is a
0 in the first part of a 1 digit array
		{
			int[] holder = new int[0];
			input = holder;
			steps++; //clear the array and add a step
		}

		checkIfClear();

		if(input.length !=0) //if the array isn't clear
		{
			int common = 0;
			int numberOfCommon = 0;
			for(int i = 0; i <= 9; i++) //goes through each digit from 0-9 to find how
often each appears
			{
				int numberOfCurrent = 0; //the amount of digits of the current digit
being searched for (i)
				for(int a = 0; a < input.length; a++)
				{
					if(input[a] == i) //finds number of occurences of current digit
						numberOfCurrent++;
```

```
                                }
                                if(numberOfCurrent > numberOfCommon) //if has most
occurences, it is saved as most common, with number of occurences saved
                                {
                                        numberOfCommon = numberOfCurrent;
                                        common = i;
                                }
                                else if(numberOfCurrent >= numberOfCommon && i > common
&& numberOfCurrent != 0) //if has an equal number of appearances to most appeared but is
larger, this number is saved
                                {
                                        numberOfCommon = numberOfCurrent;
                                        common = i;
                                }
                        }
                        int commonPlaceholder = 0;
                        for(int i = 0; i < input.length; i++) //finds rightmost occurence of most
common number
                        {
                                if(input[i] == common)
                                        commonPlaceholder = i;
                        }
                        if(common % 2 == 0)
                                        input[commonPlaceholder] = input[commonPlaceholder] -
2;
                        else
                                input[commonPlaceholder] = input[commonPlaceholder] - 1;
                        steps++;
                }
        }

        /**
                Checks if the solution has been solved.
        */
        public void checkIfClear()
        {
                if(input.length == 0)
                        isCleared = true;
        }

        /**
                Returns if the solution has been solved.
                @return if the solution is solved
```

```
	*/
	public boolean isClear() {return isCleared;}

	/**
		Returns the input length.
		@return the length of the input
	*/
	public int inpLength() {return inputLength;}

	/**
		Returns if the number of steps taken to solve the problem.
		@return the number of steps
	*/
	public int howManySteps() {return steps;}
}
/* Saket Bakshi. AP Computer Science A. Deletions Hard. Due March 10, 2019.
This class tests the Deletions Hard problem.
*/
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class Main
{
	public static void main(String[] args) throws FileNotFoundException
	{
		File inFile = new File("input.txt"); //input file and scanner
		Scanner scanned = new Scanner(inFile);
		Scanner amountOfLinesTester = new Scanner(inFile);
		int numberOfLines = 0;
		while(amountOfLinesTester.hasNextLine())
		{
			numberOfLines++;
			amountOfLinesTester.nextLine();
		}


		for(int i = 0; i < numberOfLines; i++)
		{
			String string = scanned.nextLine().replaceAll("[^0-9]",""); //change line to
array of integers
			int[] array = new int[string.length()];
```

```
                for(int currentChar = 0; currentChar < string.length(); currentChar++) //fill
in array of integers from input
                {
                        array[currentChar] = Integer.parseInt(string.substring(currentChar,
currentChar + 1));
                }

                DeletionsHard tester = new DeletionsHard(array); //make Deletions object

                while(!tester.isClear()) //do while the array isn't clear
                {
                        tester.doTurn();
                }

                System.out.println(tester.howManySteps());
        }
        scanned.close();
    }
}
```



Mowing Lab

```
/* Saket Bakshi. AP Computer Science A. Mowing. Due March 10, 2019.
This class solves the Mowing problem.
*/
import java.util.ArrayList;
import java.awt.Rectangle;
import java.awt.Point;

public class LawnMower
{
	private ArrayList<Point> horizontalBorders; //makes the lawn for the array
	private ArrayList<Point> verticalBorders;
	private String[][] lawn;
	private Rectangle mower; //makes object for the mower
	private static int counter;

	/**
		Creates a LawnMower object. This class has a mower, a lawn, and methods to create the lawn, move the mower, return info about the whole object, cut the lawn, and detect the surroundings of the mower.
		@param xPosition the x-coordinate position of the middle of the mower
		@param yPosition the y-coordinate position of the middle of the mower
		@param lawn the two-dimensional String array to show if the lawn is cut, where the trees are, and where the mower is
		@param length the vertical dimension of the lawn
		@param width the horizontal dimension of the lawn
	*/
	public LawnMower(int xPosition, int yPosition, String[][] lawn, int length, int width)
	{
		horizontalBorders = new ArrayList<>();
		verticalBorders = new ArrayList<>();
		this.lawn = lawn;
		mower = new Rectangle(xPosition, yPosition, 3, 3);
		makeTopBottomBorders(length, width);
		makeRightLeftBorders(length, width);
		counter = 0;
	}

	/**
		Returns the top and bottom borders of the lawn
		@return the arraylist for the top and bottom borders of the lawn
	*/
	public ArrayList<Point> getTopBottomBorders() {return horizontalBorders;}
```

```
/**
	Returns the left and right borders of the lawn
	@return the arraylist for the left and right borders of the lawn
*/
public ArrayList<Point> getRightLeftBorders() {return verticalBorders;}

/**
	Creates the top and bottom borders of the lawn
*/
private void makeTopBottomBorders(int length, int width)
{
	for(int i = 0; i < length; i++)
	{
		for(int j = 0; j < width; j++)
		{
			if(i == 0)
			{
				Point p = new Point(j, i+1);
				horizontalBorders.add(p);
			}
			if(i == length - 1)
			{
				Point p = new Point(j, i-1);
				horizontalBorders.add(p);
			}
		}
	}
}

/**
	Creates the right and left borders of the lawn
*/
private void makeRightLeftBorders(int length, int width)
{
	for(int i = 0; i < length; i++)
	{
		for(int j = 0; j < width; j++)
		{
			if(j == 0)
			{
				Point p = new Point(j+1, i);
				verticalBorders.add(p);
```

```
                        }
                        if(j == width - 1)
                        {
                                Point p = new Point(j-1, i);
                                verticalBorders.add(p);
                        }
                }
        }
}

/**
        Sets the location of the mower using integers as x and y coordinates
*/
public void setLocation(int x, int y) {mower.setLocation(x, y);}

/**
        Sets the location of the mower using Points for x and y coordinates
*/
public void setLocation(Point p) {mower.setLocation(p);}

/**
        Moves the mower one space down
*/
public void moveDown() {mower.translate(0, 1);}

/**
        Moves the mower one space up
*/
public void moveUp() {mower.translate(0, -1);}
/**
        Moves the mower one space to the right
*/
public void moveRight() {mower.translate(1, 0);}

/**
        Moves the mower one space to the left
*/
public void moveLeft() {mower.translate(-1,0);}

/**
        Returns the Rectangle object's data of the mower
        @return the information of the Rectangle object of the mower
*/
```

```
public Rectangle getMower() {return mower;}

/**
	Returns the location of the mower
	@return the location of the mower
*/
public Point getLocation() {return mower.getLocation();}

/**
	Returns the two-dimensional String array of the lawn
	@return a 2D array of the lawn
*/
public String[][] getLawn() {return lawn;}

/**
	Changes the area around the mower to cut grass
*/
public void cut()
{
	lawn[(int)mower.getY()][(int)mower.getX()] = "C";
	lawn[(int)mower.getY()+1][(int)mower.getX()] = "C";
	lawn[(int)mower.getY()-1][(int)mower.getX()] = "C";
	lawn[(int)mower.getY()][(int)mower.getX()-1] = "C";
	lawn[(int)mower.getY()+1][(int)mower.getX()-1] = "C";
	lawn[(int)mower.getY()-1][(int)mower.getX()-1] = "C";
	lawn[(int)mower.getY()][(int)mower.getX()+1] = "C";
	lawn[(int)mower.getY()-1][(int)mower.getX()+1] = "C";
	lawn[(int)mower.getY()+1][(int)mower.getX()+1] = "C";
}

/**
	Returns true if the mower is at the top or bottom border
	@return if the mower is at the border
*/
public boolean atTopBottomBorder()
{
	for(int i = 0; i < horizontalBorders.size(); i++)
	{
		if(mower.getLocation().equals(horizontalBorders.get(i)))
		{
			return true;
		}
	}
```

```
		return false;
	}

	/**
		Returns true if the mower is at the left or right border
		@return if the mower is at the border
	*/
	public boolean atRightLeftBorder()
	{
		for(int i = 0; i < verticalBorders.size(); i++)
		{
			if(mower.getLocation().equals(verticalBorders.get(i)))
			{
				return true;
			}
		}
		return false;
	}

	/**
		Returns true if the mower is at the bottom border
		@return if the mower is at the bottom border
	*/
	public boolean atBottomBorder() {return (int)mower.getLocation().getY() == lawn.length -
2;}

	/**
		Returns true if the mower is at the top border
		@return if the mower is at the top border
	*/
	public boolean atTopBorder() {return (int)mower.getLocation().getY() == 1;}

	/**
		Returns true if the mower is at the right border
		@return if the mower is at the right border
	*/
	public boolean atRightBorder() {return (int)mower.getLocation().getX() == lawn[0].length
- 2;}

	/**
		Returns true if the mower is at the left border
		@return if the mower is at the left border
	*/
```

```
public boolean atLeftBorder() {return (int)mower.getLocation().getX() == 1;}

/**
	Returns true if a tree is above the mower
	@return if there is a tree above
*/
public boolean treeAbove()
{
	if(atTopBorder())
	{
		return false;
	}
	if(lawn[(int)mower.getY()-2][(int)mower.getX()].equals("T"))
	{
		return true;
	}
	if(lawn[(int)mower.getY()-2][(int)mower.getX()+1].equals("T"))
	{
		return true;
	}
	if(lawn[(int)mower.getY()-2][(int)mower.getX()-1].equals("T"))
	{
		return true;
	}
	return false;
}

/**
	Returns true if a tree is below the mower
	@return if there is a tree below
*/
public boolean treeBelow()
{
	if(atBottomBorder())
	{
		return false;
	}
	if(lawn[(int)mower.getY()+2][(int)mower.getX()].equals("T"))
	{
		return true;
	}
	if(lawn[(int)mower.getY()+2][(int)mower.getX()+1].equals("T"))
	{
```

```
                return true;
        }
        if(lawn[(int)mower.getY()+2][(int)mower.getX()-1].equals("T"))
        {
                return true;
        }
        return false;
}

/**
        Moves the mower up as long as nothing is in the way or there is no border
*/
public void toTheTop()
{
        while(true)
        {
                if(atTopBorder())
                {
                        break;
                }
                if(treeAbove())
                {
                        break;
                }
                else
                {
                        moveUp();
                }
        }
}

/**
        Checks if right side is open to mow
        @return if the right is open to mow
*/
public boolean rightIsOpen()
{
        if(atRightBorder()) //checks for borders
        {
                return false;
        }
        if(lawn[(int)mower.getY()][(int)mower.getX()+2].equals("T")) //checks for trees
        {
```

```
                return false;
        }
        if(lawn[(int)mower.getY()+1][(int)mower.getX()+2].equals("T"))
        {
                return false;
        }
        if(lawn[(int)mower.getY()-1][(int)mower.getX()+2].equals("T"))
        {
                return false;
        }
        return true;
}

/**
        Checks if left side is open to mow
        @return if the left is open to mow
*/
public boolean leftIsOpen()
{
        if(atLeftBorder()) //checks for border
        {
                return false;
        }
        if(lawn[(int)mower.getY()][(int)mower.getX()-2].equals("T")) //checks for trees
        {
                return false;
        }
        if(lawn[(int)mower.getY()+1][(int)mower.getX()-2].equals("T"))
        {
                return false;
        }
        if(lawn[(int)mower.getY()-1][(int)mower.getX()-2].equals("T"))
        {
                return false;
        }
        return true;
}

/**
        Moves mower up until the right is open
*/
public void checkRightToTheTop(Point temp)
{
```

```
        if(!rightIsOpen())
        {
                while(!atTopBorder() && !(mower.getLocation().equals(temp)))
                {
                        if(rightIsOpen())
                        {
                                mower.translate(1, 0);
                                break;
                        }
                        else
                        {
                                moveUp();
                        }
                }
        }
        else
        {
                mower.translate(1, 0);
        }
}

/**
        Moves mower down until the right is open
*/
public void checkRightToTheBottom(Point temp)
{
        if(!rightIsOpen())
        {
                while(!atBottomBorder() && !(mower.getLocation().equals(temp)))
                {
                        if(rightIsOpen())
                        {
                                mower.translate(1, 0);
                                break;
                        }
                        else
                        {
                                moveDown();
                        }
                }
        }
        else
        {
```

```
                mower.translate(1, 0);
        }
}

/**
        Moves mower up until the left is open.
*/
public void checkLeftToTheTop(Point temp)
{
        if(!leftIsOpen())
        {
                while(!atTopBorder() && !(mower.getLocation().equals(temp))) //nottop
                {
                        if(leftIsOpen())
                        {
                                mower.translate(-1, 0);
                                break;
                        }
                        else
                        {
                                moveUp();
                        }
                }
        }
        else
        {
                mower.translate(-1, 0);
        }
}

/**
        Moves mower down until the left is open.
*/
public void checkLeftToTheBottom(Point temp)
{
        if(!leftIsOpen())
        {
                while(!atBottomBorder() && !(mower.getLocation().equals(temp))) //nottop
                {
                        if(leftIsOpen())
                        {
                                mower.translate(-1, 0);
                                break;
```

```
                        }
                        else
                        {
                                moveDown();
                        }
                }
        }
        else
        {
                mower.translate(-1, 0);
        }
}

/**
        Fills in a section of the lawn according to a top-right corner coordinate and a rectangular area to fill
*/
public void fillSection(int i, int j, String[][] section)
{
        for(int a = 0; a < section.length; a++)
        {
                for(int b = 0; b < section[0].length; b++)
                {
                        section[a][b] = lawn[a+i][b+j];
                }
        }
}

/**
        Checks to see if a section has either a tree or hasn't been mowed
        @return if the section can be mowed
*/
public boolean onlyC(String[][] section)
{
        for(int i = 0; i < section.length; i++)
        {
                for(int j = 0; j < section[0].length; j++)
                {
                        if(section[i][j].equals(".") || section[i][j].equals("T"))
                        {
                                return false;
                        }
                }
```

```
                }
                return true;
        }

        /**
                Scans to find what sections can be mowed
                @return the point of an area that can be mowed
        */
        public Point scan()
        {
                int countingTo = counter;
                Point p = new Point(-1, -1);
                String[][] section = new String[3][3];
                for(int i = 0; i < lawn.length-2; i++)
                {
                        for(int j = 0; j < lawn[0].length-2; j++)
                        {
                                fillSection(i, j, section);
                                if(onlyC(section))
                                {
                                        if(countingTo == 0)
                                        {
                                                Point x = new Point(j+1, i+1);
                                                counter++;
                                                return x;
                                        }
                                        else
                                        {
                                                countingTo--;
                                        }
                                }
                        }
                }
                return p;
        }
}
/* Saket Bakshi. AP Computer Science A. Mowing. Due March 10, 2019.
This class tests the Mowing problem.
*/
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.StringTokenizer;
```

```
import java.util.ArrayList;
import java.awt.Point;

public class Main
{
	public static void main(String[] args) throws FileNotFoundException
	{
		File text = new File("input.txt"); //inputs the file
		Scanner in = new Scanner(text); //creates scanner object

		int loops = Integer.parseInt(in.nextLine()); //takes number of test cases
		for(int a = 0; a < loops; a++)
		{
			StringTokenizer mowerCoordinatesAndLawnSize = new
StringTokenizer(in.nextLine(), " ");
			int length =
Integer.parseInt(mowerCoordinatesAndLawnSize.nextToken());
			int width =
Integer.parseInt(mowerCoordinatesAndLawnSize.nextToken());
			int rowPos =
Integer.parseInt(mowerCoordinatesAndLawnSize.nextToken());
			int colPos =
Integer.parseInt(mowerCoordinatesAndLawnSize.nextToken());

			String[][] lawn = new String[length][width]; //constructs 2D lawn and fills it
in
			for(int i = 0; i < length; i++)
			{
				StringTokenizer lawnCreator = new StringTokenizer(in.nextLine(),
" ");
				for(int j = 0; j < width; j++)
				{
					lawn[i][j] = lawnCreator.nextToken();
				}
			}
			LawnMower tester = new LawnMower(colPos, rowPos, lawn, length,
width);

			//mows everywhere possible without jumping the mower
			mowToTheRight(tester);
			tester.setLocation(colPos, rowPos);
			mowToTheLeft(tester);
			tester.setLocation(colPos, rowPos);
```

```
                otherMowToTheRight(tester);
                tester.setLocation(colPos, rowPos);
                otherMowToTheLeft(tester);

                for(int x = 0; x < 1000; x++)
                {
                        Point scanned = tester.scan();
                        Point stop = new Point(-1, -1);
                        if(!scanned.equals(stop))
                        {
                                tester.setLocation(scanned);
                                mowToTheRight(tester);
                                tester.setLocation(scanned);
                                mowToTheLeft(tester);
                                tester.setLocation(scanned);
                                otherMowToTheRight(tester);
                                tester.setLocation(scanned);
                                otherMowToTheLeft(tester);
                        }
                }

                printArray(tester.getLawn());
                if (a < loops-1)
                System.out.println();
        }
}


public static void mowToTheRight(LawnMower mower)
{
        mower.toTheTop();
        Point temp = mower.getLocation();
        do
        {
                mower.cut();
                if(!mower.atBottomBorder() && !mower.treeBelow())
                {
                        mower.moveDown();
                }
        }
        while(!mower.treeBelow() && !mower.atTopBottomBorder());
        mower.cut();
```

```
        mower.checkRightToTheTop(temp);

        if(temp.equals(mower.getLocation()) && !mower.rightIsOpen())
        {
                mowToTheLeft(mower);
        }
        else if(mower.rightIsOpen() && temp.equals(mower.getLocation()))
        {
                mower.moveRight();
                mowToTheRight(mower);
        }
        else
        {
                mowToTheRight(mower);
        }
}

public static void otherMowToTheRight(LawnMower mower)
{
        mower.toTheTop();
        do
        {
                mower.cut();
                if(!mower.atBottomBorder() && !mower.treeBelow())
                {
                        mower.moveDown();
                }
        }
        while(!mower.treeBelow() && !mower.atTopBottomBorder());
        mower.cut();
        Point temp2 = mower.getLocation();

        mower.toTheTop();
        mower.checkRightToTheBottom(temp2);

        if(temp2.equals(mower.getLocation()) && !mower.rightIsOpen())
        {
                otherMowToTheLeft(mower);
        }
        else if(mower.rightIsOpen() && temp2.equals(mower.getLocation()))
        {
                mower.moveRight();
                otherMowToTheRight(mower);
```

```
        }
        else
        {
                otherMowToTheRight(mower);
        }
}

public static void mowToTheLeft(LawnMower mower)
{
        mower.toTheTop();
        Point temp = mower.getLocation();
        do
        {
                mower.cut();
                if(!mower.atBottomBorder() && !mower.treeBelow())
                {
                        mower.moveDown();
                }
                else
                {
                        break;
                }
        }
        while(!mower.treeBelow() && !mower.atTopBottomBorder());
        mower.cut();

        mower.checkLeftToTheTop(temp);

        if(temp.equals(mower.getLocation()) && !mower.leftIsOpen())
        {

        }
        else if(temp.equals(mower.getLocation()) && mower.leftIsOpen())
        {
                mower.moveLeft();
                mowToTheLeft(mower);
        }
        else
        {
                mowToTheLeft(mower);
        }
}
```

```
public static void otherMowToTheLeft(LawnMower mower)
{
	mower.toTheTop();
	do
	{
		mower.cut();
		if(!mower.atBottomBorder() && !mower.treeBelow())
		{
			mower.moveDown();
		}
		else
		{
			break;
		}
	}
	while(!mower.treeBelow() && !mower.atTopBottomBorder());
	mower.cut();
	Point temp2 = mower.getLocation();

	mower.toTheTop();
	mower.checkLeftToTheBottom(temp2);

	if(temp2.equals(mower.getLocation()) && !mower.leftIsOpen())
	{

	}
	else if(temp2.equals(mower.getLocation()) && mower.leftIsOpen())
	{
		mower.moveLeft();
		otherMowToTheLeft(mower);
	}
	else
	{
		otherMowToTheLeft(mower);
	}
}

public static void printArray(String[][] array)
{
	for(int i = 0; i < array.length; i++)
	{
		for(int j = 0; j < array[0].length; j++)
		{
```

```
                System.out.print(array[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```