Name : Tanmay Deokar
Class : BE Comp SS

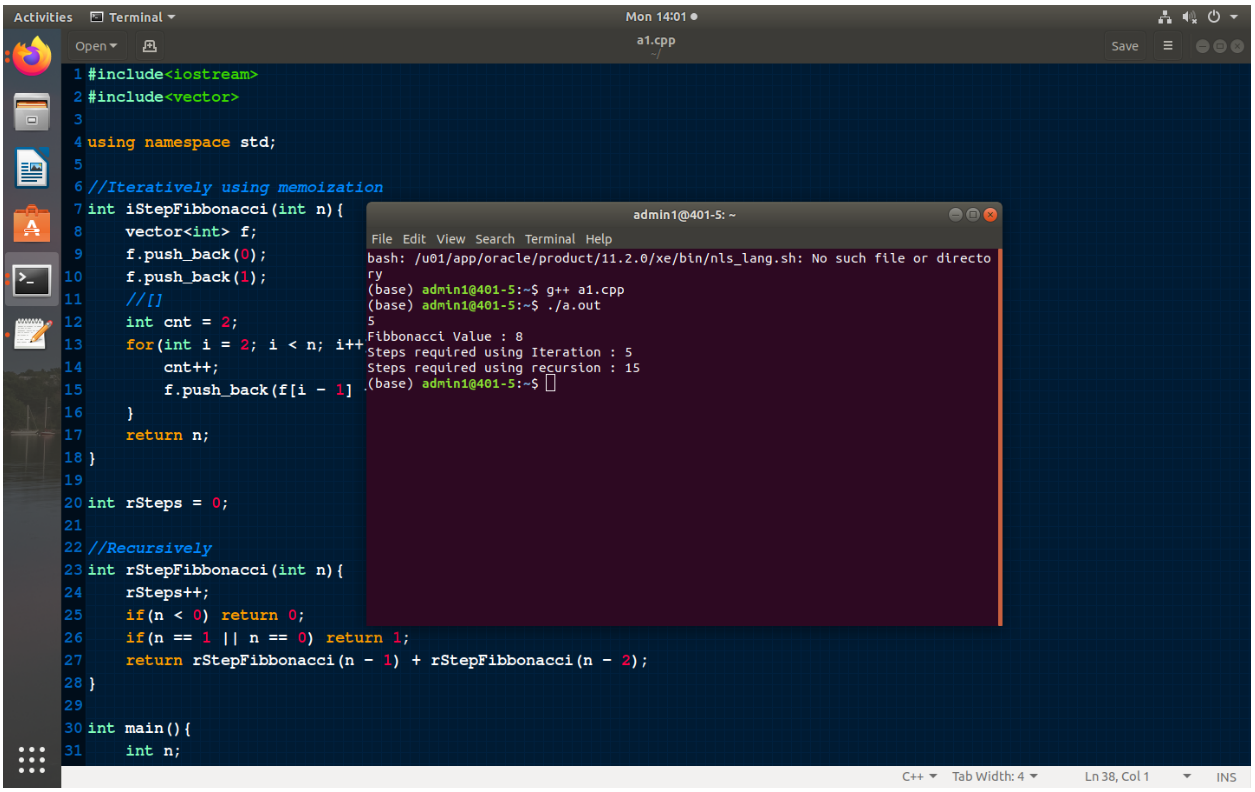
Assignment A1

Code :

```
#include<iostream>
#include<vector>

using namespace std;

//Iteratively using memoization
int iStepFibbonacci(int n){
	vector<int> f;
	f.push_back(0);
	f.push_back(1);
	//[]
	int cnt = 2;
	for(int i = 2; i < n; i++){
	cnt++;
	f.push_back(f[i - 1] + f[i - 2]);
	}
	return n;
}

int rSteps = 0;

//Recursively
int rStepFibbonacci(int n){
	rSteps++;
	if(n < 0) return 0;
	if(n == 1 || n == 0) return 1;
	return rStepFibbonacci(n - 1) + rStepFibbonacci(n - 2);
}

int main(){
	int n;
	cin >> n;
	cout << "Fibbonacci Value : " << rStepFibbonacci(n) << '\n';
	cout << "Steps required using Iteration : " << iStepFibbonacci(n) << '\n';
```

```
        cout << "Steps required using recursion : " << rSteps << '\n';
        return 0;
}
```

Output :



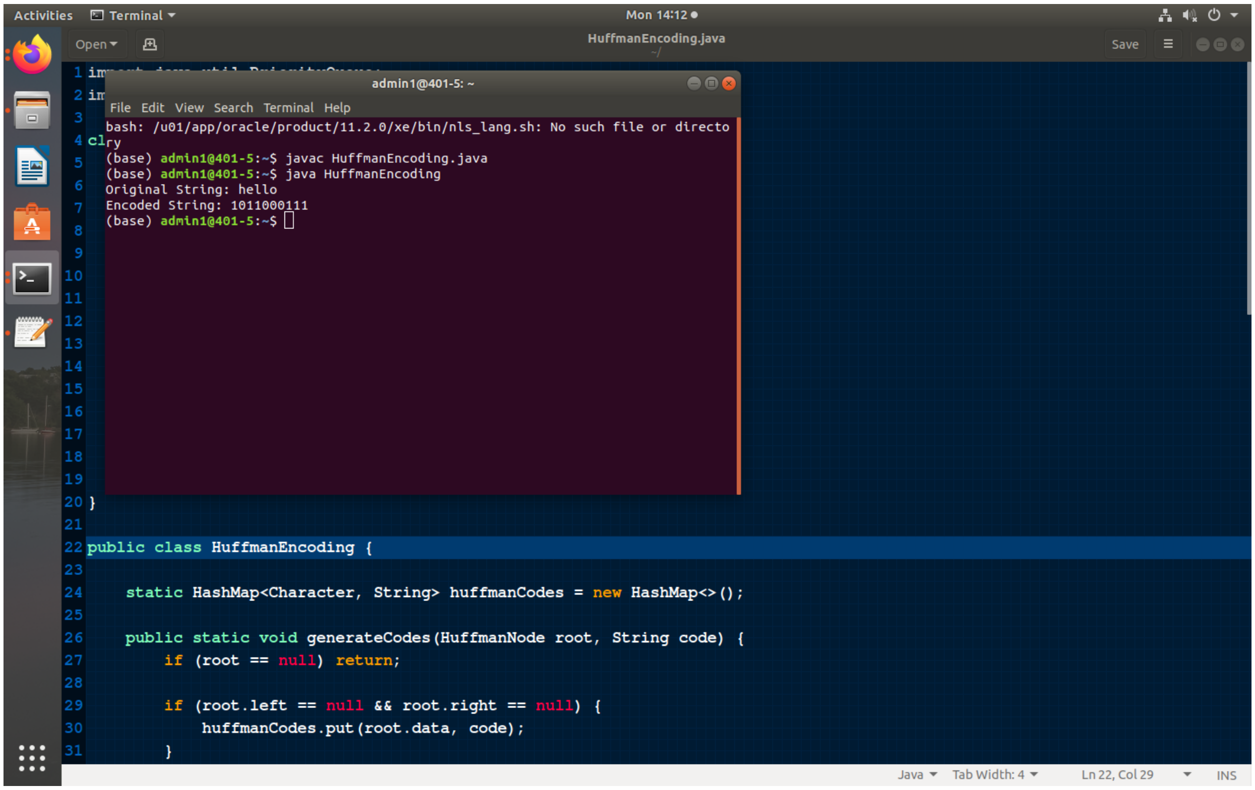
A2:

```
import java.util.PriorityQueue;
import java.util.HashMap;

class HuffmanNode implements Comparable<HuffmanNode> {
        int frequency;
        char data;
        HuffmanNode left, right;

        public HuffmanNode(int frequency, char data) {
        this.frequency = frequency;
```

```java
        this.data = data;
        this.left = null;
        this.right = null;
        }

        @Override
        public int compareTo(HuffmanNode node) {
        return this.frequency - node.frequency;
        }
}

public class HuffmanEncoding {

        static HashMap<Character, String> huffmanCodes = new HashMap<>();

        public static void generateCodes(HuffmanNode root, String code) {
        if (root == null) return;

        if (root.left == null && root.right == null) {
        huffmanCodes.put(root.data, code);
        }

        generateCodes(root.left, code + "0");
        generateCodes(root.right, code + "1");
        }

        public static void buildHuffmanTree(String input) {
        HashMap<Character, Integer> frequencyMap = new HashMap<>();

        for (char c : input.toCharArray()) {
        frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);
        }

        PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();

        for (char key : frequencyMap.keySet()) {
        priorityQueue.add(new HuffmanNode(frequencyMap.get(key), key));
        }

        while (priorityQueue.size() > 1) {
        HuffmanNode left = priorityQueue.poll();
        HuffmanNode right = priorityQueue.poll();

        int sum = left.frequency + right.frequency;
```

```
        HuffmanNode newNode = new HuffmanNode(sum, '-');
        newNode.left = left;
        newNode.right = right;

        priorityQueue.add(newNode);
        }

        HuffmanNode root = priorityQueue.poll();
        generateCodes(root, "");
        }

        public static String encode(String input) {
        StringBuilder encodedString = new StringBuilder();

        for (char c : input.toCharArray()) {
        encodedString.append(huffmanCodes.get(c));
        }

        return encodedString.toString();
        }

        public static void main(String[] args) {
        String input = "hello";
        buildHuffmanTree(input);

        String encodedString = encode(input);

        System.out.println("Original String: " + input);
        System.out.println("Encoded String: " + encodedString);
        }
}
```

Output :



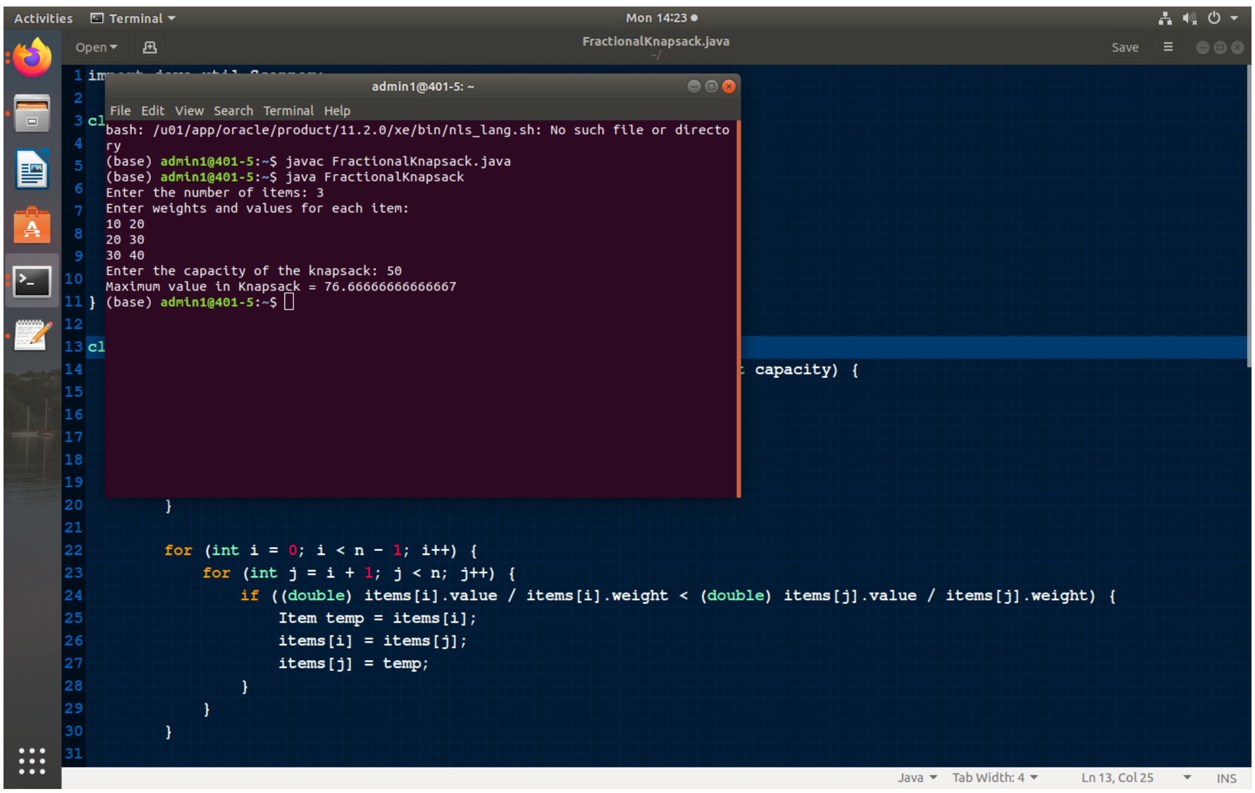
A3 :

Code :

```
import java.util.Scanner;

class Item {
        int weight;
        int value;

        public Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
        }
}

class FractionalKnapsack {
        public static double getMaxValue(int[] weights, int[] values, int capacity) {
        int n = weights.length;
        Item[] items = new Item[n];

        for (int i = 0; i < n; i++) {
```

```
        items[i] = new Item(weights[i], values[i]);
        }

        for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
                if ((double) items[i].value / items[i].weight < (double) items[j].value /
items[j].weight) {
                Item temp = items[i];
                items[i] = items[j];
                items[j] = temp;
                }
        }
        }

        double totalValue = 0;
        int currentWeight = 0;

        for (Item item : items) {
        if (currentWeight + item.weight <= capacity) {
                currentWeight += item.weight;
                totalValue += item.value;
        } else {
                int remainingCapacity = capacity - currentWeight;
                totalValue += (double) item.value * remainingCapacity / item.weight;
                break;
        }
        }

        return totalValue;
        }

        public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();

        int[] weights = new int[n];
        int[] values = new int[n];

        System.out.println("Enter weights and values for each item:");
        for (int i = 0; i < n; i++) {
        weights[i] = scanner.nextInt();
        values[i] = scanner.nextInt();
```

```
        }

        System.out.print("Enter the capacity of the knapsack: ");
        int capacity = scanner.nextInt();

        double maxValue = getMaxValue(weights, values, capacity);

        System.out.println("Maximum value in Knapsack = " + maxValue);
        }
}
```



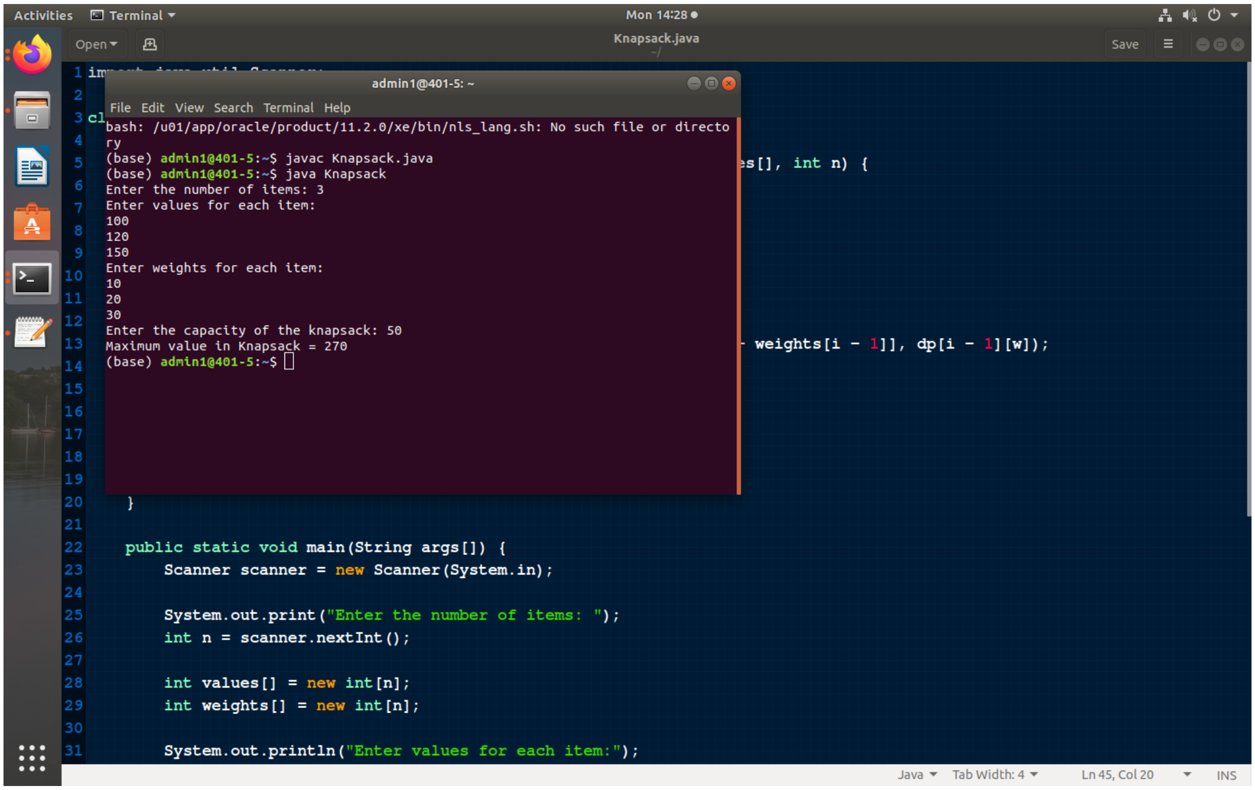
A4 :

Code :

```
import java.util.Scanner;

class Knapsack {

        public static int knapSack(int capacity, int weights[], int values[], int n) {
        int[][] dp = new int[n + 1][capacity + 1];
```

```
        for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
                if (i == 0 || w == 0)
                dp[i][w] = 0;
                else if (weights[i - 1] <= w)
                dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                else
                dp[i][w] = dp[i - 1][w];
        }
        }

        return dp[n][capacity];
        }

        public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();

        int values[] = new int[n];
        int weights[] = new int[n];

        System.out.println("Enter values for each item:");
        for (int i = 0; i < n; i++) {
        values[i] = scanner.nextInt();
        }

        System.out.println("Enter weights for each item:");
        for (int i = 0; i < n; i++) {
        weights[i] = scanner.nextInt();
        }

        System.out.print("Enter the capacity of the knapsack: ");
        int capacity = scanner.nextInt();

        int maxValue = knapSack(capacity, weights, values, n);
        System.out.println("Maximum value in Knapsack = " + maxValue);
        }
}
```

Output :



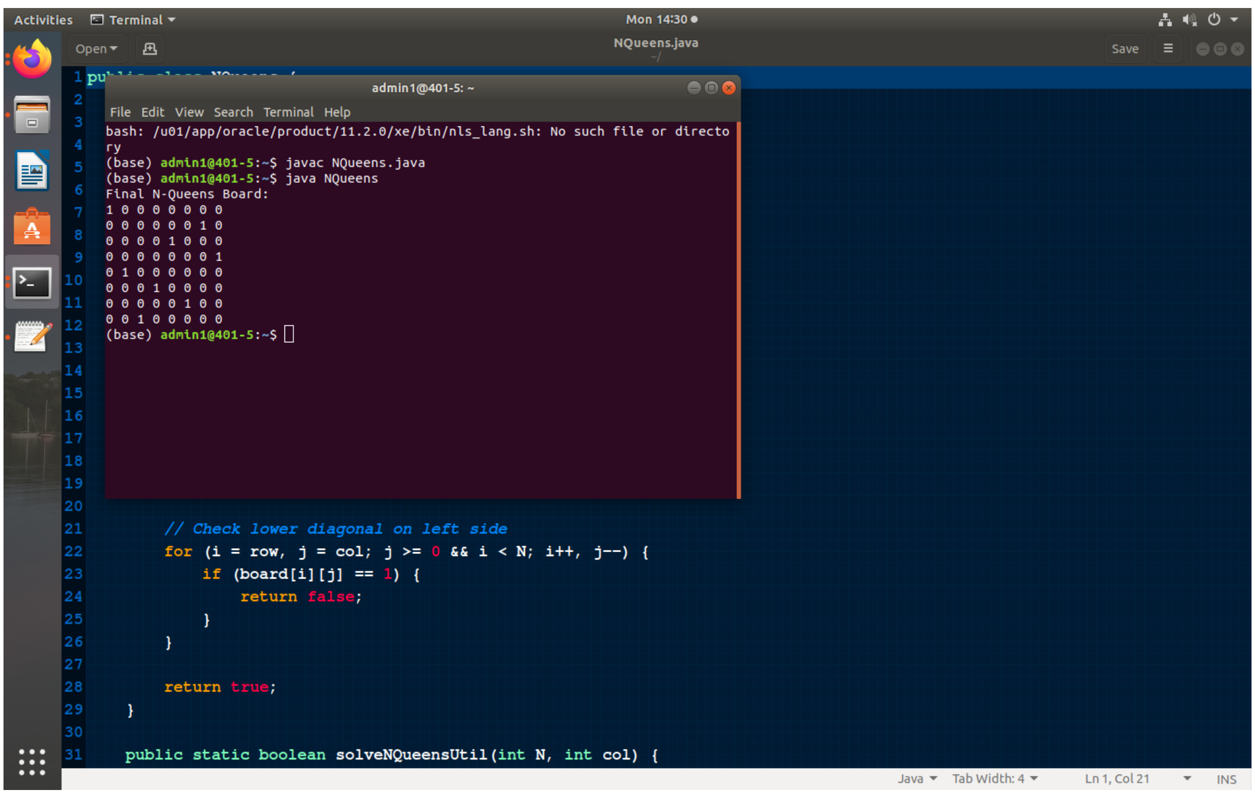
A5 :

```
public class NQueens {
        static int[][] board;

        public static boolean isSafe(int row, int col, int N) {
        int i, j;

        // Check the left side of the current row
        for (i = 0; i < col; i++) {
        if (board[row][i] == 1) {
                return false;
        }
        }

        // Check upper diagonal on left side
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
                return false;
        }
        }
```

```
// Check lower diagonal on left side
for (i = row, j = col; j >= 0 && i < N; i++, j--) {
if (board[i][j] == 1) {
        return false;
}
}

return true;
}

public static boolean solveNQueensUtil(int N, int col) {
if (col >= N) {
return true;
}

for (int i = 0; i < N; i++) {
if (isSafe(i, col, N)) {
        board[i][col] = 1;

        if (solveNQueensUtil(N, col + 1)) {
        return true;
        }

        board[i][col] = 0; // Backtrack
}
}

return false;
}

public static void solveNQueens(int N) {
board = new int[N][N];

// Place the first queen
board[0][0] = 1;

if (!solveNQueensUtil(N, 1)) {
System.out.println("Solution does not exist");
return;
}

// Print the final board
System.out.println("Final N-Queens Board:");
printBoard(N);
```

```
        }

        public static void printBoard(int N) {
        for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
                System.out.print(board[i][j] + " ");
        }
        System.out.println();
        }
        }

        public static void main(String[] args) {
        int N = 8; // Change N for different board sizes
        solveNQueens(N);
        }
}
```

Output :



MINI PROJECT :

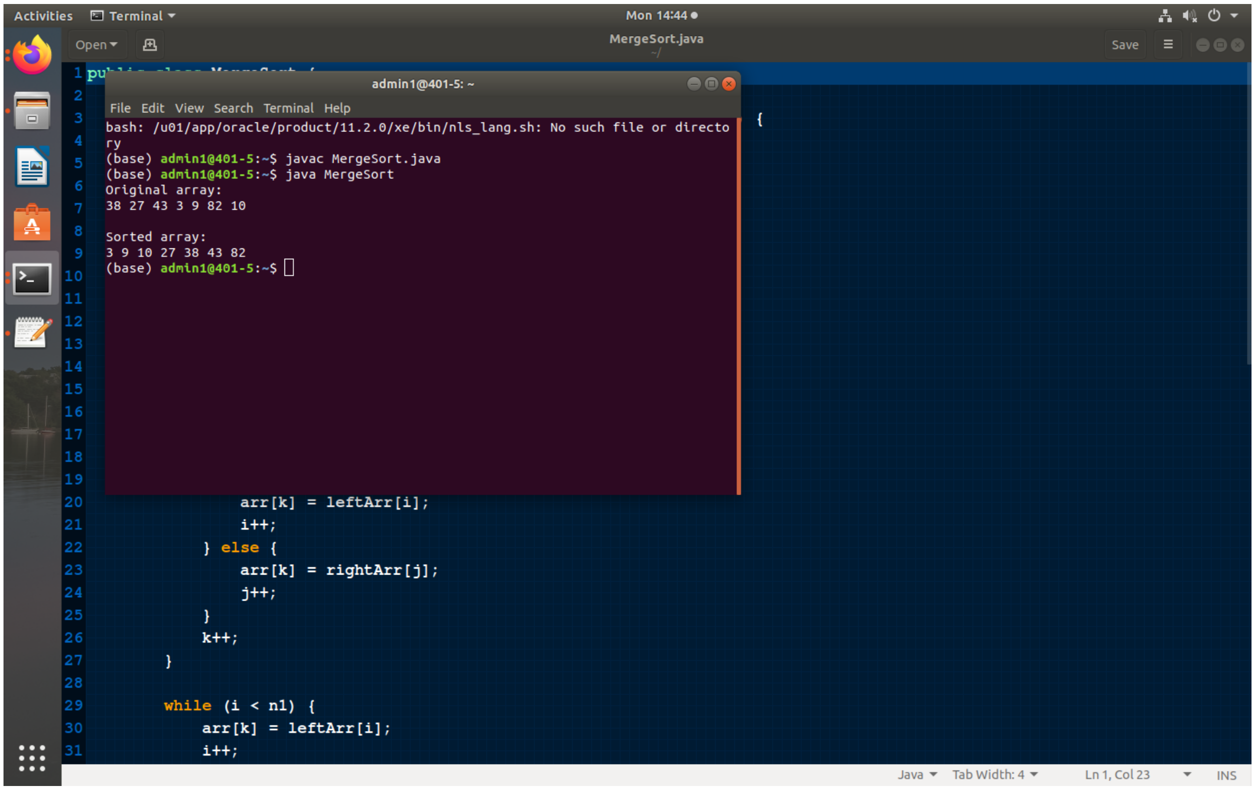
Mini Project - Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

Merge Sort :

```
public class MergeSort {

        public static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] leftArr = new int[n1];
        int[] rightArr = new int[n2];

        for (int i = 0; i < n1; ++i)
        leftArr[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
        rightArr[j] = arr[mid + 1 + j];

        int i = 0, j = 0;

        int k = left;
        while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
                arr[k] = leftArr[i];
                i++;
        } else {
                arr[k] = rightArr[j];
                j++;
        }
        k++;
        }

        while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
        }

        while (j < n2) {
```

```
        arr[k] = rightArr[j];
        j++;
        k++;
        }
        }

        public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
        int mid = (left + right) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
        }
        }

        public static void printArray(int[] arr) {
        for (int i : arr)
        System.out.print(i + " ");
        System.out.println();
        }

        public static void main(String[] args) {
        int[] arr = { 38, 27, 43, 3, 9, 82, 10 };

        System.out.println("Original array:");
        printArray(arr);

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array:");
        printArray(arr);
        }
}
```

Output :



Multithreaded merge sort :

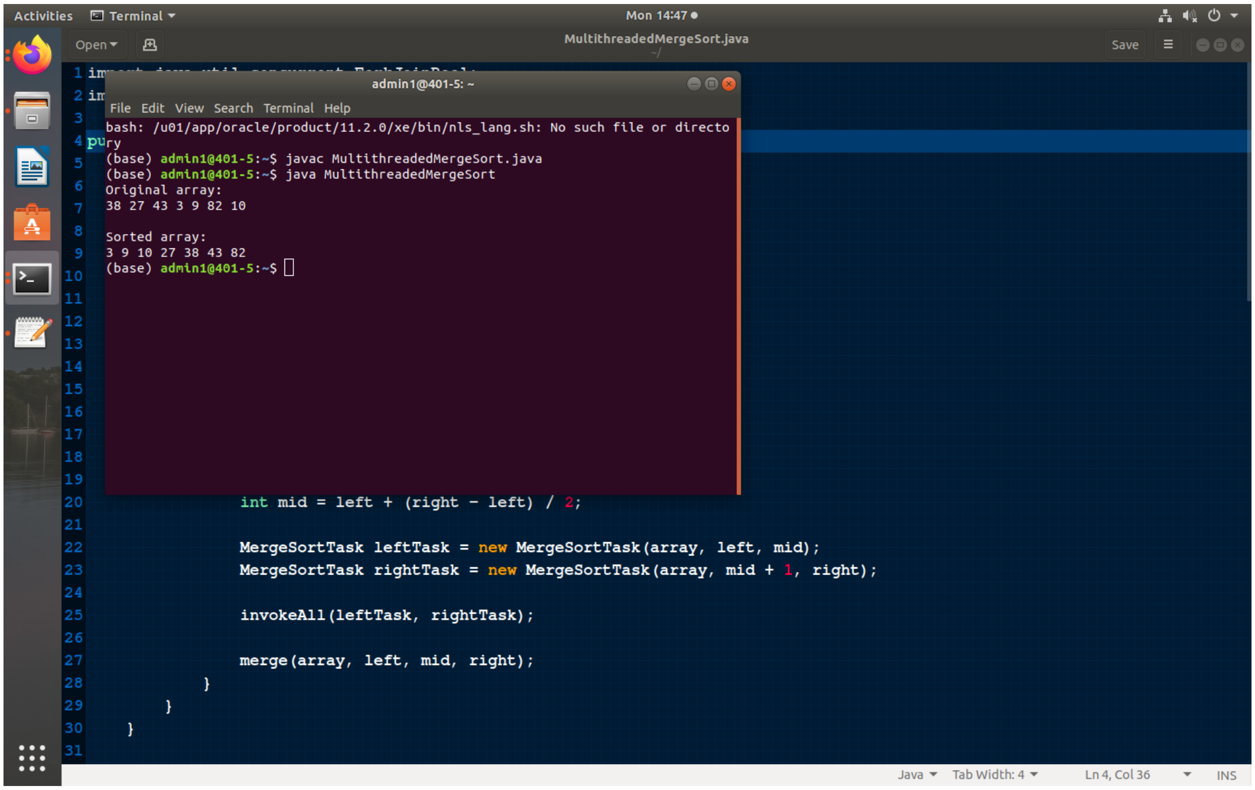
Code :

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class MultithreadedMergeSort {

    static class MergeSortTask extends RecursiveAction {
    private final int[] array;
    private final int left;
    private final int right;

    public MergeSortTask(int[] array, int left, int right) {
    this.array = array;
    this.left = left;
    this.right = right;
    }
```

```
@Override
protected void compute() {
if (left < right) {
        int mid = left + (right - left) / 2;

        MergeSortTask leftTask = new MergeSortTask(array, left, mid);
        MergeSortTask rightTask = new MergeSortTask(array, mid + 1, right);

        invokeAll(leftTask, rightTask);

        merge(array, left, mid, right);
}
}
}

public static void merge(int[] arr, int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;

int[] leftArr = new int[n1];
int[] rightArr = new int[n2];

for (int i = 0; i < n1; ++i)
leftArr[i] = arr[left + i];
for (int j = 0; j < n2; ++j)
rightArr[j] = arr[mid + 1 + j];

int i = 0, j = 0;

int k = left;
while (i < n1 && j < n2) {
if (leftArr[i] <= rightArr[j]) {
        arr[k] = leftArr[i];
        i++;
} else {
        arr[k] = rightArr[j];
        j++;
}
k++;
}

while (i < n1) {
arr[k] = leftArr[i];
i++;
```

```
        k++;
        }

        while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
        }
        }

        public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};

        System.out.println("Original array:");
        printArray(arr);

        ForkJoinPool pool = new ForkJoinPool();
        MergeSortTask task = new MergeSortTask(arr, 0, arr.length - 1);
        pool.invoke(task);

        System.out.println("\nSorted array:");
        printArray(arr);
        }

        public static void printArray(int[] arr) {
        for (int i : arr)
        System.out.print(i + " ");
        System.out.println();
        }
}
```

Output :



Time Complexity Analysis:

Sequential Merge Sort:

- Time Complexity: O(n log n)
- Best Case: O(n log n) - In all cases, the algorithm has to perform log n merge steps, and in each step, it takes linear time to merge.
- Worst Case: O(n log n) - Same as the best case. Merge Sort always performs the same number of operations regardless of the input.

Multithreaded Merge Sort:

- Time Complexity: O(n log n)
- Best Case: O(n log n) - Similar to the sequential case. The algorithm still has to perform log n merge steps, but with the potential for parallelism.
- Worst Case: O(n log n) - Even in the worst case, where parallelism may not be fully utilized, the overall time complexity remains O(n log n).

Performance Analysis:

Sequential Merge Sort:

- **Advantages**:
  - Simplicity: Easier to implement and understand.
  - Predictability: Performance doesn't depend on system architecture or the number of available threads.
- **Disadvantages**:
  - Limited Scalability: As it is sequential, it does not make full use of multi-core processors. The execution time increases linearly with the input size.

Multithreaded Merge Sort:

- **Advantages**:
  - Improved Performance on Multi-core Processors: Can utilize multiple cores to perform parallel sorting, potentially leading to faster execution times for large datasets.
  - Better Scalability: Scales better with increasing input sizes.
- **Disadvantages**:
  - Increased Complexity: The multithreaded version requires managing thread synchronization, which can introduce complexity and potential for bugs.
  - Potentially Suboptimal for Small Inputs: The overhead of thread creation and management can outweigh the benefits for small datasets.