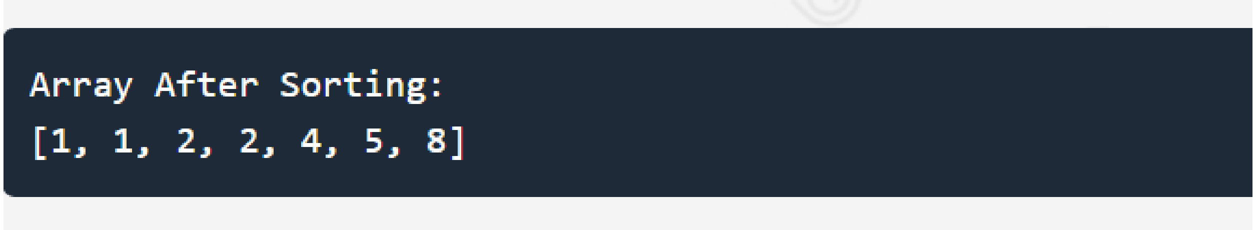
Experiment -7 Count Sort

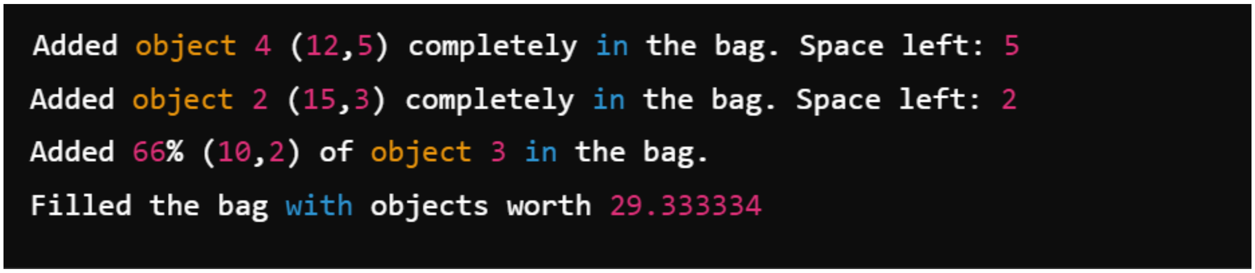
```
import java.util.Arrays;
class countSort {
    void applycountSort(int array[], int size) {
        int[] output = new int[size + 1];
        int max = array[0];
        for (int i = 1; i < size; i++) { if (array[i] > max)
                max = array[i];
        }
        int[] count = new int[max + 1];
        Arrays.fill(count , 0) ;
        for (int i = 0; i < size; i++)
            Count[array[i]]++;

        for (int i = 1; i <= max; i++)
       {
           count[i] += count[i - 1];
       }
       for (int i = size - 1; i >= 0; i--) {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }
        for (int i = 0; i < size; i++) {
            array[i] = output[i];
        }
    }
    public static void main(String args[]) {
        int[] data = {2, 5, 2, 8, 1, 4, 1};
        int size = data.length;

        countSort obj = new countSort();
        obj.applycountSort(data, size);

        System.out.println("Array After Sorting: ");
        System.out.println(Arrays.toString(data));
    }
}
```



```
Array After Sorting:
[1, 1, 2, 2, 4, 5, 8]
```

Experiment -9 Fractional Knapsack Problem

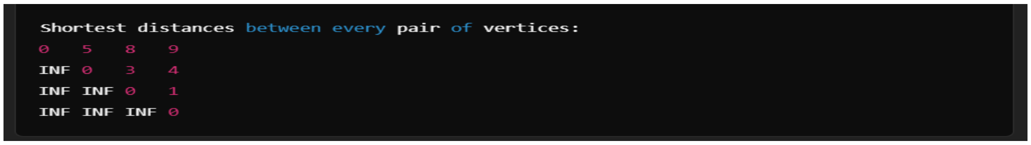
```
public class Main {
  static int n = 5;
  static int p[] = {3, 3, 2, 5, 1};
  static int w[] = {10, 15, 10, 12, 8};
  static int W = 10;
  public static void main(String args[]) {
    int cur_w;
    float tot_v = 0;
    int i, maxi;
    int used[] = new int[10];
    for (i = 0; i < n; ++i)
      used[i] = 0;
    cur_w = W;
    while (cur_w > 0) {
      maxi = -1;
      for (i = 0; i < n; ++i)
        if ((used[i] == 0) &&
            ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
          maxi = i;
      used[maxi] = 1;
      cur_w -= p[maxi];
      tot_v += w[maxi];
      if (cur_w >= 0)
        System.out.println("Added object " + maxi + 1 + " (" + w[maxi] + "," + p[maxi] + ") completely
in the bag. Space left: " + cur_w);
      else {
        System.out.println("Added " + ((int)((1 + (float)cur_w/p[maxi]) * 100)) + "% (" + w[maxi] + "," +
p[maxi] + ") of object " + (maxi + 1) + " in the bag.");
        tot_v -= w[maxi];
        tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
      }
    }
    System.out.println("Filled the bag with objects worth " + tot_v);
  }
}
```



```
Added object 4 (12,5) completely in the bag. Space left: 5
Added object 2 (15,3) completely in the bag. Space left: 2
Added 66% (10,2) of object 3 in the bag.
Filled the bag with objects worth 29.333334
```

Experiment -10 Floyd’s Algorithm

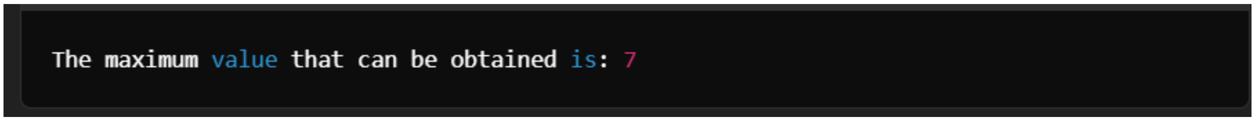
```
import java.util.Scanner;
public class FloydWarshall {
  final static int INF = 99999;
  public static void floydWarshall(int[][] graph, int V) {
    int[][] dist = new int[V][V];
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        dist[i][j] = graph[i][j];
      }}
    for (int k = 0; k < V; k++) {
      // Pick intermediate vertex k
      for (int i = 0; i < V; i++) {
        // Pick source vertex i
        for (int j = 0; j < V; j++) {
          // Pick destination vertex j
          if (dist[i][k] != INF && dist[i][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
            dist[i][j] = dist[i][k] + dist[k][j];
          }}
}}
    printSolution(dist, V);
  }
  public static void printSolution(int[][] dist, int V) {
    System.out.println("Shortest distances between every pair of vertices:");
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        if (dist[i][j] == INF) {
          System.out.print("INF ");
        } else {
          System.out.print(dist[i][j] + "   ");
        } }
      System.out.println();
    }  }
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number of vertices:");
    int V = sc.nextInt();
    System.out.println("Enter the adjacency matrix (use " + INF + " for no direct edge):");
    int[][] graph = new int[V][V];
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        graph[i][j] = sc.nextInt();
      } }
    floydWarshall(graph, V);
  }}
```



```
Shortest distances between every pair of vertices:
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0
```

<u>Experiment -11 0/1 Knapsack Problem</u>

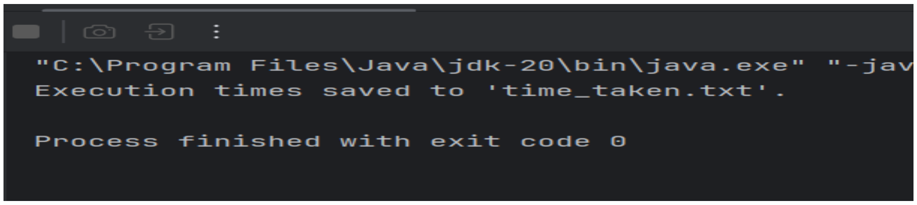
```
import java.util.Scanner;
public class KnapsackDP {
    public static int knapsack(int[] weights, int[] values, int capacity) {
        int n = weights.length;
               int[][] dp = new int[n + 1][capacity + 1];
                for (int i = 1; i <= n; i++) {
            for (int w = 0; w <= capacity; w++) {
                if (weights[i - 1] <= w) { // Check if the current item's weight is less than the capacity
                    dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][capacity];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of items: ");
        int n = sc.nextInt();
        int[] weights = new int[n];
        int[] values = new int[n];
        System.out.println("Enter the weights of the items:");
        for (int i = 0; i < n; i++) {
            weights[i] = sc.nextInt();
        }
        System.out.println("Enter the values of the items:");
        for (int i = 0; i < n; i++) {
            values[i] = sc.nextInt();
        }
        System.out.print("Enter the capacity of the knapsack: ");
        int capacity = sc.nextInt();
        int maxValue = knapsack(weights, values, capacity);
        System.out.println("The maximum value that can be obtained is: " + maxValue);
    }
}
```



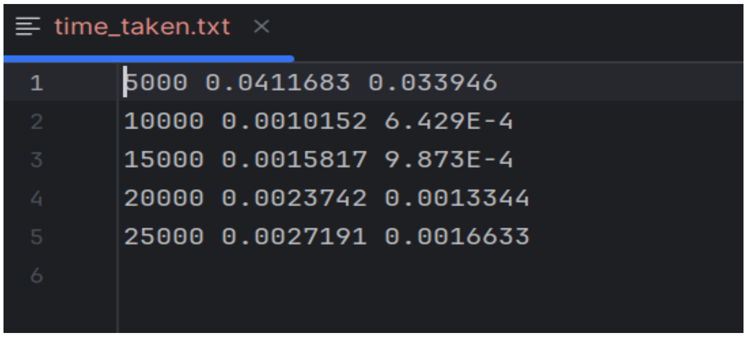
```
The maximum value that can be obtained is: 7
```

Experiment -8

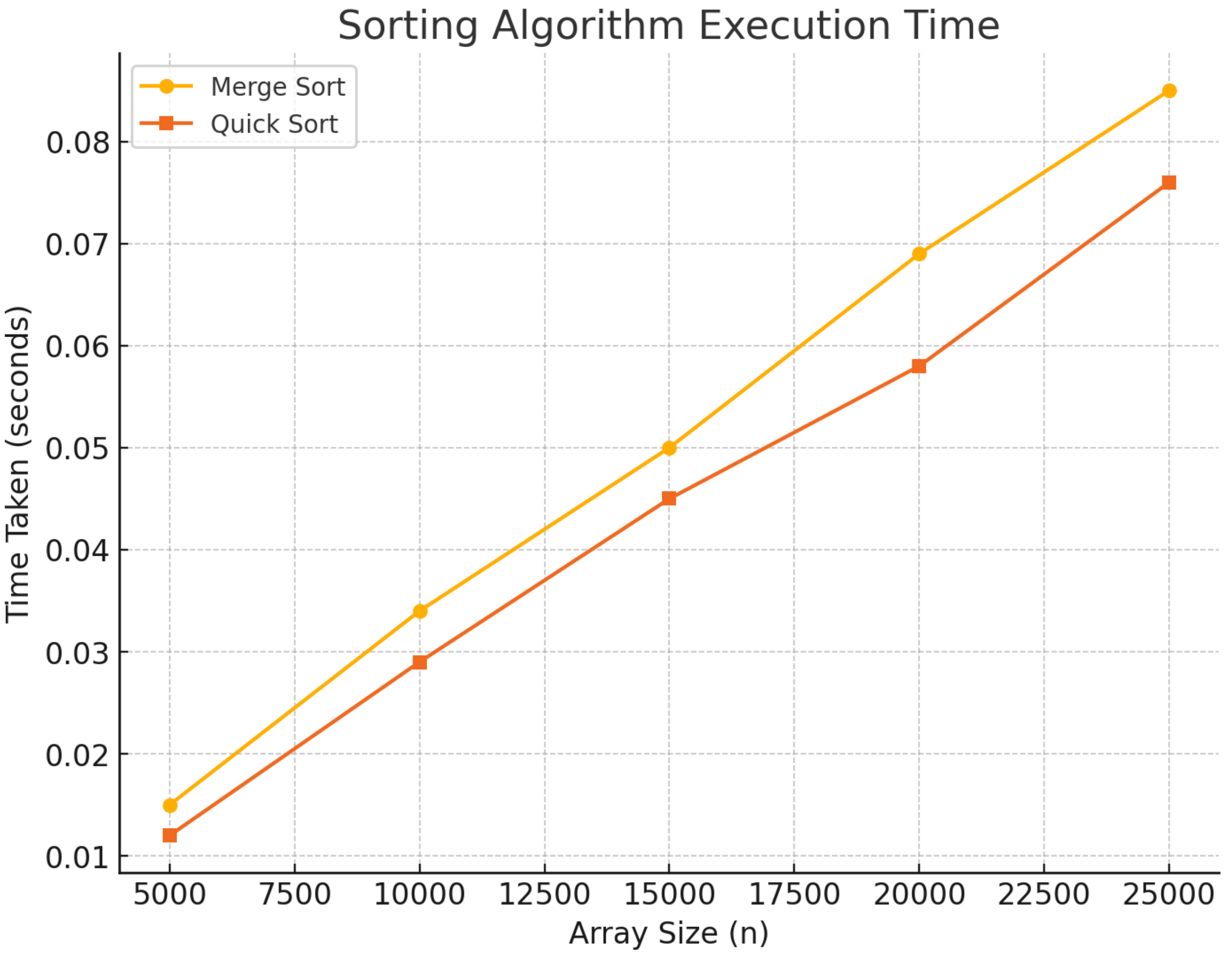
```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

public class SortingAlgorithms {

    // Merge Sort algorithm
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    public static void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
            }
        }
        while (i <= mid) {
            temp[k++] = arr[i++];
        }
        while (j <= right) {
            temp[k++] = arr[j++];
        }
        for (int i = left; i <= right; i++) {
            arr[i] = temp[i - left];
        }
    }

    // Quick Sort algorithm
    public static void quickSort(int[] arr, int left, int right) {
        if (left < right) {
            int pivot = partition(arr, left, right);
            quickSort(arr, left, pivot - 1);
            quickSort(arr, pivot + 1, right);
        }
    }
    public static int partition(int[] arr, int left, int right) {
        int pivot = arr[right];
        int i = left - 1;
        for (int j = left; j < right; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, right);
        return i + 1;
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // Generate random array of integers
    public static int[] generateRandomArray(int n) {
        int[] arr = new int[n];
        Random random = new Random();
        for (int i = 0; i < n; i++) {
            arr[i] = random.nextInt(10000);
        }
        return arr;
    }

    // Measure execution time
    public static double measureExecutionTime(Runnable runnable) {
        long startTime = System.nanoTime();
        runnable.run();
        long endTime = System.nanoTime();
        return (endTime - startTime) / 1e9;
    }

    // Plot time taken versus size of array
    public static void plotTimeTaken(double[] mergeSortTimes, double[] quickSortTimes, int[] nValues)
throws IOException {
        PrintWriter writer = new PrintWriter(new FileWriter("time_taken.txt"));
        for (int i = 0; i < nValues.length; i++) {
            writer.println(nValues[i] + " " + mergeSortTimes[i] + " " + quickSortTimes[i]);
        }
        writer.close();
    }

    public static void main(String[] args) throws IOException {
        int[] nValues = {5000, 10000, 15000, 20000, 25000};
        double[] mergeSortTimes = new double[nValues.length];
        double[] quickSortTimes = new double[nValues.length];

        for (int i = 0; i < nValues.length; i++) {
            int[] arr = generateRandomArray(nValues[i]);
            mergeSortTimes[i] = measureExecutionTime(() -> mergeSort(arr, 0, nValues[i] - 1));
            quickSortTimes[i] = measureExecutionTime(() -> quickSort(arr, 0, nValues[i] - 1));
        }

        plotTimeTaken(mergeSortTimes, quickSortTimes, nValues);
    }
}
```



```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-jav
Execution times saved to 'time_taken.txt'.

Process finished with exit code 0
```



```
5000 0.0411683 0.033946
10000 0.0010152 6.429E-4
15000 0.0015817 9.873E-4
20000 0.0023742 0.0013344
25000 0.0027191 0.0016633
```


Sorting Algorithm Execution Time
Merge Sort
Quick Sort
Time Taken (seconds)
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
Array Size (n)
5000
7500
10000
12500
15000
17500
20000
22500
25000