1. MergeSort

```
import java.util.Random;


class Order
{
   long timeStatmp;
   String orderId;
   Order(long timeStatmp, String orderId)
   {
      this.timeStatmp = timeStatmp;
      this.orderId = orderId;
   }
}
public class MergeSort
{
   public static void mergeSort(Order[] orders, int left, int right)
   {
      if(left<right)
      {
         int mid = (left+right)/2;
         mergeSort(orders,left,mid);
         mergeSort(orders,mid+1,right);
         merge(orders,left,mid,right);
      }
   }
   public static void merge(Order[] orders,int left,int mid,int right)
```

```
{
    int n1 = mid-left+1;
    int n2 = right-mid;

    Order[] leftArray = new Order[n1];
    Order[] rightArray = new Order[n2];

    for(int i=0;i<n1;i++)
    {
        leftArray[i] = orders[left+i];
    }
    for (int j=0;j<n2;j++)
    {
        rightArray[j] = orders[mid+1+j];
    }

    int i=0,j=0,k=left;
    while(i<n1 && j<n2)
    {
        if(leftArray[i].timeStatmp<rightArray[j].timeStatmp)
        {
            orders[k] = leftArray[i];
            i++;
            k++;
        }
        else
```

```
        {
            orders[k] = rightArray[j];
            j++;
            k++;
        }
    }
    while(i<n1)
    {
        orders[k] = leftArray[i];
        i++;
        k++;
    }
    while(j<n2)
    {
        orders[k] = rightArray[j];
        j++;
        k++;
    }
}
public static void main(String[] args)
{
    int n= 10;
    Order[] orders =new  Order[n];
    Random random = new Random();
    for(int i=0;i<n;i++)
    {
```

```
        orders[i] = new Order(random.nextInt(1000),"order"+i);
      }
      System.out.println("before Sort");
      for(Order o:orders)
      {
        System.out.println(o.orderId+" - "+o.timeStatmp);
      }
      mergeSort(orders,0,orders.length-1);

      System.out.println("after Sort");
      for (Order o:orders)
      {
        System.out.println(o.orderId+" - "+o.timeStatmp);
      }
   }
}
```

2.QuickSort

```
class Movies
{
   String title;
   double imdbRating;
   int releaseYear;
   int watchTimePopularity;

   Movies(String title, double imdbRating, int releaseYear, int watchTimePopularity)
   {
      this.title = title;
      this.imdbRating = imdbRating;
```

```
        this.releaseYear = releaseYear;
        this.watchTimePopularity = watchTimePopularity;
    }
}
public class QuickSort
{
    public static void quickSort(Movies[] movies, int low, int high, String parameter)
    {
        if (low < high)
        {
            int pivotIndex = partition(movies, low, high, parameter);
            quickSort(movies, low, pivotIndex - 1, parameter);
            quickSort(movies, pivotIndex, high, parameter);
        }
    }

    public static int partition(Movies[] movies, int low, int high, String parameter)
    {
        Movies pivot = movies[(low + high) / 2];
        while (low <= high)
        {
            while (compare(movies[low], pivot, parameter) > 0) low++;
            while (compare(movies[high], pivot, parameter) < 0) high--;

            if (low <= high)
            {
                swap(movies, low, high);
                low++;
                high--;
            }
        }
        return low;
    }

    private static int compare(Movies m1, Movies m2, String parameter)
    {
        switch (parameter)
        {
```

```
            case "rating":
                return Double.compare(m1.imdbRating, m2.imdbRating);
            case "year":
                return Integer.compare(m1.releaseYear, m2.releaseYear);
            case "popularity":
                return Integer.compare(m1.watchTimePopularity, m2.watchTimePopularity);
            default:
                return 0;
        }
    }

    private static void swap(Movies[] movies, int i, int j)
    {
        Movies temp = movies[i];
        movies[i] = movies[j];
        movies[j] = temp;
    }

    public static void main(String[] args)
    {
        Movies[] movies = {
                new Movies("Inception", 8.8, 2010, 5000000),
                new Movies("Avengers: Endgame", 8.4, 2019, 9000000),
                new Movies("The Dark Knight", 9.0, 2008, 8000000),
                new Movies("Interstellar", 8.6, 2014, 7000000),
                new Movies("Oppenheimer", 8.7, 2023, 6000000)
        };

        String parameter = "rating";
        quickSort(movies, 0, movies.length - 1, parameter);

        System.out.println("Sorted by " + parameter.toUpperCase() + " (Descending):");
        for (Movies m : movies) {
            System.out.println(m.title + " | " + m.imdbRating + " | " + m.releaseYear + " | " +
m.watchTimePopularity);
        }
    }
}
```

3.FractionalKnapsack

```
import java.util.*;

class Item
{
   String name;
   double weight;
   double value;
   boolean divisible;

   Item(String name, double weight, double value, boolean divisible)
   {
      this.name=name;
      this.weight=weight;
      this.value=value;
      this.divisible=divisible;
   }
}
public class FractionalKnapsack
{
   public static double maximizeUtility(List<Item> items, double capacity)
   {
      items.sort((a,b)->Double.compare(b.value/b.weight, a.value/a.weight));

      double totalValue=0, weight=0;
      for(Item i:items)
      {
         if(weight+i.weight<=capacity)
         {
            weight+=i.weight;
            totalValue+=i.value;
            System.out.println("Taking full: " + i.name);
         }
         else if(i.divisible)
         {
            double remain=capacity-weight;
            totalValue+=i.value*(remain/i.weight);
```

```
            System.out.println("Taking " + remain + "kg of: " + i.name);
            break;
          }
      }
      System.out.printf("Total Utility: %.2f (%.2f kg used of %.2f)\n", totalValue, weight,
capacity);
      return totalValue;
   }
   public static void main(String[] args)
   {
      List<Item> items = Arrays.asList(
            new Item("Medicine",10,120,false),
            new Item("Food",20,100,true),
            new Item("Water",30,90,true),
            new Item("Blankets",15,60,false),
            new Item("Clothes",25,75,false)
      );

      System.out.println("Emergency Relief Optimization");
      double capacity=50;
      double result = maximizeUtility(items,capacity);
      System.out.println(" ✅ Maximum Utility Value: " + result);
   }
}
```

4.dijekstra

```
import java.util.*;

class Edge {
   int to;       // destination node
   int weight;   // travel time
   Edge(int to, int weight) {
      this.to = to;
      this.weight = weight;
   }
}

public class DijekstraAlgo {
```

```
    // Dijkstra's Algorithm
    public static void dijkstra(List<List<Edge>> graph, int source, int n) {
        int[] dist = new int[n];            // shortest distance from source
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[source] = 0;

        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        pq.add(new int[]{source, 0});

        while (!pq.isEmpty()) {
            int[] node = pq.poll();
            int u = node[0];
            int d = node[1];

            if (d > dist[u]) continue; // skip outdated entry

            for (Edge e : graph.get(u)) {
                int v = e.to;
                int newDist = d + e.weight;
                if (newDist < dist[v]) {
                    dist[v] = newDist;
                    pq.add(new int[]{v, newDist});
                }
            }
        }

        // Print results
        System.out.println("Shortest travel time from Source " + source + " to all
intersections:");
        for (int i = 0; i < n; i++) {
            System.out.println("To node " + i + " → " + (dist[i] == Integer.MAX_VALUE ? "Not
reachable" : dist[i] + " mins"));
        }
    }

    public static void main(String[] args) {
        int n = 6; // number of intersections (0 to 5)
```

```
        List<List<Edge>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) graph.add(new ArrayList<>());

        // Add roads (edges) between intersections
        graph.get(0).add(new Edge(1, 4));
        graph.get(0).add(new Edge(2, 2));
        graph.get(1).add(new Edge(2, 1));
        graph.get(1).add(new Edge(3, 5));
        graph.get(2).add(new Edge(3, 8));
        graph.get(2).add(new Edge(4, 10));
        graph.get(3).add(new Edge(5, 2));  // hospital
        graph.get(4).add(new Edge(5, 3));  // hospital

        int source = 0; // ambulance location

        dijkstra(graph, source, n);
    }
}
```

5.MultiStage

```
import java.util.*;

public class SwiftCargo {

    static class Edge {

        int from, to;

        double cost;

        Edge(int from, int to, double cost) {

            this.from = from;

            this.to = to;

            this.cost = cost;

        }

    }
```

```
static class MultistageGraph {
    int stages;
    List<List<Integer>> stageNodes;
    Map<Integer, List<Edge>> adj;
    MultistageGraph(int stages) {
        this.stages = stages;
        stageNodes = new ArrayList<>();
        adj = new HashMap<>();
        for (int i = 0; i < stages; i++) stageNodes.add(new ArrayList<>());
    }
    void addNode(int stage, int node) {
        stageNodes.get(stage).add(node);
        adj.putIfAbsent(node, new ArrayList<>());
    }
    void addEdge(int from, int to, double cost) {
        adj.putIfAbsent(from, new ArrayList<>());
        adj.get(from).add(new Edge(from, to, cost));
    }
    boolean updateEdge(int from, int to, double newCost) {
        List<Edge> edges = adj.get(from);
        if (edges == null) return false;
        boolean updated = false;
        for (Edge e : edges) {
            if (e.to == to) {
                e.cost = newCost;
                updated = true;
```

```
            }
        }
        return updated;
    }
}
static class Result {
    double cost;
    List<Integer> path;
    Result(double cost, List<Integer> path) {
        this.cost = cost;
        this.path = path;
    }
}
static Result findOptimalPath(MultistageGraph g, int source, int destination) {
    Map<Integer, Double> dp = new HashMap<>();
    Map<Integer, Integer> parent = new HashMap<>();
    dp.put(source, 0.0);
    for (int stage = 0; stage < g.stages; stage++) {
        Map<Integer, Double> nextDp = new HashMap<>();
        for (int u : g.stageNodes.get(stage)) {
            if (!dp.containsKey(u)) continue;
            double costU = dp.get(u);
            List<Edge> edges = g.adj.get(u);
            if (edges == null) continue;
            for (Edge e : edges) {
```

```
            int nextStage = -1;
            for (int s = stage + 1; s < g.stages; s++) {
                if (g.stageNodes.get(s).contains(e.to)) {
                    nextStage = s;
                    break;
                }
            }
            if (nextStage == -1) continue;
            double newCost = costU + e.cost;
            if (!nextDp.containsKey(e.to) || newCost < nextDp.get(e.to)) {
                nextDp.put(e.to, newCost);
                parent.put(e.to, u);
            }
        }
    }
    dp.putAll(nextDp);
}
List<Integer> path = new ArrayList<>();
if (!dp.containsKey(destination)) return new Result(Double.POSITIVE_INFINITY, path);
int cur = destination;
while (cur != source) {
    path.add(cur);
    cur = parent.get(cur);
}
path.add(source);
Collections.reverse(path);
```

```
    return new Result(dp.get(destination), path);
}
static List<Result> batchOptimalPaths(MultistageGraph g, List<int[]> requests) {
    List<Result> results = new ArrayList<>();
    for (int[] req : requests) {
        Result res = findOptimalPath(g, req[0], req[1]);
        results.add(res);
    }
    return results;
}
public static void main(String[] args) {
    MultistageGraph g = new MultistageGraph(3);
    g.addNode(0, 0);
    g.addNode(0, 1);

    g.addNode(1, 2);
    g.addNode(1, 3);

    g.addNode(2, 4);
    g.addNode(2, 5);

    g.addEdge(0, 2, 4.0);
    g.addEdge(0, 3, 2.0);
    g.addEdge(1, 2, 3.0);
    g.addEdge(1, 3, 2.5);
    g.addEdge(2, 4, 5.0);
```

```
        g.addEdge(2, 5, 6.0);
        g.addEdge(3, 4, 2.0);
        g.addEdge(3, 5, 3.5);

        Result r1 = findOptimalPath(g, 0, 4);
        System.out.println("optimal cost: " + r1.cost);
        System.out.println("optimal path: " + r1.path);

        g.updateEdge(3, 4, 10.0);

        Result r2 = findOptimalPath(g, 0, 4);
        System.out.println("after traffic update the Optimal cost: " + r2.cost);
        System.out.println("after traffic update the Optimal path: " + r2.path);

        List<int[]> requests = new ArrayList<>();
        requests.add(new int[]{0, 4});
        requests.add(new int[]{1, 5});
        List<Result> batch = batchOptimalPaths(g, requests);
        for (int i = 0; i < batch.size(); i++) {
            System.out.println("Request " + (i + 1) + " - cost: " + batch.get(i).cost + ", path: " +
batch.get(i).path);
        }
    }
}
```

6. 0/1 Knapsack

```
public class DisasterReliefEasy {


   public static void main(String[] args) {
      // Items: Food, Medicine, Blanket
      int[] weight = {10, 20, 30};   // weight in kg
      int[] value = {60, 100, 120};  // utility value
      int capacity = 50;            // truck can carry 50 kg


      int n = weight.length;
      int[][] dp = new int[n + 1][capacity + 1];


      // Build table dp[n][capacity]
      for (int i = 1; i <= n; i++) {
         for (int w = 1; w <= capacity; w++) {
            if (weight[i - 1] <= w) {
               dp[i][w] = Math.max(value[i - 1] + dp[i - 1][w - weight[i - 1]], dp[i - 1][w]);
            } else {
               dp[i][w] = dp[i - 1][w];
            }
         }
      }


      System.out.println("🚚 Maximum Utility Value: " + dp[n][capacity]);


      // Find which items are selected
```

```
        System.out.println(" ✅ Items Selected:");
        int w = capacity;
        for (int i = n; i > 0 && dp[i][w] > 0; i--) {
            if (dp[i][w] != dp[i - 1][w]) {
                System.out.println("Item " + i + " (Weight: " + weight[i - 1] + ", Value: " + value[i - 1]
+ ")");
                w -= weight[i - 1];
            }
        }
    }
}
```

7.Greedy Coloring

```
import java.util.*;
public class UniversityTimetable {
    static class Graph {
        int V;
        List<Integer>[] adj;
        @SuppressWarnings("unchecked")
        Graph(int V) {
            this.V = V;
            adj = new ArrayList[V];
            for (int i = 0; i < V; i++) adj[i] = new ArrayList<>();
        }
        void addEdge(int u, int v) {
            adj[u].add(v);
```

```
    adj[v].add(u);
}
int[] greedyColoring(int[] roomCapacities, int[] courseSizes) {
    int[] result = new int[V];
    Arrays.fill(result, -1);
    boolean[] available = new boolean[V];
    Arrays.fill(available, true);

    result[0] = 0;

    for (int u = 1; u < V; u++) {
        Arrays.fill(available, true);

        for (int v : adj[u]) {
            if (result[v] != -1) {
                available[result[v]] = false;
            }
        }

        int c;
        for (c = 0; c < V; c++) {
            if (available[c]) break;
        }
        result[u] = c;
    }
```

```
        if (roomCapacities != null && courseSizes != null) {
            Map<Integer, Integer> slotLoad = new HashMap<>();
            for (int i = 0; i < V; i++) {
                slotLoad.put(result[i], slotLoad.getOrDefault(result[i], 0) + courseSizes[i]);
            }
            for (int slot : slotLoad.keySet()) {
                if (slotLoad.get(slot) > roomCapacities[slot]) {
                    System.out.println("Warning: Slot " + slot + " exceeds room capacity.");
                }
            }
        }
        return result;
    }
    void printTimetable(int[] colors) {
        System.out.println("Course : Exam Slot");
        for (int i = 0; i < V; i++) {
            System.out.println("Course " + i + " -> Slot " + colors[i]);
        }
        int maxSlot = Arrays.stream(colors).max().orElse(0) + 1;
        System.out.println("Total slots used: " + maxSlot);
    }
}
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter number of courses ");
    int n = sc.nextInt();
```

```
        Graph g = new Graph(n);
        System.out.print("Enter number of conflicts ");
        int m = sc.nextInt();
        System.out.println("Enter conflict pairs ");
        for (int i = 0; i < m; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            g.addEdge(u, v);
        }
        System.out.print("Do you want to input room capacities?Enter (y/n) ");
        sc.nextLine();
        String ans = sc.nextLine();
        int[] roomCapacities = null;
        int[] courseSizes = null;
        if (ans.equalsIgnoreCase("y")) {
            roomCapacities = new int[n];
            courseSizes = new int[n];
            System.out.println("Enter course size (no. of students)");
            for (int i = 0; i < n; i++) courseSizes[i] = sc.nextInt();
            System.out.println("Enter room capacities per slot (max slots = number of courses)");
            for (int i = 0; i < n; i++) roomCapacities[i] = sc.nextInt();
        }
        int[] colors = g.greedyColoring(roomCapacities, courseSizes);
        g.printTimetable(colors);
    }
}
```

8.BranchBound

```
import java.util.*;
public class SwiftShipTSP {
   static class Node implements Comparable<Node> {
      int level;
      int pathCost;
      int bound;
      List<Integer> path;
      Node(int level, int pathCost, int bound, List<Integer> path) {
         this.level = level;
         this.pathCost = pathCost;
         this.bound = bound;
         this.path = new ArrayList<>(path);
      }
      @Override
      public int compareTo(Node other) {
         return Integer.compare(this.bound, other.bound);
      }
   }
   static int calculateBound(Node node, int[][] costMatrix, int N) {
      int bound = node.pathCost;
      boolean[] visited = new boolean[N];
      for (int city : node.path) visited[city] = true;
      for (int i = 0; i < N; i++) {
         if (!visited[i]) {
```

```
            int minEdge = Integer.MAX_VALUE;
            for (int j = 0; j < N; j++) {
                if (i != j && !visited[j]) {
                    minEdge = Math.min(minEdge, costMatrix[i][j]);
                }
            }
            bound += (minEdge == Integer.MAX_VALUE) ? 0 : minEdge;
        }
    }
    return bound;
}
static List<Integer> tspBranchAndBound(int[][] costMatrix) {
    int N = costMatrix.length;
    PriorityQueue<Node> pq = new PriorityQueue<>();
    List<Integer> bestPath = new ArrayList<>();
    int minCost = Integer.MAX_VALUE;
    Node root = new Node(0, 0, 0, new ArrayList<>(List.of(0)));
    root.bound = calculateBound(root, costMatrix, N);
    pq.add(root);
    while (!pq.isEmpty()) {
        Node node = pq.poll();
        if (node.bound >= minCost) continue;
        if (node.level == N - 1) {

            int last = node.path.get(node.path.size() - 1);
            if (costMatrix[last][0] > 0) {
```

```
            int totalCost = node.pathCost + costMatrix[last][0];
            if (totalCost < minCost) {
                minCost = totalCost;
                bestPath = new ArrayList<>(node.path);
                bestPath.add(0);
            }
        }
        continue;
    }

    int lastCity = node.path.get(node.path.size() - 1);
    for (int nextCity = 0; nextCity < N; nextCity++) {
        if (!node.path.contains(nextCity) && costMatrix[lastCity][nextCity] > 0) {
            List<Integer> newPath = new ArrayList<>(node.path);
            newPath.add(nextCity);
            int newCost = node.pathCost + costMatrix[lastCity][nextCity];
            Node child = new Node(node.level + 1, newCost, 0, newPath);
            child.bound = calculateBound(child, costMatrix, N);
            if (child.bound < minCost) pq.add(child);
        }
    }
  }
  System.out.println("Minimum cost: " + minCost);
  return bestPath;
}
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of cities: ");
        int N = sc.nextInt();
        int[][] costMatrix = new int[N][N];
        System.out.println("Enter cost matrix (NxN):");
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                costMatrix[i][j] = sc.nextInt();
        List<Integer> optimalRoute = tspBranchAndBound(costMatrix);
        System.out.println("Optimal route:");
        for (int city : optimalRoute) {
            System.out.print(city + " -> ");
        }
        System.out.println("End");
    }
}
```