BFS

```
import java.util.*;
public class BFS {
   public static void main(String[] args) {
      int vertices = 5;
      Graph graph = new Graph(vertices);
      graph.addEdge('A', 'B');
      graph.addEdge('A', 'C');
      graph.addEdge('A', 'D');
      graph.addEdge('C', 'E');
      graph.addEdge('C', 'B');
      char startVertex = 'A'; // Starting vertex
      System.out.println("BFS Traversal:");
      bfs(graph, startVertex);
   }
   static void bfs(Graph G, char s) {
      int[] status = new int[G.vertices];
      int[] d = new int[G.vertices];
      int[] PI = new int[G.vertices];
      for (int x = 0; x < G.vertices; x++) {
         status[x] = -1;
         d[x] = Integer.MAX_VALUE;
         PI[x] = -1;
      }
      int startIdx = s - 'A';
      status[startIdx] = 0;
      d[startIdx] = 0;
      PI[startIdx] = -1;
      Queue<Integer> queue = new LinkedList<>();
      queue.add(startIdx);
      while (!queue.isEmpty()) {
         int u = queue.poll();
         System.out.print((char) ('A' + u) + " ");
         for (int v : G.adjList.get(u)) {
            if (status[v] == -1) {
               status[v] = 0;
               d[v] = d[u] + 1;
               PI[v] = u;
               queue.add(v);
            }
         }
         status[u] = 1;
      }
```

```
    }

    static class Graph {
        int vertices;
        List<List<Integer>> adjList;
        public Graph(int vertices) {
            this.vertices = vertices;
            adjList = new ArrayList<>(vertices);
            for (int i = 0; i < vertices; i++) {
                adjList.add(new LinkedList<>());
            }
        }
        void addEdge(char u, char v) {
            adjList.get(u - 'A').add(v - 'A');
            adjList.get(v - 'A').add(u - 'A'); // Assuming undirected graph
        }
    }
}
```

KRUSKAL MST

```
// Java program for Kruskal's algorithm
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
public class KruskalMST {
    // Defines edge structure
    static class Edge {
        int src, dest, weight;
        public Edge(int src, int dest, int weight)
        {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }
    // Defines subset element structure
    static class Subset {
        int parent, rank;
        public Subset(int parent, int rank)
        {
            this.parent = parent;
            this.rank = rank;
```

```
    }
  }
  // Starting point of program execution
  public static void main(String[] args)
  {
    int V = 4;
    List<Edge> graphEdges = new ArrayList<Edge>(
      List.of(new Edge(0, 1, 10), new Edge(0, 2, 6),
          new Edge(0, 3, 5), new Edge(1, 3, 15),
          new Edge(2, 3, 4)));
    // Sort the edges in non-decreasing order
    // (increasing with repetition allowed)
    graphEdges.sort(new Comparator<Edge>() {
      @Override public int compare(Edge o1, Edge o2)
      {
        return o1.weight - o2.weight;
      }
    });
    kruskals(V, graphEdges);
  }
  // Function to find the MST
  private static void kruskals(int V, List<Edge> edges)
  {
    int j = 0;
    int noOfEdges = 0;
    // Allocate memory for creating V subsets
    Subset subsets[] = new Subset[V];
    // Allocate memory for results
    Edge results[] = new Edge[V];
    // Create V subsets with single elements
    for (int i = 0; i < V; i++) {
      subsets[i] = new Subset(i, 0);
    }
    // Number of edges to be taken is equal to V-1
    while (noOfEdges < V - 1) {
      // Pick the smallest edge. And increment the index for next iteration
      Edge nextEdge = edges.get(j);
      int x = findRoot(subsets, nextEdge.src);
      int y = findRoot(subsets, nextEdge.dest);
      // If including this edge doesn't cause cycle, include it in result and increment the index
of result for next edge
      if (x != y) {
        results[noOfEdges] = nextEdge;
        union(subsets, x, y);
```

```
            noOfEdges++;
          }
          j++;
        }
        // Print the contents of result[] to display the  built MST
        System.out.println( "Following are the edges of the constructed MST:");
        int minCost = 0;
        for (int i = 0; i < noOfEdges; i++) {
          System.out.println(results[i].src + " -- "+ results[i].dest + " == "+ results[i].weight);
          minCost += results[i].weight;

        }
        System.out.println("Total cost of MST: " + minCost);
    }
    // Function to unite two disjoint sets
    private static void union(Subset[] subsets, int x, int y)
    {
        int rootX = findRoot(subsets, x);
        int rootY = findRoot(subsets, y);
        if (subsets[rootY].rank < subsets[rootX].rank) {
          subsets[rootY].parent = rootX;
        }
        else if (subsets[rootX].rank < subsets[rootY].rank) {
          subsets[rootX].parent = rootY;
        }
        else {
          subsets[rootY].parent = rootX;
          subsets[rootX].rank++;
        }
    }
    // Function to find parent of a set
    private static int findRoot(Subset[] subsets, int i)
    {
        if (subsets[i].parent == i)
          return subsets[i].parent;
        subsets[i].parent = findRoot(subsets, subsets[i].parent);
        return subsets[i].parent;
    }
}
```

SEPARATE CHAINING HASH TABLE

```
import java.util.*;

public class SeparateChainingHashTable {
```

```
private LinkedList<Integer>[] table;
private int size;
public SeparateChainingHashTable(int size) {
    this.size = size;
    table = new LinkedList[size];
    for (int i = 0; i < size; i++) {
        table[i] = new LinkedList<>();
    }
}
private int hash(int key) {
    return key % size;
}
public void insert(int key) {
    int index = hash(key);
    table[index].add(key);
}
public boolean search(int key) {
    int index = hash(key);
    return table[index].contains(key);
}
public void display() {
    for (int i = 0; i < size; i++) {
        System.out.print(i + " -> ");
        for (Integer key : table[i]) {
            System.out.print(key + " ");
        }
        System.out.println();
```

```
        }
    }
    public static void main(String[] args) {
        int[] keys = {50, 700, 76, 85, 92, 73, 101};
        int tableSize = 7;
        SeparateChainingHashTable hashTable = new SeparateChainingHashTable(tableSize);
        for (int key : keys) {
            hashTable.insert(key);
        }
        System.out.println("Hash Table with Separate Chaining:");
        hashTable.display();
    }
}
```

LINEAR PROBING HASH TABLE

```
public class LinearProbingHashTable {
    private Integer[] table;
    private int size;
    private int capacity;
    public LinearProbingHashTable(int capacity) {
        this.capacity = capacity;
        size = 0;
        table = new Integer[capacity];
    }
    private int hash(int key) {
        return key % capacity;
    }
    public void insert(int key) {
```

```
    if (size == capacity) {
        System.out.println("Hash table is full");
        return;
    }
    int index = hash(key);
    while (table[index] != null) {
        index = (index + 1) % capacity;
    }
    table[index] = key;
    size++;
}
public boolean search(int key) {
    int index = hash(key);
    int originalIndex = index;
    while (table[index] != null) {
        if (table[index] == key) {
            return true;
        }
        index = (index + 1) % capacity;
        if (index == originalIndex) {
            return false; // Key not found after full loop through the table
        }
    }
    return false;
}
public void display() {
    for (int i = 0; i < capacity; i++) {
```

```
            if (table[i] != null) {
                System.out.println(i + " -> " + table[i]);
            }
        }
    }
    public static void main(String[] args) {
        int[] keys = {50, 700, 76, 85, 92, 73, 101};
        int tableSize = 7;
        LinearProbingHashTable hashTable = new LinearProbingHashTable(tableSize);
        for (int key : keys) {
            hashTable.insert(key);
        }
        System.out.println("Hash Table with Linear Probing:");
        hashTable.display();
    }
}
```

COUNTING INVERSIONS

```
public class CountInversions {
    public static long countInversions(int[] arr) {
        if (arr == null || arr.length <= 1)
            return 0;
        int[] temp = new int[arr.length];
        return mergeSortAndCount(arr, temp, 0, arr.length - 1);
    }
    private static long mergeSortAndCount(int[] arr, int[] temp, int left, int right) {
        if (left >= right)
            return 0;
```

```
    int mid = (left + right) / 2;
    long inversionCount = 0;
    inversionCount += mergeSortAndCount(arr, temp, left, mid);
    inversionCount += mergeSortAndCount(arr, temp, mid + 1, right);
    inversionCount += mergeAndCount(arr, temp, left, mid, right);
    return inversionCount;
}
private static long mergeAndCount(int[] arr, int[] temp, int left, int mid, int right) {
    System.arraycopy(arr, left, temp, left, right - left + 1);
    int i = left;
    int j = mid + 1;
    int k = left;
    long inversionCount = 0;
    while (i <= mid && j <= right) {
        if (temp[i] <= temp[j]) {
            arr[k++] = temp[i++];
        } else {
            arr[k++] = temp[j++];
            inversionCount += (mid - i + 1);
        }
    }
    while (i <= mid) {
        arr[k++] = temp[i++];
    }
    while (j <= right) {
        arr[k++] = temp[j++];
    }
```

```
        return inversionCount;
    }
    public static void main(String[] args) {
        int[] arr = { 1, 3, 2, 5, 6, 4 };
        long inversionCount = countInversions(arr);
        System.out.println("Number of inversions: " + inversionCount);
    }
}
```

ROBIN KARP

```
import java.util.*;
public class Rabin_Karp {
    public static int Search(String text, String pattern) {
        char[] txt = text.toCharArray();
        char[] pat = pattern.toCharArray();
        int n = txt.length;
        int m = pat.length;
        int i, j;
        int prime = 101;
        int power = 1;
        int txtHash = 0, patHash = 0;
        for (i = 0; i < m - 1; i++)
            power = (power << 1) % prime;
        for (i = 0; i < m; i++) {
            patHash = ((patHash << 1) + pat[i]) % prime;
            txtHash = ((txtHash << 1) + txt[i]) % prime;
        }
        for (i = 0; i <= n - m; i++) {
```

```
            if (txtHash == patHash) {
                for (j = 0; j < m; j++) {
                    if (txt[i + j] != pat[j])
                        break;
                }
                if (j == m)
                    return i;
            }
            if (i < n - m) {
                txtHash = (((txtHash - txt[i] * power) << 1) + txt[i + m]) % prime;
                if (txtHash < 0)
                    txtHash = (txtHash + prime);
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        int result = Search(text, pattern);
        if (result != -1)
            System.out.println("Pattern found at index " + result);
        else
            System.out.println("Pattern not found");
    }
}
```

BRUTE FORCE

```
import java.util.*;
public class BruteForceSearch {
   public static int search(String text, String pattern) {
      int n = text.length();
      int m = pattern.length();
      for (int i = 0; i <= n - m; i++) {
         int j;
         for (j = 0; j < m; j++) {
            if (text.charAt(i + j) != pattern.charAt(j))
               break;
         }
         if (j == m)
            return i; // Pattern found at index i
      }
      return -1; // Pattern not found
   }
   public static void main(String[] args) {
      String text = "ABABDABACDABABCABAB";
      String pattern = "ABABCABAB";
      int result = search(text, pattern);
      if (result != -1)
         System.out.println("Pattern found at index " + result);
      else
         System.out.println("Pattern not found");
   }
}
```

DFS

```
import java.util.*;

public class DFSRecursive {

    public static void dfs(Map<Character, List<Character>> graph, char start, Set<Character>
visited) {

        if (visited == null) {

            visited = new HashSet<>();

        }

        visited.add(start);

        System.out.print(start + " ");  // or perform any other action

        for (char neighbor : graph.get(start)) {

            if (!visited.contains(neighbor)) {

                dfs(graph, neighbor, visited);

            }

        }

    }

    public static void main(String[] args) {

        Map<Character, List<Character>> graph = new HashMap<>();

        graph.put('A', Arrays.asList('B', 'C'));

        graph.put('B', Arrays.asList('A', 'D', 'E'));

        graph.put('C', Arrays.asList('A', 'F'));

        graph.put('D', Arrays.asList('B'));

        graph.put('E', Arrays.asList('B', 'F'));

        graph.put('F', Arrays.asList('C', 'E'));

        Set<Character> visited = new HashSet<>();

        dfs(graph, 'A', visited);

    }

}
```

LCS

```
public class LongestCommonSubsequence {
    // Function to calculate the length of LCS
    static int[][] Length_LCS(String X, String Y) {
        int m = X.length();
        int n = Y.length();
        int[][] c = new int[m + 1][n + 1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    c[i][j] = 0;
                } else if (X.charAt(i - 1) == Y.charAt(j - 1)) {
                    c[i][j] = c[i - 1][j - 1] + 1;
                } else {
                    c[i][j] = Math.max(c[i - 1][j], c[i][j - 1]);
                }
            }
        }
        return c;
    }
    // Function to print the LCS
    static String Print_LCS(String X, String Y, int[][] c) {
        StringBuilder s = new StringBuilder();
        int i = X.length();
        int j = Y.length();
        while (i > 0 && j > 0) {
            if (X.charAt(i - 1) == Y.charAt(j - 1)) {
```

```
            s.insert(0, X.charAt(i - 1));
            i--;
            j--;
        } else if (c[i - 1][j] >= c[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
    return s.toString();
  }
  public static void main(String[] args) {
    String X = "ABCBDAB";
    String Y = "BDCABA";
    int[][] c = Length_LCS(X, Y);
    String lcs = Print_LCS(X, Y, c);
    System.out.println("Length of LCS: " + c[X.length()][Y.length()]);
    System.out.println("LCS: " + lcs);
  }
}
```

DIJKSTRA

```
import java.util.*;
class Graph {
  private int V; // Number of vertices
  private LinkedList<Edge>[] adj; // Adjacency list representation
  class Edge {
    int dest;
```

```
    int weight;
    Edge(int dest, int weight) {
        this.dest = dest;
        this.weight = weight;
    }
}
Graph(int V) {
    this.V = V;
    adj = new LinkedList[V];
    for (int i = 0; i < V; i++) {
        adj[i] = new LinkedList<>();
    }
}
void addEdge(int src, int dest, int weight) {
    adj[src].add(new Edge(dest, weight));
    adj[dest].add(new Edge(src, weight)); // If the graph is undirected
}
void dijkstra(int src) {
    PriorityQueue<Node> pq = new PriorityQueue<>(V, new Node());
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    pq.add(new Node(src, 0));
    while (!pq.isEmpty()) {
        Node node = pq.poll();
        int u = node.vertex;
```

```
        for (Edge edge : adj[u]) {
            int v = edge.dest;
            int weight = edge.weight;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.add(new Node(v, dist[v]));
            }
        }
    }
    printSolution(dist);
}
void printSolution(int[] dist) {
    System.out.println("Vertex \t\t Distance from Source");
    for (int i = 0; i < V; i++) {
        System.out.println(i + " \t\t " + dist[i]);
    }
}
class Node implements Comparator<Node> {
    public int vertex;
    public int cost;
    public Node() {}
    public Node(int vertex, int cost) {
        this.vertex = vertex;
        this.cost = cost;
    }
    @Override
    public int compare(Node node1, Node node2) {
```

```
            return Integer.compare(node1.cost, node2.cost);
        }
    }
    public static void main(String[] args) {
        int V = 5;
        Graph graph = new Graph(V);
        graph.addEdge(0, 1, 9);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(0, 4, 3);
        graph.addEdge(2, 1, 2);
        graph.addEdge(2, 3, 4);
        graph.dijkstra(0);
    }
}
```

PRIMS

```
import java.util.*;
class Graph {
    private int V; // Number of vertices
    private LinkedList<Edge>[] adj; // Adjacency list representation
    class Edge implements Comparable<Edge> {
        int dest;
        int weight;
        Edge(int dest, int weight) {
            this.dest = dest;
            this.weight = weight;
        }
```

```
    @Override
    public int compareTo(Edge compareEdge) {
        return this.weight - compareEdge.weight;
    }
}
Graph(int V) {
    this.V = V;
    adj = new LinkedList[V];
    for (int i = 0; i < V; i++) {
        adj[i] = new LinkedList<>();
    }
}
void addEdge(int src, int dest, int weight) {
    adj[src].add(new Edge(dest, weight));
    adj[dest].add(new Edge(src, weight)); // If the graph is undirected
}
void primMST() {
    PriorityQueue<Edge> pq = new PriorityQueue<>();
    boolean[] inMST = new boolean[V];
    int[] key = new int[V];  // To store key values used to pick minimum weight edge in cut
    int[] parent = new int[V]; // To store constructed MST
    // Initialize all keys as INFINITE
    Arrays.fill(key, Integer.MAX_VALUE);
    // Start with the first vertex (vertex 0)
    key[0] = 0;
    parent[0] = -1; // First node is always the root of MST
    pq.add(new Edge(0, key[0]));
```

```
    while (!pq.isEmpty()) {
        // The vertex with the smallest key value
        Edge edge = pq.poll();
        int u = edge.dest;
        // Include vertex in MST
        inMST[u] = true;
        // Traverse through all adjacent vertices of u
        for (Edge neighbor : adj[u]) {
            int v = neighbor.dest;
            int weight = neighbor.weight;
            // If v is not in MST and weight of u-v is smaller than current key of v
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.add(new Edge(v, key[v]));
                parent[v] = u;
            }
        }
    }
    // Print the constructed MST
    printMST(parent);
}
void printMST(int[] parent) {
    System.out.println("Edge \tWeight");
    for (int i = 1; i < V; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + getWeight(parent[i], i));
    }
}
```

```
    int getWeight(int u, int v) {
        for (Edge edge : adj[u]) {
            if (edge.dest == v) {
                return edge.weight;
            }
        }
        return 0;
    }
    public static void main(String[] args) {
        int V = 5;
        Graph graph = new Graph(V);
        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 3, 6);
        graph.addEdge(1, 2, 3);
        graph.addEdge(1, 3, 8);
        graph.addEdge(1, 4, 5);
        graph.addEdge(2, 4, 7);
        graph.addEdge(3, 4, 9);
        graph.primMST();
    }
}
```

WEIGHTED INTERVAL SCHEDULING

```
import java.util.Arrays;
import java.util.Comparator;
class Interval {
    int start, end, weight;
    Interval(int start, int end, int weight) {
```

```
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
}
public class WeightedIntervalScheduling {
    // Find the latest non-conflicting interval
    private static int latestNonConflict(Interval[] intervals, int i) {
        for (int j = i - 1; j >= 0; j--) {
            if (intervals[j].end <= intervals[i].start) {
                return j;
            }
        }
        return -1;
    }
    // Function to find the maximum weight of non-overlapping intervals
    public static int findMaxWeight(Interval[] intervals) {
        Arrays.sort(intervals, Comparator.comparingInt(a -> a.end));
        int n = intervals.length;
        int[] dp = new int[n];
        dp[0] = intervals[0].weight;
        for (int i = 1; i < n; i++) {
            int inclProf = intervals[i].weight;
            int l = latestNonConflict(intervals, i);
            if (l != -1) {
                inclProf += dp[l];
            }
```

```
            dp[i] = Math.max(inclProf, dp[i - 1]);
        }
        return dp[n - 1];
    }
    public static void main(String[] args) {
        Interval[] intervals = {
                new Interval(1, 2, 50),
                new Interval(3, 5, 20),
                new Interval(6, 19, 100),
                new Interval(2, 100, 200)
        };
        System.out.println("The maximum weight of non-overlapping intervals is: " +
findMaxWeight(intervals));
    }
}
```

MATRIX CHAIN MULTIPLICATION

```
public class MatrixChainMultiplication {
    public static int matrixChainOrder(int[] p) {
        int n = p.length - 1;
        int[][] m = new int[n][n];
        for (int i = 1; i < n; i++) {
            m[i][i] = 0;
        }
        for (int L = 2; L <= n; L++) {
            for (int i = 0; i < n - L + 1; i++) {
                int j = i + L - 1;
                m[i][j] = Integer.MAX_VALUE;
```

```
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i] * p[k + 1] * p[j + 1];
                if (q < m[i][j]) {
                    m[i][j] = q;
                }
            }
        }
    }
    return m[0][n - 1];
  }
  public static void main(String[] args) {
    int[] p = {1, 2, 3, 4};
    System.out.println("Minimum number of multiplications is: " + matrixChainOrder(p));
  }
}
```

Greedy Algorithm

10) Fractional knapsack

```
package greedy-method;
import java.util.Arrays;
Public class Fractional-Knapsack {
    Preivate class Items implements Comparable<Items>{
        int p, w;
        double d;
        Items (int profit, int weight){
            p= profit;
            w= weight;
            d= (double) p/w;
        }
        public int compareTo(Items x){
            return (int) (x.d - this.d);
        }
    }
    public double maximizeProfit(int[] p, int[] w, int W){
        double maxProfit = 0;
        int n = w.length;
        Items[] itemlist = new Items[n];
        for (int i=0; i<n; i++)
            itemlist[i]= new Items(p[i], w[i]);

        public Arrays.sort(itemlist);
        for (int i=0; i<n; i++){
            if (W - itemlist[i].w >=0){
                W= W - itemlist[i].w;
                maxProfit = maxProfit + itemlist[i].p;
            }
            else {
                maxProfit = maxProfit + (itemlist[i].d * W);
                break;
            }
        }
        return maxProfit;
    }
    public static void main (String args[]){
        int[] p= {60, 40, 90, 120};
        int[] w= {10, 40, 20, 30};
        int W= 50;
        Fractional-Knapsack obj = new Fractional-Knapsack();
        System.out.println("Maximum Profit: " + obj.maximize
            Profit(p, w, W))
}
```

Quick Sort

```
import java.util.Arrays;
class QuickSort{
    public static void swap(int[]A, int i, int j){
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
    public static int PARTITION(int[]A, int p, int r){
        int x = A[r];
        int i = p-1;
        for (int j=p; j<r; j++){
            if (A[j]<=x){
                i++;
                swap(A, i, j);
            }
        }
        swap(A, i+1, r);
        return (i+1);
    }
    public static void QUICK_SORT(int[] A, int p, int r){
        if (p<r){
            int q = PARTITION(A,p,r);
            QUICK_SORT(A, p, q-1);
            QUICK_SORT(A, q+1, r);
        }
    }
    public static void main(String[] args){
        int[]A= {5, 2, 4, 6, 1, 3};
        int p=0, r=A.length-1;
        QUICK_SORT(A, p, r);
        System.out.println(Arrays.toString(A));
```

Merge Sort

```
import java.util.Arrays;
public class Merge_sort {
    public static void Merge (int[]A, int p, int q, int r) {
        int n1 = q-p+1;
        int n2 = r-q;
        int[] L = new int[n1+1];
        int[] R = new int [n2+1];
        for (int i=0; i<n1; i++) {
            L[i] = A[p+i];
        }
        for (int j=0; j<n2; j++) {
            R[j] = A[q+1+j];
        }
        L[n1] = Integer.MAX_VALUE;
        R[n2] = Integer
        int i=0, j=0;
        for (int k=p; k<=r; k++) {
            if (L[i]<=R[j]) {
                A[k] = L[i];
                i++;
            }
            else {
                A[k] = R[j];
                j++
            }   }
    }
    public static void MERGE_SORT (int [] A, int p, int r) {
        if (p<r) {
            int q = (p+r)/2;
            MERGE_SORT (A, p, q);
            MERGE_SORT (A, q+1, r);
    }   } MERGE (A, p, q, r);
    public static void main (String [] args) {
        int[] A = {5, 2, 4, 6, 1, 3};
        int p=0; r=A.length-1;
        MERGE_SORT (A, p, r);
        System.out.println (Arrays.toString(A));
    }
}
```

1) Huffman Coding.

```
package greedy-methods;
import java.util.PriorityQueue;
import java.util.Comparator;
Class Node{
    int items;
    char c;
    Node left;
    Node right;
Class ImplementComparator implements Comparator<Node>{
    public int compare(Node x, Node y){
        return x.item - y.item;
    }
}
public class Huffman_Coding{
    public static void Codes(Node root, String s){
        if(root.left==null && root.right==null &&
            Character.isLetter(root.c)){
            System.out.println(root.c + " : " + s);
            return;
        }
        Codes(root.left, s+"0");
        Codes(root.right, s+"1");
    }
    public static void main(String[] args){
        int n = 6;
        char[] chars = {'a', 'b', 'c', 'd', 'e', 'f'};
        int[] freq = {45, 13, 12, 16, 9, 5};
        PriorityQueue<Node> q = new PriorityQueue<Node>
                (n, new ImplementComparator());
        for(int i=0; i<n; i++){
            Node h= new Node();
            h.c= chars[i];
            h.item = freq[i];
            h.left = null;
            h.right = null;
            q.add(h);
        }
        Node root = null;
        while(q.size() >1){
            Node x = q.peek();
            q.poll();
            Node y= q.peek();
            q.poll();
            Node z= new Node();
            z.item = x.item + y.item;
```

```
        z.c = t.c;
        z.left = x;
        z.right = y;
        root = z;
        pq.add(z);
    }
    System.out.println("Char: Huffman code");
    System.out.println("-----------");
    codes(root, "");
}
```

3.