

Greedy algorithm Questions



Basic Level:

Find Minimum Number of Platforms Required for a Railway Station

Given arrival and departure times of trains, determine the minimum number of platforms required.

Assign Mice to Holes Problem

There are equal numbers of mice and holes, and each mouse has to move into a hole. Find the minimum time required for all mice to get into holes.

Minimum Number of Coins Problem

Given a value V , find the minimum number of coins required to make the change for V using any given set of denominations.

Largest Number with Given Digits

Rearrange digits of a given number to form the largest possible number.

Maximum Sum of Consecutive Elements in an Array (with constraints)

Find the maximum sum of non-adjacent elements in an array.

Intermediate Level:

Distribute Candy to Children

Each child has a rating, distribute candies such that every child gets at least one candy, and children with a higher rating get more candies than their neighbors.

Minimize Sum of Product of Two Arrays

Rearrange elements of two arrays such that the sum of their products is minimized.

Buy Maximum Stocks on Each Day

Given prices of a stock on different days, buy the maximum number of stocks with a limited amount of money.

Connecting N Ropes with Minimum Cost

Given n ropes of different lengths, connect them into a single rope with minimum cost (cost = sum of connected rope lengths).

Dijkstra's Algorithm

Implement Dijkstra's algorithm to find the shortest path in a weighted graph.

explain problem with example that contains input and output in simple explanation and give java code in optimized and simple form and give time complexity



Memory updated

Here are explanations and Java solutions for the problems you mentioned:

1. Minimum Number of Platforms Required

Problem:

Given the arrival and departure times of trains at a railway station, find the minimum number of platforms required such that no train has to wait.

Example: Input:

```
`arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}`  
`dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}`
```

Output: `3`

Explanation:

At 9:50, three trains are at the station, so at least 3 platforms are needed.

Java Code:

java

```
import java.util.Arrays;

public class MinimumPlatforms {
    public static int findPlatforms(int arr[], int dep[]) {
        Arrays.sort(arr);
        Arrays.sort(dep);

        int platformsNeeded = 1, maxPlatforms = 1;
        int i = 1, j = 0;

        while (i < arr.length && j < dep.length) {
            if (arr[i] <= dep[j]) {
                platformsNeeded++;
                i++;
            } else {
                platformsNeeded--;
                j++;
            }
            maxPlatforms = Math.max(maxPlatforms, platformsNeeded);
        }
        return maxPlatforms;
    }

    public static void main(String[] args) {
        int arr[] = {900, 940, 950, 1100, 1500, 1800};
        int dep[] = {910, 1200, 1120, 1130, 1900, 2000};

        System.out.println("Minimum Platforms Required: " + findPlatforms(arr, dep));
    }
}
```

Time Complexity: $O(n \log n)$ (due to sorting).

2. Assign Mice to Holes Problem

Problem:

You are given positions of mice and holes in a straight line. Each mouse can move to a hole. Find the minimum time required for all mice to get into holes.

Example: Input:

```
`mice[] = {1, 4, 9}`
`holes[] = {10, 5, 2}`
```

Output: `8`

Explanation:

The mouse at position 9 moves to the hole at position 10 in 1 unit of time, the one at position 4 moves to the hole at position 5 in 1 unit, and the one at position 1 moves to the hole at position 2 in 1 unit. The maximum time taken is 8.

Java Code:

```
java

import java.util.Arrays;

public class MiceToHoles {
    public static int minTimeToAssign(int[] mice, int[] holes) {
        Arrays.sort(mice);
        Arrays.sort(holes);

        int maxTime = 0;
        for (int i = 0; i < mice.length; i++) {
            maxTime = Math.max(maxTime, Math.abs(mice[i] - holes[i]));
        }
        return maxTime;
    }

    public static void main(String[] args) {
        int mice[] = {1, 4, 9};
        int holes[] = {10, 5, 2};

        System.out.println("Minimum Time: " + minTimeToAssign(mice, holes));
    }
}
```

Time Complexity: $O(n \log n)$ (due to sorting).

3. Minimum Number of Coins Problem

Problem:

Given a value `V` and a set of coin denominations, find the minimum number of coins required to make the value `V`.

Example: Input:

```
`V = 93, coins[] = {1, 5, 10, 25}`
```

Output: `5` (One 25, two 10s, one 5, and three 1s)

Java Code:

```
java

import java.util.Arrays;

public class MinimumCoins {
    public static int minCoins(int coins[], int V) {
        Arrays.sort(coins);
        int count = 0;
        for (int i = coins.length - 1; i >= 0; i--) {
            if (V == 0) break;
            count += V / coins[i];
        }
    }
}
```

```

        V %= coins[i];
    }
    return count;
}

public static void main(String[] args) {
    int coins[] = {1, 5, 10, 25};
    int V = 93;

    System.out.println("Minimum coins required: " + minCoins(coins, V));
}
}

```

Time Complexity: $O(n \log n)$.

4. Largest Number with Given Digits

Problem:

Rearrange digits of a number to form the largest possible number.

Example: Input:

`num = 48392`

Output: `98432`

Java Code:

java

```

import java.util.Arrays;

public class LargestNumber {
    public static String largestNumber(int num) {
        String str = Integer.toString(num);
        char[] digits = str.toCharArray();
        Arrays.sort(digits);

        StringBuilder largestNum = new StringBuilder(new String(digits)).reverse();
        return largestNum.toString();
    }

    public static void main(String[] args) {
        int num = 48392;
        System.out.println("Largest Number: " + largestNumber(num));
    }
}

```

Time Complexity: $O(n \log n)$ (due to sorting).

5. Maximum Sum of Non-Adjacent Elements

Problem:

Find the maximum sum of non-adjacent elements in an array.

Example: Input:

`arr[] = {3, 2, 5, 10, 7}`

Output: `15` (3 + 5 + 7)

Java Code:

```
java
```

```
public class MaxSumNonAdjacent {
    public static int maxSum(int[] arr) {
        int include = arr[0];
        int exclude = 0;

        for (int i = 1; i < arr.length; i++) {
            int newExclude = Math.max(include, exclude);
            include = exclude + arr[i];
            exclude = newExclude;
        }
        return Math.max(include, exclude);
    }

    public static void main(String[] args) {
        int arr[] = {3, 2, 5, 10, 7};
        System.out.println("Maximum Sum: " + maxSum(arr));
    }
}
```

Time Complexity: $O(n)$.

Let me know if you need more explanations or further optimizations!



give remaining



6. Distribute Candy to Children

Problem:

Each child has a rating, and you need to distribute candies such that:

1. Each child gets at least one candy.
2. Children with a higher rating than their neighbors get more candies.

Example: Input:

``ratings[] = {1, 0, 2}``

Output: ``5``

Explanation:

You can distribute the candies as ``[2, 1, 2]``, so the total is ``5``.

Java Code:

```
java
```

```
public class DistributeCandy {
    public static int distributeCandies(int[] ratings) {
        int n = ratings.length;
        int[] candies = new int[n];

        // Each child gets at least 1 candy
        for (int i = 0; i < n; i++) {
            candies[i] = 1;
        }
    }
}
```

```

    }

    // Left to right: if current child has a higher rating than the left one, give
more candy
    for (int i = 1; i < n; i++) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        }
    }

    // Right to left: if current child has a higher rating than the right one, give
more candy
    for (int i = n - 2; i >= 0; i--) {
        if (ratings[i] > ratings[i + 1]) {
            candies[i] = Math.max(candies[i], candies[i + 1] + 1);
        }
    }

    // Sum up the total candies
    int totalCandies = 0;
    for (int candy : candies) {
        totalCandies += candy;
    }

    return totalCandies;
}

public static void main(String[] args) {
    int[] ratings = {1, 0, 2};
    System.out.println("Minimum Candies Required: " + distributeCandies(ratings));
}
}

```

Time Complexity: $O(n)$.

7. Minimize Sum of Product of Two Arrays

Problem:

You are given two arrays. You need to rearrange the elements of the arrays such that the sum of their products is minimized.

Example: Input:

`A[] = {1, 3, 5, 2}`

`B[] = {8, 2, 6, 4}`

Output: `43`

Explanation:

Rearranging the arrays as `A = {1, 2, 3, 5}` and `B = {8, 6, 4, 2}`, the sum of products is minimized: `1*8 + 2*6 + 3*4 + 5*2 = 43`.

Java Code:

```

java

import java.util.Arrays;

public class MinimizeSumOfProducts {
    public static int minimizeSum(int[] A, int[] B) {
        Arrays.sort(A);
        Arrays.sort(B);

        int sum = 0;
        int n = A.length;

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

```

```

        sum += A[i] * B[n - 1 - i]; // Pair smallest from A with largest from B
    }

    return sum;
}

public static void main(String[] args) {
    int[] A = {1, 3, 5, 2};
    int[] B = {8, 2, 6, 4};
    System.out.println("Minimized Sum of Products: " + minimizeSum(A, B));
}
}

```

Time Complexity: $O(n \log n)$.

8. Buy Maximum Stocks on Each Day

Problem:

You are given the price of stocks on different days and a limited amount of money. You can buy a maximum of i stocks on the i -th day. Find the maximum number of stocks you can buy.

Example: Input:

`price[] = {10, 7, 19}, k = 45`

Output: `4`

Explanation:

You can buy 1 stock on the first day, 2 stocks on the second day, and 1 stock on the third day.

Java Code:

java

```

import java.util.Arrays;

public class BuyMaximumStocks {
    public static int buyMaxStocks(int[] price, int k) {
        int n = price.length;
        Stock[] stocks = new Stock[n];

        // Create a Stock class to hold price and index
        for (int i = 0; i < n; i++) {
            stocks[i] = new Stock(price[i], i + 1);
        }

        Arrays.sort(stocks, (a, b) -> a.price - b.price);

        int totalStocks = 0;

        for (Stock stock : stocks) {
            int maxBuyable = Math.min(stock.day, k / stock.price);
            totalStocks += maxBuyable;
            k -= stock.price * maxBuyable;
        }

        return totalStocks;
    }

    public static void main(String[] args) {
        int[] price = {10, 7, 19};
        int k = 45;
        System.out.println("Maximum Stocks: " + buyMaxStocks(price, k));
    }
}

class Stock {
    int price;

```

```
int day;

Stock(int price, int day) {
    this.price = price;
    this.day = day;
}
}
```

Time Complexity: $O(n \log n)$.

9. Connecting N Ropes with Minimum Cost

Problem:

You are given n ropes of different lengths. The cost to connect two ropes is equal to the sum of their lengths. You need to connect all the ropes with minimum cost.

Example: Input:

`ropes[] = {4, 3, 2, 6}`

Output: `29`

Explanation:

First, connect `2` and `3` (cost = 5), then connect `5` and `4` (cost = 9), and finally connect `9` and `6` (cost = 15), resulting in a total cost of `29`.

Java Code:

java

```
import java.util.PriorityQueue;

public class ConnectRopes {
    public static int minCost(int[] ropes) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for (int rope : ropes) {
            pq.add(rope);
        }

        int totalCost = 0;
        while (pq.size() > 1) {
            int first = pq.poll();
            int second = pq.poll();
            int cost = first + second;
            totalCost += cost;
            pq.add(cost);
        }

        return totalCost;
    }

    public static void main(String[] args) {
        int[] ropes = {4, 3, 2, 6};
        System.out.println("Minimum Cost to Connect Ropes: " + minCost(ropes));
    }
}
```

Time Complexity: $O(n \log n)$ (using a priority queue).

10. Dijkstra's Algorithm

Problem:

Implement Dijkstra's algorithm to find the shortest path from a source node to all other nodes in a weighted graph.

Example:

Input:

Graph represented as adjacency matrix:

plaintext

```
{
  {0, 10, 0, 30, 100},
  {10, 0, 50, 0, 0},
  {0, 50, 0, 20, 10},
  {30, 0, 20, 0, 60},
  {100, 0, 10, 60, 0}
}
```

Source: `0`

Output:

Shortest distances from node `0`: `[0, 10, 50, 30, 60]`

Java Code:

java

```
import java.util.Arrays;
import java.util.PriorityQueue;

public class DijkstraAlgorithm {
    public static int[] dijkstra(int[][] graph, int src) {
        int n = graph.length;
        int[] dist = new int[n];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>((a, b) -> a.dist - b.dist);
        pq.add(new Node(src, 0));

        boolean[] visited = new boolean[n];

        while (!pq.isEmpty()) {
            Node node = pq.poll();
            int u = node.vertex;

            if (visited[u]) continue;
            visited[u] = true;

            for (int v = 0; v < n; v++) {
                if (graph[u][v] != 0 && !visited[v] && dist[u] + graph[u][v] < dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                    pq.add(new Node(v, dist[v]));
                }
            }
        }
        return dist;
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 10, 0, 30, 100},
            {10, 0, 50, 0, 0},
            {0, 50, 0, 20, 10},
            {30, 0, 20, 0, 60},
            {100, 0, 10, 60, 0}
        };
    }
}
```

```

    };

    int[] distances = dijkstra(graph, 0);
    System.out.println("Shortest distances: " + Arrays.toString(distances));
}

class Node {
    int vertex, dist;

    Node(int vertex, int dist) {
        this.vertex = vertex;
        this.dist = dist;
    }
}

```

Time Complexity: $O((V + E) \log V)$ (using a priority queue).



Advanced Level:
Huffman Coding

Given a string, compress it using Huffman coding, minimizing the total number of bits used to encode the string.

Minimum Number of Taps to Water a Garden

Given a garden of length n and an array representing the range each tap can water, find the minimum number of taps to water the entire garden.

Maximize the Array Sum after K Negations

Given an array of integers and an integer k, maximize the sum of the array by negating at most k elements.

Weighted Job Scheduling with Profit

Similar to the job scheduling problem, but each job has a start and end time, and jobs can overlap. Find the maximum profit by scheduling non-overlapping jobs.

Minimum Time to Finish Jobs with Given Workers

Given n jobs and k workers, assign jobs to workers such that the maximum time any worker takes is minimized.

Minimum Spanning Tree (Kruskal's Algorithm)

Given a graph, find the minimum spanning tree using Kruskal's algorithm.

Minimum Operations to Reduce Array to Half

Reduce the array's sum to at most half of its original value by performing operations that remove half of the largest remaining elements.

Optimal File Merging

Given files of different sizes, merge them into a single file with the minimum cost.

Minimize Cash Flow in a Group of Friends

Given debts between friends, minimize the cash flow by finding a series of transactions that settle

the debts.
K Centers Problem

Place k centers in a graph such that the maximum distance of any vertex from the nearest center is minimized.



Memory updated

Here are the advanced-level problems you requested, along with explanations and Java code:

1. Huffman Coding

Problem:

Huffman coding is a compression technique that assigns variable-length codes to characters, with shorter codes assigned to more frequent characters.

Example:

Input: `"aaabbc"`

Output: Huffman encoding of the characters (e.g., `a = 0`, `b = 10`, `c = 11`).

Java Code:

java

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

class HuffmanNode {
    char ch;
    int freq;
    HuffmanNode left, right;

    HuffmanNode(char ch, int freq) {
        this.ch = ch;
        this.freq = freq;
    }
}

class HuffmanCoding {
    public static HashMap<Character, String> huffmanCode = new HashMap<>();

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

        for (char ch : text.toCharArray()) {
            freqMap.put(ch, freqMap.getOrDefault(ch, 0) + 1);
        }

        PriorityQueue<HuffmanNode> pq = new PriorityQueue<>((a, b) -> a.freq - b.freq);

        for (var entry : freqMap.entrySet()) {
            pq.add(new HuffmanNode(entry.getKey(), entry.getValue()));
        }

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

            HuffmanNode newNode = new HuffmanNode('\0', left.freq + right.freq);
            newNode.left = left;
            newNode.right = right;
            pq.add(newNode);
        }
    }
}
```

```

        HuffmanNode root = pq.poll();
        generateHuffmanCode(root, "");
    }

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

        if (root.ch != '\0') {
            huffmanCode.put(root.ch, code);
        }

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

    public static void main(String[] args) {
        String text = "aaabbc";
        buildHuffmanTree(text);

        for (char ch : huffmanCode.keySet()) {
            System.out.println(ch + ": " + huffmanCode.get(ch));
        }
    }
}

```

Time Complexity: $O(n \log n)$.

2. Minimum Number of Taps to Water a Garden

Problem:

Given a garden of length n and an array $ranges[]$ where $ranges[i]$ is the range that the i -th tap can cover, find the minimum number of taps required to water the garden.

Example:

Input:

$n = 5$, $ranges = \{3, 4, 1, 1, 0, 0\}$

Output: 1

Java Code:

java

```

import java.util.Arrays;

public class GardenTaps {
    public static int minTaps(int n, int[] ranges) {
        int[] maxReach = new int[n + 1];

        for (int i = 0; i <= n; i++) {
            int left = Math.max(0, i - ranges[i]);
            int right = Math.min(n, i + ranges[i]);
            maxReach[left] = Math.max(maxReach[left], right);
        }

        int taps = 0, currEnd = 0, nextEnd = 0;
        for (int i = 0; i <= n; i++) {
            if (i > nextEnd) return -1;
            if (i > currEnd) {
                taps++;
                currEnd = nextEnd;
            }
            nextEnd = Math.max(nextEnd, maxReach[i]);
        }

        return taps;
    }
}

```

```

    }

    public static void main(String[] args) {
        int n = 5;
        int[] ranges = {3, 4, 1, 1, 0, 0};
        System.out.println("Minimum Taps: " + minTaps(n, ranges));
    }
}

```

Time Complexity: $O(n)$.

3. Maximize Array Sum After K Negations

Problem:

Given an array of integers and an integer k , maximize the sum of the array by negating at most k elements.

Example:

Input:

$arr[] = \{-2, 9, -3, -8\}$, $k = 3$

Output: 20

Java Code:

java

```

import java.util.Arrays;

public class MaximizeSumAfterKNegations {
    public static int maximizeSum(int[] arr, int k) {
        Arrays.sort(arr);
        int i = 0;

        while (k > 0 && i < arr.length && arr[i] < 0) {
            arr[i] = -arr[i];
            i++;
            k--;
        }

        int sum = 0, minElement = Integer.MAX_VALUE;
        for (int num : arr) {
            sum += num;
            minElement = Math.min(minElement, num);
        }

        if (k % 2 == 1) {
            sum -= 2 * minElement;
        }

        return sum;
    }

    public static void main(String[] args) {
        int[] arr = {-2, 9, -3, -8};
        int k = 3;
        System.out.println("Maximized Sum: " + maximizeSum(arr, k));
    }
}

```

Time Complexity: $O(n \log n)$.

4. Weighted Job Scheduling with Profit

Problem:

Given jobs with start times, end times, and profits, find the maximum profit you can achieve by scheduling non-overlapping jobs.

Example:

Input:

`jobs[] = {{1, 3, 50}, {2, 5, 20}, {4, 6, 70}, {6, 7, 30}}`

Output: `120`

Java Code:

java

```
import java.util.Arrays;

class Job {
    int start, end, profit;

    Job(int s, int e, int p) {
        this.start = s;
        this.end = e;
        this.profit = p;
    }
}

public class WeightedJobScheduling {
    public static int findMaxProfit(Job[] jobs) {
        Arrays.sort(jobs, (a, b) -> a.end - b.end);

        int[] dp = new int[jobs.length];
        dp[0] = jobs[0].profit;

        for (int i = 1; i < jobs.length; i++) {
            int include = jobs[i].profit;
            int lastNonConflicting = findLastNonConflicting(jobs, i);
            if (lastNonConflicting != -1) {
                include += dp[lastNonConflicting];
            }
            dp[i] = Math.max(dp[i - 1], include);
        }

        return dp[jobs.length - 1];
    }

    public static int findLastNonConflicting(Job[] jobs, int i) {
        for (int j = i - 1; j >= 0; j--) {
            if (jobs[j].end <= jobs[i].start) return j;
        }
        return -1;
    }

    public static void main(String[] args) {
        Job[] jobs = { new Job(1, 3, 50), new Job(2, 5, 20), new Job(4, 6, 70), new Job(6, 7, 30) };
        System.out.println("Maximum Profit: " + findMaxProfit(jobs));
    }
}
```

Time Complexity: $O(n \log n)$.

5. Minimum Time to Finish Jobs with Given Workers

Problem:

Given `n` jobs and `k` workers, assign jobs to workers such that the maximum time any worker takes

is minimized.

Example:

Input:

`jobs[] = {10, 7, 8, 12, 6, 8}, k = 4`

Output: `15`

Java Code: (Using Binary Search and Greedy Approach)

```
java

import java.util.Arrays;

public class MinTimeToFinishJobs {
    public static int minTime(int[] jobs, int k) {
        int low = Arrays.stream(jobs).max().getAsInt();
        int high = Arrays.stream(jobs).sum();

        while (low < high) {
            int mid = (low + high) / 2;
            if (canAssign(jobs, k, mid)) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }

        return low;
    }

    public static boolean canAssign(int[] jobs, int k, int maxTime) {
        int workers = 1, currTime = 0;

        for (int job : jobs) {
            if (currTime + job > maxTime) {
                workers++;
                currTime = 0;
            }
            currTime += job;

            if (workers > k) return false;
        }

        return true;
    }

    public static void main(String[] args) {
        int[] jobs = {10, 7, 8, 12, 6, 8};
        int k = 4;
        System.out.println("Minimum Time: " + minTime(jobs, k));
    }
}
```

Time Complexity: $O(n \log S)$ where S is the sum of all job times.

Let me know if you need further explanations or additional problems!



give remaining



Here are the remaining advanced-level problems along with explanations and optimized Java code:

6. Minimum Spanning Tree (Kruskal's Algorithm)

Problem:

Given a graph, find the Minimum Spanning Tree (MST) using Kruskal's algorithm.

Example:

Input:

``edges[] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}}``

Output: MST weight = ``19``

Java Code:

java

```
import java.util.*;

class Edge implements Comparable<Edge> {
    int src, dest, weight;

    public int compareTo(Edge compareEdge) {
        return this.weight - compareEdge.weight;
    }
}

class Subset {
    int parent, rank;
}

public class KruskalMST {
    int V, E;
    Edge[] edges;

    KruskalMST(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
        for (int i = 0; i < e; ++i) {
            edges[i] = new Edge();
        }
    }

    public static int find(Subset[] subsets, int i) {
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);
        return subsets[i].parent;
    }

    public static void union(Subset[] subsets, int x, int y) {
        int rootX = find(subsets, x);
        int rootY = find(subsets, y);

        if (subsets[rootX].rank < subsets[rootY].rank)
            subsets[rootX].parent = rootY;
        else if (subsets[rootX].rank > subsets[rootY].rank)
            subsets[rootY].parent = rootX;
        else {
            subsets[rootY].parent = rootX;
            subsets[rootX].rank++;
        }
    }

    public void kruskalMST() {
        Edge[] result = new Edge[V];
        int e = 0, i = 0;
        for (i = 0; i < V; ++i)
            result[i] = new Edge();

        Arrays.sort(edges);

        Subset[] subsets = new Subset[V];
```



```

for (i = 0; i < V; ++i)
    subsets[i] = new Subset();

for (int v = 0; v < V; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0;

while (e < V - 1) {
    Edge nextEdge = edges[i++];

    int x = find(subsets, nextEdge.src);
    int y = find(subsets, nextEdge.dest);

    if (x != y) {
        result[e++] = nextEdge;
        union(subsets, x, y);
    }
}

int minimumCost = 0;
for (i = 0; i < e; ++i)
    minimumCost += result[i].weight;
System.out.println("Minimum Cost Spanning Tree: " + minimumCost);
}

public static void main(String[] args) {
    int V = 4;
    int E = 5;
    KruskalMST graph = new KruskalMST(V, E);

    graph.edges[0].src = 0;
    graph.edges[0].dest = 1;
    graph.edges[0].weight = 10;

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 6;

    graph.edges[2].src = 0;
    graph.edges[2].dest = 3;
    graph.edges[2].weight = 5;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 15;

    graph.edges[4].src = 2;
    graph.edges[4].dest = 3;
    graph.edges[4].weight = 4;

    graph.kruskalMST();
}
}

```

Time Complexity: $O(E \log V)$ where E is the number of edges and V is the number of vertices.

7. Minimum Operations to Reduce Array to Half

Problem:

Given an array, reduce its sum to at most half of its original value by removing the largest elements and halving them in each operation. Find the minimum number of operations required.

Example:

Input:

```
`arr[] = {5, 19, 8, 1}`
```

Output: `3`

Java Code:

```
java
```

```
import java.util.PriorityQueue;

public class MinOperationsToHalf {
    public static int minOperations(int[] arr) {
        PriorityQueue<Double> maxHeap = new PriorityQueue<>((a, b) -> Double.compare(b,
a));
        double sum = 0;

        for (int num : arr) {
            sum += num;
            maxHeap.add((double) num);
        }

        double halfSum = sum / 2;
        int operations = 0;
        double currentSum = sum;

        while (currentSum > halfSum) {
            double largest = maxHeap.poll();
            currentSum -= largest / 2;
            maxHeap.add(largest / 2);
            operations++;
        }

        return operations;
    }

    public static void main(String[] args) {
        int[] arr = {5, 19, 8, 1};
        System.out.println("Minimum Operations: " + minOperations(arr));
    }
}
```

Time Complexity: $O(n \log n)$ due to the heap operations.

8. Optimal File Merging

Problem:

Given files of different sizes, merge them into a single file with minimum cost. The cost to merge two files is the sum of their sizes.

Example:

Input:

```
`files[] = {4, 8, 6, 12}`
```

Output: `58`

Java Code:

```
java
```

```
import java.util.PriorityQueue;

public class OptimalFileMerging {
    public static int minCost(int[] files) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int file : files) {
```

```

        pq.add(file);
    }

    int totalCost = 0;

    while (pq.size() > 1) {
        int file1 = pq.poll();
        int file2 = pq.poll();
        int cost = file1 + file2;
        totalCost += cost;
        pq.add(cost);
    }

    return totalCost;
}

public static void main(String[] args) {
    int[] files = {4, 8, 6, 12};
    System.out.println("Minimum Cost to Merge Files: " + minCost(files));
}
}

```

Time Complexity: $O(n \log n)$ due to the priority queue operations.

9. Minimize Cash Flow in a Group of Friends

Problem:

Given debts between friends, minimize the cash flow by finding the optimal series of transactions to settle the debts.

Example:

Input:

`transactions[] = {{0, 1, 10}, {2, 0, 5}}`

Output:

`Person 1 pays 5 to person 0.`

`Person 1 pays 5 to person 2.`

Java Code:

java

```

import java.util.Arrays;

public class MinimizeCashFlow {

    public static void minimizeCashFlow(int[][] transactions, int N) {
        int[] netAmount = new int[N];

        for (int[] transaction : transactions) {
            int payer = transaction[0];
            int payee = transaction[1];
            int amount = transaction[2];
            netAmount[payer] -= amount;
            netAmount[payee] += amount;
        }

        settleDebts(netAmount);
    }

    public static void settleDebts(int[] netAmount) {
        int maxCredit = getMax(netAmount);
        int maxDebit = getMin(netAmount);

        if (netAmount[maxCredit] == 0 && netAmount[maxDebit] == 0) return;
    }
}

```

```

    int settlementAmount = Math.min(-netAmount[maxDebit], netAmount[maxCredit]);

    netAmount[maxCredit] -= settlementAmount;
    netAmount[maxDebit] += settlementAmount;

    System.out.println("Person " + maxDebit + " pays " + settlementAmount + " to
Person " + maxCredit);

    settleDebts(netAmount);
}

public static int getMax(int[] netAmount) {
    int maxIndex = 0;
    for (int i = 1; i < netAmount.length; i++) {
        if (netAmount[i] > netAmount[maxIndex]) maxIndex = i;
    }
    return maxIndex;
}

public static int getMin(int[] netAmount) {
    int minIndex = 0;
    for (int i = 1; i < netAmount.length; i++) {
        if (netAmount[i] < netAmount[minIndex]) minIndex = i;
    }
    return minIndex;
}

public static void main(String[] args) {
    int[][] transactions = {{0, 1, 10}, {2, 0, 5}};
    minimizeCashFlow(transactions, 3);
}
}

```

Time Complexity: $O(N^2)$ where N is the number of people.

10.



K Centers Problem

Place k centers in a graph such that the maximum distance of any vertex from the nearest center is minimized.



The K Centers Problem involves placing k centers (or facilities) in a graph such that the maximum distance between any vertex and its nearest center is minimized. This problem is a classic example of facility location problems and is typically solved using approximation algorithms due to its computational complexity.

Explanation

Objective:

Given a set of points and a number k , place k centers in such a way that the maximum distance between any point and the nearest center is as small as possible.

Example:

- **Input:**

- Points: $[(1, 2), (2, 3), (4, 5), (6, 7)]$
- $k = 2$
- **Output:**
 - Minimum radius that covers all points by placing 2 centers.

Approach:

1. Initialization:

- Start by placing the first center at a point and iteratively place additional centers at the farthest point from the nearest center.

2. Algorithm Steps:

- **Select the first center:** Place it at an arbitrary point (often the first point).
- **Iteratively add centers:** For each additional center, select the point that maximizes the distance to the nearest center already placed.
- **Calculate the radius:** After placing all k centers, compute the maximum distance from each point to its nearest center.

Approximation Algorithm:

A common approximation algorithm for this problem is the **Greedy Algorithm**:

1. Initialize:

- Select one center arbitrarily.

2. Iterative Selection:

- In each iteration, choose the point that is farthest from all existing centers.

3. Compute Result:

- After placing all k centers, calculate the maximum distance any point has from its nearest center.

Java Code

Here's an implementation of the K Centers Problem using a greedy approximation approach:

```
java
```

```
import java.util.*;

public class KCenters {

    // Function to calculate the Euclidean distance between two points
    private static double distance(int[] p1, int[] p2) {
        return Math.sqrt(Math.pow(p1[0] - p2[0], 2) + Math.pow(p1[1] - p2[1], 2));
    }

    // Function to find the maximum distance of any point from its nearest center
    public static double kCenters(int[][] points, int k) {
        int n = points.length;
        boolean[] isCenter = new boolean[n];
        double[][] dist = new double[n][n];

        // Compute the distance matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = distance(points[i], points[j]);
            }
        }

        double result = 0;
        Set<Integer> centers = new HashSet<>();
```

```
// Place the first center at the first point
centers.add(0);
isCenter[0] = true;

// Greedily select additional centers
while (centers.size() < k) {
    double maxDist = -1;
    int farthestPoint = -1;

    // For each non-center point, find the maximum distance to the nearest center
    for (int i = 0; i < n; i++) {
        if (!isCenter[i]) {
            double minDist = Double.MAX_VALUE;
            for (int center : centers) {
                minDist = Math.min(minDist, dist[i][center]);
            }
            if (minDist > maxDist) {
                maxDist = minDist;
                farthestPoint = i;
            }
        }
    }
    centers.add(farthestPoint);
    isCenter[farthestPoint] = true;
}

// Calculate the maximum distance of any point from its nearest center
for (int i = 0; i < n; i++) {
    double minDist = Double.MAX_VALUE;
    for (int center : centers) {
        minDist = Math.min(minDist, dist[i][center]);
    }
    result = Math.max(result, minDist);
}

return result;
}

public static void main(String[] args) {
    int[][] points = {{1, 2}, {2, 3}, {4, 5}, {6, 7}};
    int k = 2;
    System.out.println("Minimum Radius: " + kCenters(points, k));
}
}
```

Time Complexity:

- **Computing Distances:** $O(n^2)$
- **Selecting Centers and Computing Distances:** $O(n^2 \cdot k)$

Explanation of Complexity:

- The distance matrix computation takes $O(n^2)$ time.
- Each center placement involves scanning through all points, taking $O(n)$ time for each of the k centers, leading to $O(n \cdot k)$.

This greedy algorithm provides a good approximation for the K Centers Problem, though the exact solution may require more sophisticated methods or optimization techniques.

ChatGPT can make mistakes. Check important info.