Title: Activity Selection Problem

Description:
You are given n activities with start and end times. Select the maximum number of activities that can be performed by a single person, as

Input Format:

```
n
start1 end1
start2 end2
...
startn endn
```

Output Format:
Maximum number of non-overlapping activities

Sample Input:

```
6
1 3
2 4
3 5
0 6
5 7
8 9
```

Sample Output:
4

Greedy Approach:
Sort activities by end time. Always pick the next activity that starts after the last selected activity ends.

Related LeetCode Link:
https://leetcode.com/problems/maximum-length-of-pair-chain/

Java Template Code:

```java
import java.util.*;

class Activity {
    int start, end;
    Activity(int s, int e) {
        start = s;
        end = e;
    }
}

public class ActivitySelection {
    public static void main(String[] args) {
        Activity[] activities = {
            new Activity(1, 3),
            new Activity(2, 4),
            new Activity(3, 5),
            new Activity(0, 6),
            new Activity(5, 7),
            new Activity(8, 9)
        };

        Arrays.sort(activities, Comparator.comparingInt(a -> a.end));

        int count = 1;
        int lastEnd = activities[0].end;

        for (int i = 1; i < activities.length; i++) {
            if (activities[i].start >= lastEnd) {
                count++;
                lastEnd = activities[i].end;
            }
        }
```

Title: Fractional Knapsack Problem

Description:
Given n items with values and weights, and a knapsack with capacity W, find the maximum value you can obtain. You can take fractions

Input Format:
n W
value1 weight1
value2 weight2
...
valuen weightn

Output Format:
Maximum value (2 decimal places)

Sample Input:
3 50
60 10
100 20
120 30

Sample Output:
240.00

Greedy Approach:
Sort items by value/weight ratio. Take as much as possible from the highest ratio item.

Related LeetCode Link:
https://leetcode.com/problems/maximum-units-on-a-truck/

Java Template Code:

```java
import java.util.*;

class Item {
    int value, weight;
    Item(int v, int w) {
        value = v;
        weight = w;
    }
}

public class FractionalKnapsack {
    public static void main(String[] args) {
        Item[] items = {
            new Item(60, 10),
            new Item(100, 20),
            new Item(120, 30)
        };
        int capacity = 50;

        Arrays.sort(items, (a, b) -> Double.compare((double)b.value/b.weight, (double)a.value/a.weight));

        double totalValue = 0.0;
        for (Item item : items) {
            if (capacity >= item.weight) {
                capacity -= item.weight;
                totalValue += item.value;
            } else {
                totalValue += item.value * ((double)capacity / item.weight);
                break;
            }
        }

        System.out.printf("Maximum value: %.2f\n", totalValue);
    }
```

Title: Minimum Number of Coins

Description:
Given an amount and coin denominations, find the minimum number of coins needed to make the amount.

Input Format:
amount
denominations (space-separated)

Output Format:
Minimum number of coins

Sample Input:
93
1 2 5 10 20 50 100

Sample Output:
4

Greedy Approach:
Sort denominations in descending order. Pick the largest denomination possible until the amount is zero.

Related LeetCode Link:
https://leetcode.com/problems/coin-change/

Java Template Code:

```java
import java.util.*;

public class MinCoins {
    public static void main(String[] args) {
        int amount = 93;
        int[] coins = {1, 2, 5, 10, 20, 50, 100};
        Arrays.sort(coins);
        int count = 0;

        for (int i = coins.length - 1; i >= 0; i--) {
            while (amount >= coins[i]) {
                amount -= coins[i];
                count++;
            }
        }

        System.out.println("Minimum coins needed: " + count);
    }
}
```

Title: Job Sequencing Problem

Description:
Given n jobs with deadlines and profits, schedule jobs to maximize total profit. Each job takes 1 unit of time.

Input Format:

```
n
job_id1 deadline1 profit1
job_id2 deadline2 profit2
...
job_idn deadlinen profitn
```

Output Format:
Maximum total profit

Sample Input:

```
4
a 4 20
b 1 10
c 1 40
d 1 30
```

Sample Output:

```
60
```

Greedy Approach:
Sort jobs by profit. Schedule each job to the latest available slot before its deadline.

Related LeetCode Link:
https://leetcode.com/problems/maximum-profit-in-job-scheduling/

Java Template Code:

```
import java.util.*;

class Job {
    String id;
    int deadline, profit;
    Job(String i, int d, int p) {
        id = i;
        deadline = d;
        profit = p;
    }
}

public class JobSequencing {
    public static void main(String[] args) {
        Job[] jobs = {
            new Job("a", 4, 20),
            new Job("b", 1, 10),
            new Job("c", 1, 40),
            new Job("d", 1, 30)
        };

        Arrays.sort(jobs, (a, b) -> b.profit - a.profit);

        int maxDeadline = Arrays.stream(jobs).mapToInt(j -> j.deadline).max().getAsInt();
        boolean[] slots = new boolean[maxDeadline + 1];
        int totalProfit = 0;

        for (Job job : jobs) {
            for (int j = job.deadline; j > 0; j--) {
                if (!slots[j]) {
                    slots[j] = true;
                    totalProfit += job.profit;
                    break;
```

Title: Minimum Refueling Stops

Description:
You are driving to a destination target km away with startFuel. Given gas stations (distance, fuel), find the minimum number of refueling s

Input Format:

```
target startFuel
n
station1_distance station1_fuel
...
stationn_distance stationn_fuel
```

Output Format:
Minimum number of refueling stops or -1 if not possible

Sample Input:

```
100 10
4
10 60
20 30
30 30
60 40
```

Sample Output:

```
2
```

Greedy Approach:
Use a max-heap to store fuel from stations passed. Refuel from the station with the most fuel when needed.

Related LeetCode Link:
https://leetcode.com/problems/minimum-number-of-refueling-stops/

Java Template Code:

```
import java.util.*;

public class MinRefuelingStops {
    public static void main(String[] args) {
        int target = 100, startFuel = 10;
        int[][] stations = {{10, 60}, {20, 30}, {30, 30}, {60, 40}};

        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        int fuel = startFuel, stops = 0, i = 0;

        while (fuel < target) {
            while (i < stations.length && stations[i][0] <= fuel) {
                maxHeap.add(stations[i][1]);
                i++;
            }

            if (maxHeap.isEmpty()) {
                System.out.println("-1");
                return;
            }

            fuel += maxHeap.poll();
            stops++;
        }

        System.out.println("Minimum refueling stops: " + stops);
    }
}
```