✅ Greedy Algorithms

✅ Greedy Algorithms – Greedy Florist

📌 Problem Statement:

You and your **k** friends want to buy **n** flowers. Each flower has a price. But the florist has a rule:

- If someone has already bought **x** flowers, the **next flower** they buy will cost `price * (x + 1)`.

Your goal is to **minimize the total cost** of buying all flowers.

📥 Example:

java

CopyEdit

`Input:n = 3, k = 3c = [2, 5, 6]Output: 13Explanation:Each person buys one flower.Cost = 6 + 5 + 2 = 13`

🔁 Brute Force (Inefficient):

- Try all ways to assign flowers to people.
- **Time Complexity**: Factorial — not feasible.

🧠 Greedy Intuition:

- Buy **expensive flowers first**.
- Distribute purchases in **round-robin fashion**, keeping track of how many each person has bought.

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(1)

```java
import java.util.*;

public class GreedyFlorist {
    public static int getMinimumCost(int k, int[] c) {
        Arrays.sort(c); // Sort prices in ascending
        int n = c.length;
        int cost = 0;
        int flowerBought = 0;
```

```
        for (int i = n - 1; i >= 0; i--) {
            int round = flowerBought / k;
            cost += c[i] * (round + 1);
            flowerBought++;
        }

        return cost;
    }
}
```

Input:
k = 2, c = [2, 5, 6]

Output: 15
Explanation:
First person buys 6 (cost: 6×1), then second person buys 5 (5×1), then first person buys 2 (2×2). Total = 6 + 5 + 4 = 15

✅ Greedy Algorithms – Marc's Cakewalk

📌 Problem Statement:

Marc loves cupcakes, but to stay healthy, he follows a rule:

- If he eats the i-th cupcake in order of **decreasing calorie count**, the calorie gain is: `calories[i] * (2^i)`

You must help him **minimize** total calorie gain.

📥 Example:

```
Input:
calories = [5, 10, 7]

Output: 79

Explanation:
Eat in order: 10, 7, 5
Calories: 10×2⁰ + 7×2¹ + 5×2² = 10 + 14 + 40 = 64
```

🔁 Brute Force:

- Try all permutations to find the best order.
- **Time Complexity**: O(n!)

🧠 Greedy Intuition:

- Always eat the **highest calorie** cupcake first → minimum multiplier.

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(1)

```java
import java.util.*;

public class MarcsCakewalk {
    public static long marcsCakewalk(int[] calories) {
        Arrays.sort(calories); // Sort in ascending
        long totalCalories = 0;
        int n = calories.length;

        for (int i = 0; i < n; i++) {
            totalCalories += (long) calories[n - 1 - i] * (1L << i); // 2^i
        }

        return totalCalories;
    }
}
```

✅ Output

```
Input: [1, 3, 2]
Output: 11
Explanation: Eat 3×1 + 2×2 + 1×4 = 3 + 4 + 4 = 11
```

✅ Greedy Algorithms – Luck Balance

📌 Problem Statement:

Lena loves to compete in contests. Each contest has:

- A luck value `L`
- An importance flag `T` (1 = important, 0 = unimportant)

She can lose any **unimportant** contest.

She can **lose at most** `k` **important contests**.
Losing a contest **adds** its luck to her score.
Winning a contest **subtracts** its luck from her score.

> **Goal:** Maximize total luck balance.

📥 Example:

```
Input:
k = 2
contests = [[5, 1], [1, 1], [4, 0]]

Output: 10

Explanation:
- Lose 5 and 4 (luck += 9)
- Win 1 (luck -= 1)
Total: 5 + 4 - 1 = 8
```

🔁 Brute Force:

- Try all combinations of contests to win/lose.
- **Time Complexity**: Exponential

🧠 Greedy Intuition:

- Always lose unimportant contests.
- For important ones:
  - Sort by highest luck.
  - Lose top `k`, win the rest.

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(n)

```java
import java.util.*;

public class LuckBalance {
    public static int luckBalance(int k, int[][] contests) {
        List<Integer> important = new ArrayList<>();
        int totalLuck = 0;

        for (int[] contest : contests) {
            int luck = contest[0];
            int type = contest[1];

            if (type == 0) {
```

```
                totalLuck += luck; // Always lose unimportant contests
            } else {
                important.add(luck); // Store important ones
            }
        }

        // Sort important contests descending
        Collections.sort(important, Collections.reverseOrder());

        // Lose top k important
        for (int i = 0; i < important.size(); i++) {
            if (i < k)
                totalLuck += important.get(i); // Lose
            else
                totalLuck -= important.get(i); // Win
        }

        return totalLuck;
    }
}
```

✅ Output

```
Input:
k = 2, contests = [[5, 1], [2, 1], [1, 1], [8, 0]]

Output: 18
Explanation:
Lose 5 and 2 → +7
Win 1 → -1
Lose 8 (unimportant) → +8
Total: 14
```

✅ Greedy Algorithms – Priyanka and Toys

📌 Problem Statement:

Priyanka wants to buy toys.
Each toy has a **weight**.
She can buy all toys within a range of `w to w + 4` in a **single container**.
She wants to **minimize the number of containers** used.

📥 Example:

```
Input:
weights = [1, 2, 3, 17, 10]

Output: 3

Explanation:
- Container 1: [1, 2, 3]
- Container 2: [10]
- Container 3: [17]
```

🔁 Brute Force:

- Try placing each toy in every container.
- **Time Complexity**: O(2^n)

🧠 Greedy Intuition:

- **Sort** the weights.
- Start from smallest.
- Put as many toys as possible within 4 units of that one.
- Move to the next unplaced toy.

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(1)

```java
import java.util.*;

public class PriyankaAndToys {
    public static int toys(int[] weights) {
        Arrays.sort(weights);
        int containers = 0;
        int i = 0;
        int n = weights.length;

        while (i < n) {
            int limit = weights[i] + 4; // Current container range
            containers++;

            // Skip all toys within range
            while (i < n && weights[i] <= limit) {
                i++;
            }
        }
```

```
        return containers;
    }
}
```

✅ Output

```
Input: [1, 2, 3, 21, 22, 23, 24, 25]
Output: 2
Explanation:
- Container 1: [1, 2, 3]
- Container 2: [21–25]
```

✅ Greedy Algorithms – Beautiful Pairs

📌 Problem Statement:

You are given two arrays `A` and `B` of the same length. A *beautiful pair* is an element that exists in **both** arrays (can be matched once).

However, you are **allowed to change one element in** `B` to **any integer**.

> **Goal:** Maximize the number of beautiful pairs after this one change.

📥 Example:

```
Input:
A = [1, 2, 3, 4]
B = [1, 2, 3, 3]

Output: 4

Explanation:
- Initial pairs: [1, 2, 3] → 3 matches
- Modify B[3] = 4 → Add 4th match
```

🔁 Brute Force:

- Try all modifications in `B` and count matches.
- **Time Complexity**: $O(n^2)$

🧠 Greedy Intuition:

- Count frequency of each element in both arrays.
- Find initial matches using min(A[i], B[i])
- Then:

- If total matches < A.length → We can **increase** by 1
- If total matches == A.length → One change will **break** a match → decrease by 1

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

```java
import java.util.*;

public class BeautifulPairs {
    public static int beautifulPairs(int[] A, int[] B) {
        Map<Integer, Integer> freqA = new HashMap<>();
        Map<Integer, Integer> freqB = new HashMap<>();

        for (int a : A) freqA.put(a, freqA.getOrDefault(a, 0) + 1);
        for (int b : B) freqB.put(b, freqB.getOrDefault(b, 0) + 1);

        int matches = 0;

        for (int key : freqA.keySet()) {
            if (freqB.containsKey(key)) {
                matches += Math.min(freqA.get(key), freqB.get(key));
            }
        }

        // We are allowed to change 1 element in B
        if (matches == A.length) return matches - 1;
        else return matches + 1;
    }
}
```

✅ Output

```
Input:
A = [1, 1, 2, 2]
B = [1, 2, 2, 3]

Output: 4
Explanation:
- Initial matches: 3
- Change 3 → 1 → Now 4 matches
```

✅ Greedy Algorithms – Maximum Perimeter Triangle

📌 Problem Statement:

Given an array of stick lengths, choose **3 of them** to form a triangle with **maximum perimeter**.
If more than one triangle has the same perimeter, pick the one with the **longest maximum side**.
If no triangle is possible, return `-1`.

> Triangle Rule: The sum of any two sides must be **greater than the third**.

📥 Example:

```
Input:
sticks = [1, 1, 1, 3, 3]

Output:
[1, 3, 3]

Explanation:
- Possible triangles: (1, 1, 1), (1, 3, 3)
- Max perimeter = 7 from (1, 3, 3)
```

🔁 Brute Force:

- Try all combinations of 3 sticks.
- Check if triangle is valid.
- Track max perimeter.
- **Time Complexity**: $O(n^3)$

🧠 Greedy Intuition:

- Sort array in **descending** order.
- Try consecutive triplets:
  - Since sorted, if `a < b + c`, then triangle is valid.
- First such triplet = largest possible perimeter.

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(1)

```java
import java.util.*;

public class MaxPerimeterTriangle {
    public static List<Integer> maximumPerimeterTriangle(int[] sticks) {
```

```
        Arrays.sort(sticks);
        int n = sticks.length;

        for (int i = n - 1; i >= 2; i--) {
            int a = sticks[i - 2];
            int b = sticks[i - 1];
            int c = sticks[i];

            if (a + b > c) {
                return Arrays.asList(a, b, c);
            }
        }

        return Arrays.asList(-1);
    }
}
```

✅ Output

```
Input: [1, 2, 3, 4, 5, 10]
Output: [3, 4, 5]

Input: [1, 1, 1, 2, 3, 5]
Output: [1, 1, 1]
```

✅ Greedy Algorithms – Largest Permutation

📌 Problem Statement:

You are given an array `arr` of size `n` and a number `k`.
You can **swap any two elements** at most `k` times.

Your goal is to make the **largest lexicographical permutation** possible in **at most** `k` swaps.

📥 Example:

```
Input:
arr = [4, 2, 3, 5, 1]
k = 1

Output:
[5, 2, 3, 4, 1]

Explanation:
```

```
- Only one swap allowed.
- Swap 4 and 5 to get largest possible.
```

🔁 Brute Force:

- Try all combinations of up to `k` swaps.
- Compute and compare all permutations.
- **Time Complexity**: O(n! * k)

🧠 Greedy Intuition:

- To get the largest permutation:
  - Put the largest number at position 0.
  - Then second largest at position 1, and so on…
- For each position `i`:
  - If current element is not the correct (largest possible):
    - Find its correct position and swap.
    - Decrease `k` by 1
    - Stop if `k == 0`

🚀 Optimal Greedy Solution

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

```java
import java.util.*;

public class LargestPermutation {
    public static int[] largestPermutation(int k, int[] arr) {
        int n = arr.length;

        // Map each number to its index for O(1) lookup
        Map<Integer, Integer> pos = new HashMap<>();
        for (int i = 0; i < n; i++) {
            pos.put(arr[i], i);
        }

        for (int i = 0; i < n && k > 0; i++) {
            int expected = n - i;

            if (arr[i] != expected) {
                int indexToSwap = pos.get(expected);
```

```
                // Swap in array
                pos.put(arr[i], indexToSwap);
                pos.put(expected, i);

                int temp = arr[i];
                arr[i] = arr[indexToSwap];
                arr[indexToSwap] = temp;

                k--;
            }
        }

        return arr;
    }
}
```

✅ Output

```
Input:
arr = [4, 2, 3, 5, 1], k = 1
Output: [5, 2, 3, 4, 1]

Input:
arr = [2, 1, 3], k = 1
Output: [3, 1, 2]
```